Design, formalization and realization of Harmonic Box Coordination Language: an externally timed specification substrate for arbitrarily reliable distributed systems

Samuel R. J. George

A dissertation submitted to the University of Bristol in accordance with the
requirements for award of the degree of Doctor of Philosophy in the
Faculty of Engineering
Merchant Venturers School
June 2014

Word count: 73,603

Abstract

The functional specifications of high integrity systems include details needed to harden them against hardware and implementation failures, leading to a lack of design transparency, duplicative certification exercises and implementation inflexibility. This thesis develops a new ontology-driven meta-model for specifying such systems, in which the language itself is instantiated over a canonical coordinate system in spacetime.

We define a system of localized timed types, denoted by a canonical tree of identifiers, and a dense model of time, which we call Pre-HBCL (Pre-Harmonic Box Coordination Language), for which we give a deep embedding in the Coq proof assistant. We go on to develop a full coordination programming language of harmonic boxes (full HBCL), defined over these types; the language is parametrized on arbitrary inner box languages, and we provide a simple example that gives rise to a hardware description language. We give a full semantics at progressive levels of formality. We argue that the decoupling of a light-weight ontology from formalizing logic language produces a more flexible approach than can be achieved with monolithic specification languages, allowing bisimulation properties to be established between HBCL programs and more directly physical axiomatizations of hardware. We demonstrate how the use of a reflexive morphism in HBCL simplifies the establishment of timing properties and composition of specifications.

We develop an interpreter exported from a proof assistant. In doing so, we demonstrate an unusually easy route to establishing soundness of our language up to the soundness of the formalizing logic, and discuss the completeness limitations. To this end, we review issues in computability, and construct a comparison in which formal logics are included in the same schema as coordination and specification languages. We conclude with an empirical demonstration of properties derived from the interpreter, and evaluate the extent to which the work substantiates our conjectures.

Acknowledgements

I gratefully acknowledge the advice and support of my supervisors in completing this work. My thanks go especially to Prof. Greg Michaelson, who has tirelessly advised and encouraged me to the conclusion of this thesis. I am thankful also to Dr. Ian Holyer for overseeing matters at Bristol, and grateful to Prof. Dhiraj Pradhan for his help in the initial stages of my doctoral study. I would also like to thank Dr. Steve Gregory, my progress reviewer, for his useful comments.

I wish to thank the University of Bristol for the award of a Centenary Postgraduate Research Scholarship, which contributed to making this work possible.

To my parents

Declaration

Contents

List of Figures

List of Tables

Chapter 1

Introduction and motivation

This thesis introduces the Harmonic Box Coordination Language (HBCL), which is a *coordination language*. Coordination languages share much in common with concurrent languages, and both are more expressive of parallel behaviours than many traditional programming languages such as C. In contrast with sequential languages, which usually deal with concurrency (if they deal with it at all) by importing the indeterminable scheduling semantics of thread-spawning system calls, coordination and concurrent languages formalize concurrency within the semantics of a programming language itself.

1.1 Motivating a language

There is no shortage of programming or specification languages, and so we must make the case for proposing a new one. The aim of HBCL is to provide a convenient method for specifying arbitrarily complex concurrent programs that are amenable to semi-automatic reflexive transformation into other HBCL programs, but which witness the semantics of the original program under a state space *interpretation function* when mapped to defined hardware configurations. These hardware configurations may be in a degraded condition, which the transformed program, through its interpretation function, might be specified to tolerate. These concepts are familiar from fault-tolerant hardware design, but their application to high-level languages is a novelty.

Definition: **Interpretation function**

An injective mapping between two system histories, which can be defined with a bisimulation predicate. Usually, we are interested in showing that a simpler and more intuitive system has all of its states witnessed by a more complex, and uglier, implementation.

1.2 Hypothesis

In this work, we set out to show that:

1. The specification of deterministic functions over absolute time and an ontology of harmonically timed types in a space-time coordinate system provides a precise description of correctness in real-world distributed computation, and gives rise to simpler composition and transformation semantics than is achievable with the partial orders of an asynchronous or locally clocked model.

2. Deeply embedding the axiomatization of an ontologically aware language in the formal logic of a proof assistant allows statements to be made about how the semantics of that language relate to other formalisms (such as a physical description of hardware) expressed in the same logic. This class of statement is larger than that which may be stated in the toolkit of a monolithic specification environment.

3. It is easier to establish the soundness and completeness of a deep embedding of a coordination language by constructing a proof-carrying interpreter that derives its properties from the proof assistant's logic than by producing an *ex post facto* proof over a predicate characterization of the language.

1.3 Contributions

We make the following contributions. The first six are directly in support of our three hypotheses:

- **Ontology of global clocking and coordinate system** Axiomatic specifications of computation (hypothesis 1).

- **Ontology observation semantics** The axes of understanding of Pre-HBCL (hypothesis 1).

- **Dependent type-theoretical semantics** Typed reality and type of interpreters; linkage of predicate and operational definitions by functions to parametrized σ-types (hypothesis 2).

- **Polarization of specification languages into formal logics and ontology** Demonstration that design space of specification toolkits can be covered by formal logics and deep embeddings of ontologies (hypothesis 2).

- **Formalization in a proof assistant** Predicate and operational characterization of full HBCL in Coq (hypothesis 3).

- **Working simulation and proof** Demonstration of simple route to establishing formal properties; working simulation gives conviction to structure of coinductive bisimilarity proof structure and empirical evidence of correctness of interpreter function (hypothesis 3).

The other contributions are useful innovations that arose in solving problems:

- **Separation of coordination and expression constructions** Box languages as parameters.
- **Formalization of FIFOs** Communication by absolutely timed and deterministic clocked FIFOS.
- **Agents and entropy** Clear boundary between determinism of model and unified treatment of agents and entropy as external influences *via* input streams.
- **Instance and library closure semantics** All resolution semantics independent of a root namespace.

1.4 Thesis structure

In chapter 2 we review what a coordination language is, and how it fits within general models of computation. The difference between programming languages and specification languages is examined, and from this we look into issues of what it means for a program to be *correct*. Further, we look at equivalence classes of various types of sequential and parallel languages, their expressivity and limitations on the ability of some classes to simulate others. At an early stage, a distinction is drawn between synchronous and asynchronous languages. In particular, care is taken to examine those languages that have an especially similar structure or application focus to HBCL. HBCL belongs to the family of *synchronous* coordination languages. We discuss what this means in detail, and go on to make a further definition and distinction between *internally timed* and *externally timed* synchronous languages. In this chapter, we also examine how languages are embedded in the formulæ of logics that can be mechanistically validated as such by computer programs. The chapter covers what it means to embed languages as formulæ in those logics, and goes on to review deductive systems for reasoning about these logics. In particular, it probes the limiting factors of confidence in such systems, and allows us to calibrate our aspirations for the formalization of HBCL against what is practically possible with available tools.

Chapter 3 gives an intuitive overview of HBCL and review of timing issues, followed by a set of progressive examples that is illustrative of the structures that can be represented in HBCL (or, indeed, any general purpose coordination language).

Chapter 4 goes on to define and present an abstraction of the formal syntax and semantics in the form of a predicate-enriched semantic domain, and points to the appendices where detailed operational rules can be found.

Chapter 5 introduces the module-functorial structure of HBCL and provides a detailed formalization in Coq. The details of the formalization of the coordination language and expression language are presented in section 5.5 and section 5.6. The first part of section 5.5 deals with the formalization of the coordination language, illustrated by salient extracts from the Coq sources. This is followed by a detailed description of the coordination part of the concrete interpreter, again accompanied by suitable extracts from the Coq code. This pattern is repeated for the example expression language in section 5.6.

Chapter 6 presents the results of running the simple examples discussed in chapter 3 through the interpreter. It goes on to introduce a further set of more realistic progressive examples that more fully test the scalability of the language, and reviews the kind of statements that can be made in Coq about HBCL specifications.

Chapter 7 concludes the thesis, including, in section 7.3, a critical evaluation of what has been presented. We also assess the potential that arises for further work.

Figure 1.1 shows this outline pictorially, with dependencies between different chapters. An arrow pointing to a particular chapter indicates that it benefits from being read after that from which the arrow emanates. The arrows should be read transitively. Multiple paths through the diagram exist: a shorter path indicates a workable progression to the final chapter, but reading only these chapters will result in an incomplete picture. A chapter to which multiple arrows point has multiple dependencies.

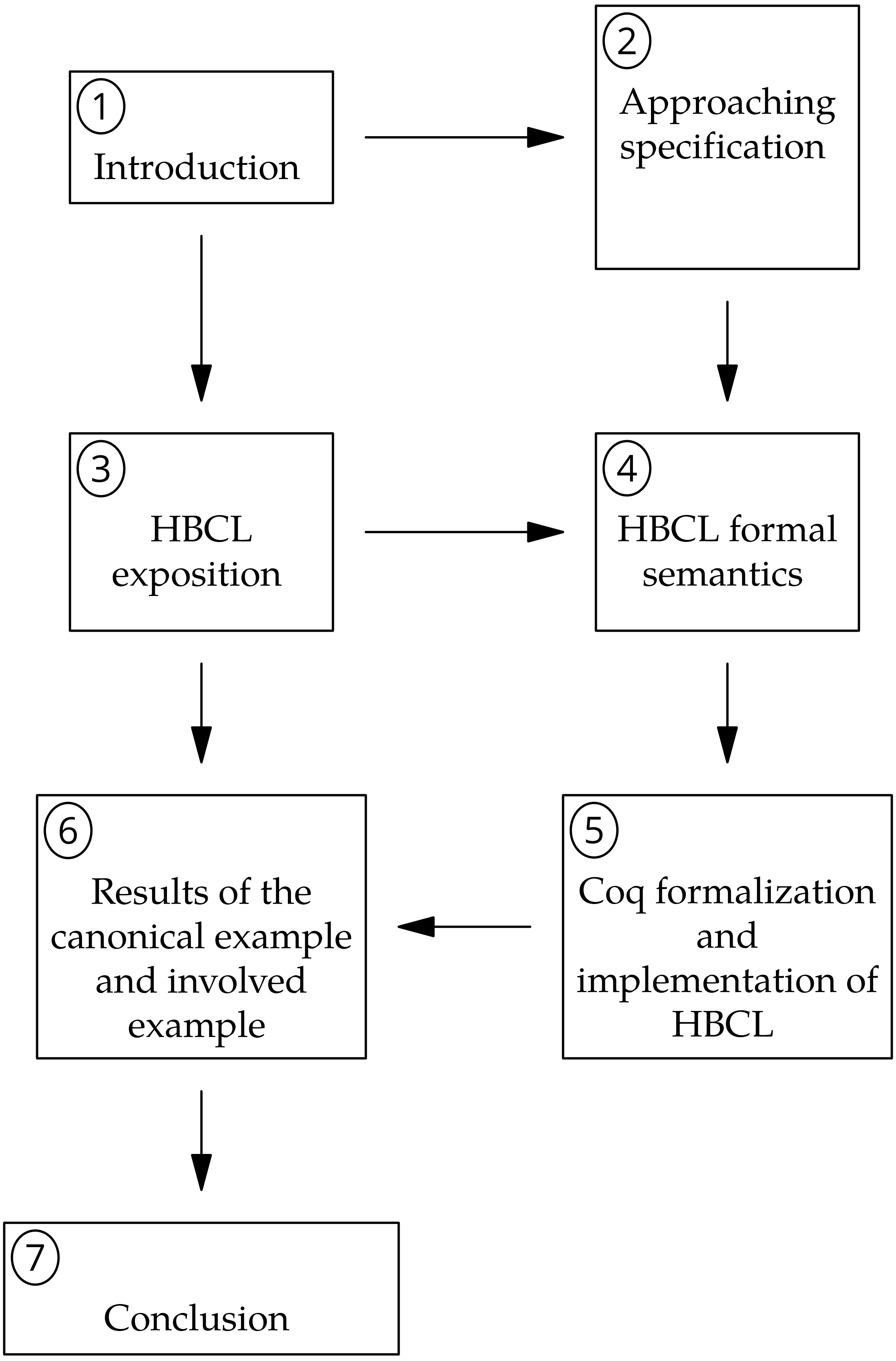

1
Introduction
2
Approaching specification
3
HBCL exposition
4
HBCL formal semantics
6
Results of the canonical example and involved example
5
Coq formalization and implementation of HBCL
7
Conclusion

Figure 1.1: Thesis roadmap

Chapter 2

Specification: computation, coordination and certainty

This chapter discusses an apparently wide range of topics: sequential and coordination languages, specification languages and proof tools. The present work is concerned with formalizing a coordination language in a proof tool, but to see why one would want to do this, and to contrast it with other approaches, we must examine the substantial overlaps among the issues covered here. This is brought together at the end of the chapter with a table of comparison, which we use to motivate and differentiate our approach in the remainder of the thesis.

2.1 Languages, automata and semantic models

In order to say anything useful about coordination languages as programming languages, we need a working definition of both.

Informally, a programming language describes logical systems that can be implemented in the physical world as a sequence of system states. The language is executable if it can be physically represented and initialized in such a way that the laws of physics alone produce an evolution of states through time that is consistent with the semantics of the language. Automata theory is the discipline of studying the abstract properties of such machines, and a Turing machine is the canonical species of automaton that can perform general computational tasks [186]. The first modern computers were the earliest physical analogues of the archetypal Turing machine, although the actual architectures used were based on von Neumann random-access models [190].[1] All buildable computers are finite and therefore imperfect.

It is also helpful, for the purposes of this chapter, to think of a coordination (or indeed any programming) language as a formal language, or at least two formal languages in one. The static language defines the valid input strings of the language as it is writ-

[1]Charles Babbage's analytical engine [138] is the first design for a Turing-equivalent machine.

ten down. Frequently this comes as two further languages, a language accepted by a parser (embodying the syntactic rules, often as a context-free language so that tractable parsers can be written) and a language accepted by the compiler, which embodies the static semantics restricting the valid parse trees generated by the first language[2]. The strings accepted by the compiler can be thought of as instantiating a final automaton (in the case of a stream-based coordination language, an ω-automaton[3]), which is the language of input streams to dynamic executions. This final automaton, in its generalized form as quantified over all possible static-semantically valid programs, can also usefully be thought of as an abstract machine: one that is capable of simulating a Turing machine if the language is Turing-complete. It is the dynamic semantics of a language, as characterized by its abstract machine, which define the useful class of languages that is generally loosely referred to as 'programming languages': this is the most interesting part of a programming language, and the only element of such languages that will be discussed in this chapter.

Coordination languages differ from generic programming languages in that their semantics have an internal axiomatization of space or concurrency, and this makes them particularly suitable for certain tasks that are by necessity concurrent. However, for analytical and practical reasons, it is sometimes useful to reduce their semantics to a formalized notion of a monolithic abstract machine. This abstract machine, defining a class of coordination language-accepting run-time automata, emphasizes the trace semantics of a coordination language as the evolution of a global state space. It allows them to be simulated on a sequential machine and also permits the entire ensemble of program components to be reasoned about as a whole. The collapse of a coordination language into a monolithic machine often leads to a model-theoretic formulation of its semantics, and some languages that take this approach therefore describe themselves instead as 'models', 'calculi' or 'architectures'. Some lack concrete syntax entirely, but for the reasons just described, they will all admit description as 'languages'. We examine them in detail in section 2.6.

2.2 Expression languages as computable functions

In the coordination/expression dichotomy, the purpose of an expression language is to do a one-shot computation: to produce some outputs from some inputs. No state is kept between invocations, so the computation is re-enterable. Turing machines can compute

[2]The accepted taxonomy of languages describes a hierarchy of levels of expressivity, with each level defined by an *accepting automaton* that validates legal members of the language. These are the so-called Chomsky types [49]. Where the present thesis concerns dynamic semantics, it is concerned with infinite streams, and the language of such valid streams are described by an extension to the finite languages which are called ω-languages, as defined by Büchi automata [38].

[3]The fact that they are called ω-automata corresponds with the common usage of 'ω' as a representation of the first infinite cardinal, which is the cardinality of the natural numbers $\mathbb{N}$, or $\aleph_0$.

any computable function [186].

Turing machines define a class of idealized machine, which progresses through a series of discrete states according to some table which, for any given state, defines what actions the machine should carry out to bring it to the next state. It achieves arbitrary complexity through the provision of a notional tape of finite cells of infinite length. These cells contain information that can be read from and written to by the machine. Non-deterministic versions of the machine provide multiple possible next states, and probabalistic machines provide a means of choosing between them, for example using a 'random' tape of entropy. It was shown in the 1930s that anything that can be expressed with this sort of sequential computation can be expressed as a term in the λ-calculus; both are equivalent to the notion of the μ-recursive function.[4] All of these formalisms are equivalent in their ability to express computable functions. By 'equivalent', we mean that any function expressible in one can be encoded in another, and produce consistent results: they are morphisms over relations between any two domains and co-domains. From this, the definitions start to take on the look of an equivalence class, and many others have since produced reformulations and equivalent models that compute equivalent classes of functions. The Church-Turing conjecture concerns how the various definitions of computable functions are to be regarded. They are definitions (or axiomatizations) of an idea, one that seems to be strongly linked to physics, and it is Turing's approach that comes closest to capturing this. More recent work has been done by Gandy [82] and Sieg [171]. This is not the place for a detailed treatment of this subject. The reader is referred to a text such as [52] or [193] for discussions of computability issues.

In the present work, we incline to the physical definition of the functions capturable with an expression language. In support of this, it is relevant to note that the Church models of computation do not have to be — in spite of the fact that they usually are — emulated by a sequential Turing-inspired machine. Direct physical reduction machines have been built which express λ-calculus semantics directly in hardware. These include GRIP [156], ALICE [58], and more recently, Naylor's FPGA-based 'Reduceron' [147].

Languages for Turing machine analogues and the λ-calculus are capable of expressing non-terminating programs: `while(1)` loops in C without `break` or `return` statements are the most obvious example in the sequential case. Primitive recursive functions do not suffer from this problem because the argument on each recursive call becomes smaller in some sense. Similarly, tools implementing a simply typed λ-calculi only define terminating computations: they are *stongly normalizing*.[5] However, there are functions which are effectively computable but cannot be expressed in a primitive recursive

[4]The λ-calculus is due to Church [50]; The μ operator is due to Kleene [119].

[5]The original λ-calculus of Church was typeless, but there are typed versions, which are easier to control for practical software engineering: [20] is the authoritative work.

form: the classic example is the Ackermann function. Turing-complete languages (that is, those that can model a Turing machine) are thus capable of expressing computations which cannot, in general, be guaranteed to halt. This is a problem for a coordination language when used in a safety-critical environment: it is neither deterministic nor acceptable for a one-shot computation, defined in an expression language, to report to its invoking coordination language that it did not malfunction, but nevertheless ran out of time in doing its work. The solution is to place bounds on the size arguments and use ingenuity to find proofs on an expression-by-expression basis that a *particular* expression *will* terminate. Various tools can help in applying patterns of previously distilled insight to these problems, in order to limit the amount of human ingenuity that is required in each case.

The Hume expression language deals with these logical realities very deftly [96].

2.3 Entropy

There is an intuitive link between the idea of entropy as information content and the entropy of physical systems: physical systems with a higher degree of disorder require more information to be used in order to describe them. The information needed to describe a plain piece of card is less than that needed to describe the same card but punched full of holes: holes that may or may not 'mean' something, such as a computer program or the number of teacakes sold in a chain of cafes on a particular day.

The idea of informational entropy originates in Shannon's work on quantifying the information content of communication channels [170]. Shannon entropy is the resulting measure of information. For present purposes, we do not need to delve into the mathematics of entropy, or the intriguingly mathematical connections between the two, but we do need to observe that informational entropy is a measure of the unpredictability of a communication channel. It is entropy as a source of unpredictability that is key to making a probabalistic Turing machine out of a deterministic Turing machine: it is the source of the random tape. If entropy is being received as an infinite stream (or tape), the entropy rate of our idealized source of information will be the same as the bit rate of the information itself. Entropy is not needed to specify a probabalistic computation, but it is needed to reason about the distribution of results given multiple runs: if we wanted to do this, the mathematical definition of entropy would be necessary. The present project is confined to deterministic computations, so we need not do this here.

2.3.1 Finite and infinite computations in the presence of non-determinism

A probabalistic Turing machine can be simulated by a Turing machine that makes decisions on what to do next based on the content of a random tape. A non-deterministic Turing machine is one that has points in its execution where more than one action may occur next. Unlike a probabalistic Turing machine, it does not specify a way of *choosing* what to do next, so in a real semantic sense, the computation can be thought of as defining all of these choices at once. This leads to a rapidly divergent execution path. As each path may give rise to further choices, each of those may lead to yet further choices, and so on, leading to a set of possible execution histories that grows exponentially with the length of the trace. These divergences describe a tree structure, and Plotkin gives a simple proof [157] that if the computation described by the set of possible choices is finite, then the program must be certain to terminate. The proof goes on to show that if there is an infinite set of possible traces, then the computation that produces at least some of them must be infinite, and that therefore some of the possible paths will not terminate, and thus not return a result. This is *bounded* non-determinism.

The idea of a computational system exhibiting *unbounded non-determinism* is that such a system allows computations to be defined that will always return results, but that the set of such results, as determined by the final node on the execution trace, is infinite. If such systems are possible, they do not appear to fit with Plotkin's proof, so what is going on? Clinger dealt with these problems in his Ph.D. thesis [51], and this is where the idea of fairness comes in. If a program is written for a non-deterministic Turing machine, there is no way of specifying how, in an execution, the non-deterministic choice is to be made. If there were two choices — one to halt immediately, and the other to make the choice again — the implementation may always choose to go round again and therefore never terminate. The programmer of the non-deterministic machine has no way of specifying that some fairer policy to choose between the two options should be adopted, such as to toss a coin.

We suggest that a source of Shannon entropy provides this, and so a *probabalistic* Turing machine can *simulate* a system that exhibits unbounded non-determinism. The entropy ensures that some class of programs, which is wider than those that can be certain to terminate on a non-deterministic Turing machine, will nevertheless terminate eventually, but that termination may not happen for an indefinitely long time. Even for this simplest example program, the program is guaranteed to halt (potentially well after the heat death of the universe, in a physical incarnation), but the set of possible results of the computation is infinite.[6] Whether we regard this as meaning that a result will *never* be returned depends on whether we believe that there is such a thing as a completed infinity that bounds our reality. In mainstream mathematics, completed infinities (at least since Cantor) are perfectly normal; constructivists resist the idea of completed infinities.

[6]We can equate the length of the trace with the size of the set of answers if each time the choice is made, a natural number is incremented by unity.

2.4 The controversy of executable specifications

The business of a computation is to take some inputs and produce some outputs. Executable specifications are sets of instructions for how to produce the latter from the former. They concern the *how* of computation, rather than the *what*. The 'what' of computation, mathematically, is a relation over all possible inputs and outputs. While it is trivial to construct such a relation from an executable specification, the controversy of whether this is desirable concerns whether the predicate that is produced in this way is a reliable way of axiomatizing the intentions of the specifier.

While functional programs appear better suited to specifying executable specifications than imperative programs, it is nevertheless still possible to write many different functional programs which compute the same thing, still dealing with some of the *how* of computation: such programs can thus lack canonicity.

The idea that a functional program could be used as a specification was put forward by Turner [188]. There was a flurry of debate in the 1990s as to whether functional specifications *should* be executable or not. A paper by Hayes and Jones [99] argued that executable specifications risk enshrining irrelevant implementation details, and cannot accommodate non-deterministic systems whose relations describe surjective functions. Furthermore, Hayes and Jones argued that specification languages should be able to express non-computable functions so that they can deal with certain theoretical aspects of computation that go beyond what is physically computable. Fuchs [80] put forward counter-arguments to this position, based mainly on the pragmatic idea that useful non-executable specifications can often be made executable. This can be done by introducing constructive elements and transformations that turn them into executable programs. Gravell and Henderson [91] reviewed the arguments, and concluded that executable specifications need not be harmful, but should be used in conjunction with other methods in order to gain confidence that a specification matches intuitive requirements.

In this thesis, we incline towards the position of Gravell and Henderson, and suggest a methodology which extracts the advantages of both executable and non-executable specifications. We advocate a type-theoretical approach, where σ-types parametrized on the type of an input are used to give the type of an output, and the function type specified using these two types is subsequently used to give a *type* of executable specification in the host logic.[7] If that type is inhabited, then there exists an executable specification, but it is canonical if and only if all the other inhabitants of that function type produce the same mapping from inputs to outputs. In other words, an executable specification thus defined is canonical if and only if the relation defining the predicate forming the σ-type is (provably) injective.

It is possible to prototype such predicative definitions of the relation of a function

[7]Assuming that the host logic is based on the λ-calculus and therefore has reduction semantics.

by stubbing out the predicate and admitting any proof obligations while the function is developed. One can then develop the predicate in tandem with filling in the proof obligations it generates, strengthening the executable analogue of the specification. The inspiration for what this predicate should consist of comes from the normal way of specifying what a function should do in a specification language, by expressing logical formulæ that must hold when quantified over all mappings in the function space. If, when this process is thought to be complete, a lemma cannot be proven stating that the predicate is injective, then it *suggests*, but does not (and the undecidability of predicate logic tells us *cannot*) prove that not all of the function space is understood. This prompts a search for new components of the allied predicate. This process offers a way to ensure that in all such cases, the objections of the Hayes and Jones argument are adequately discharged through showing that the executable function is not over-specified, and also that the allied predicate specification is not under-specified. If a terminating case of this process is found, the predicate, or a tautological equivalence class of which it is a member, becomes the canonical specification. If we were to repeat the process, we would know we were specifying the same function *modulo* the relation over its inputs and outputs, because the predicates thus obtained would (if we were right in thinking they did the same thing) be tautologies over their domains, and we should try to prove them so.

Using this methodology in high integrity systems, there is still a human risk of systematically misunderstanding the problem space, that could lead to the wrong functions *and* the wrong predicates. In this case we would have an un-'pleasant' specification, to use Dijkstra's terminology [64]. This might be mitigated by allowing several teams to work on this process independently, who all agree that they want to work on the same intuition of what the specification should be, even if their understandings of what this intuition is differ slightly: every attendee of a meeting writes different notes. If they can then find a proof in the host logic that the predicates on the function relation they have found are tautological, then a high degree of confidence can be obtained that their specifications match the intuition of what was wanted. It is similarly likely that any ways in which each individual's intuition differed were not material to what their team, as a whole, was trying to achieve. Even the first part of this methodology can only be very partially explored in a thesis, but it is nonetheless possible to present a development that is compatible with these principles, and that is what is done in the remainder of this work.

It ought to be stressed that this use of predicates does not mean we are talking about the triples of Hoare's axiomatic basis of computer programming [107] or other predicative theories of programming, such as that described by Hehner [100]), or unified theories of programming [109]. Mostly, these envisage some state space on which a program operates, and look at what can be said of the state space before and after the program has done its work. In the present undertaking, we are discussing logics which are founded in typed λ-calculi, and we therefore view computation more using Church's eyes than Turing's (see section 2.2). Our idea of a function is more the mathematical one of a relation over a domain and co-domain than the imperative one of some kind of sequential procedure on a single domain, and it is those relations that our predicates define.

The rest of this chapter is devoted to examining the dichotomy between coordination languages and expression languages, and their semantics. Proof assistants are amply expressive to describe any kind of specification, executable or not, but the generality of proof assistants and the lack of verified toolchains (a lack which some are addressing, see section 2.10.4) to refine usable implementations have left a niche for another category of specification tool. These toolkits allow the refinement of specifications as axioms of what a system must do, and what its invariants are, into a fully executable specification, in the sort of process described by Fuchs. We introduce them now, before going on to explain why it is that we prefer to specify HBCL (and programs we write in it) directly in a general purpose logic and proof assistant.

2.4.1 Specification language toolkits and verifying compilers

Specification languages have executable subsets, and language toolkits help us to find them through a process of model refinement that might eventually be exported to a directly executable language and machine. A verifying compiler is one that will only compile 'correct' programs according to some specification. It can be a component of a model refinement toolkit, although such toolkits often only emit pseudo-code that must be hand-translated into some other language. One view of a compiler is as a piece of software that transforms, as part of an implementation refinement, a recipe for a computable function into one with more directly physical semantics. Neither verifying compilers, nor specification language toolkits, are necessarily verif*ied*, and this places limits on the confidence that may be placed in the systems they help to develop. We advance the argument that the reason this is tolerated is because it is the most rigorous kind of specification approach that is available, not the most rigorous that is possible (again, see section 2.10.4).

The leading specification and refinement toolkit of the moment is arguably the Event-B [11] specification language and methodology, and its realization in the Rodin tool [12, 164], which provides an implementation.[8] Event-B is an evolved form of the B method [10], which itself owes a good deal to Z notation [4].

2.4.1.1 Specification languages as logical middleware

Specification languages can be viewed as logical middleware. They are not general purpose logic tools, such as proof-checkers or proof assistants, although we might reason

[8]Some work has been done on formalizing B in proof assistants [32].

about them with such tools. They are not programming languages: refined programs for onward compilation may be an output of such systems. These formalisms can be convenient and expressive. However, they can be dispensed with if the axiomatization of a temporal modality is pushed into the coordination semantics of a concrete coordination language, while the logic element of such a language is expressed as a direct deep embedding of that concrete language in a proof assistant's logical system. In this case, the executability of expressions and coordination constructs in the embedded concrete coordination language is guaranteed if the proof assistant can be made to export a suitable interpreter of the axiomatized coordination semantics, whose termination properties are warranted by the host logic. We therefore keep axiomatizations as thin as possible, since the proof assistant is the common denominator and the place where invariants and imperatives of the system are specified. We could axiomatize kinematics in the proof assistant entirely separately from the axiomatization of a concrete language, and specify the relationship between the two at the proof assistant level, thus avoiding bloating either structure and enhancing independence from the logic. That is the approach taken by this work. There needs to be some ontological glue between the two: again, we would like it to be as thin as possible. In the present work, this role is taken by Pre-HBCL (see chapter 3).[9]

2.4.1.2 The drawbacks of specification toolkits

Specification toolkits are captive ecosystems. They contain logic languages for specifying invariants and existential imperatives; they contain imperative structures to specify computational procedures; they produce proof obligations to ensure that computational procedures respect the logical specification, and contain deductive systems and proof-checkers to discharge these obligations. These proof systems lack the power and convenience of full-fledged proof assitants. The objection that none of this is itself verified could be mitigated by developing multiple implementations, or overcome by verifying them. However, to verify them means they must be axiomatized in some other logic, and this itself is a process that may be prone to error, and we of course then approach a circular argument when we ask: how do we verify the verifier? *Quis custodiet ipsos custodes?*[10] There are logical problems with this that run up against Tarski's undefinability theorem [177] and Gödel's incompleteness theorems [88]. We outline some of the ways this has been pragmatically accommodated in existing work on self-verifiability, and suggest some new ways, based on a structuralist approach with networks of physical morphism proofs to limit the effects of these problems, in section 2.9.2. This leads to

[9]If the coordination language makes it possible to construct specifications that are not deadlock/livelock free, then the exported tool must refuse to run such programs. As a corollary of soundness, HBCL is free of this problem: see chapter 4.

[10]Harrison quotes and asks the same oft-repeated question [98] (the phrase has multiple attributions, but this version appears to be from Juvenal's sixth Satire [116], pp. 67–68, ll. 347–348) in a very similar context.

the idea of approaching absolute confidence in a limit. One thing is clear: such confidence can only be derived from very general logical tools that are powerful enough to axiomatize each other,[11] and specification toolkits are too domain-specific to the task of specifying computer systems to be satisfactory for this purpose.

Many safety properties of systems refer to the same imperatives whose importance is a purely human designation. For, example, in any vehicle control system, it is likely to be an invariant of the system that vehicles do not crash. This in turn is a stipulation that implicitly acknowledges the laws of physics, and it is our intuition that if we axiomatize the laws of physics in multiple logics or specification systems, those axioms must be the *same* physical axioms and axiomatized human imperatives,[12] and that differences in structure ought to be accounted for as relations under morphisms for key properties. However, specification languages would grow uncontrollably if they attempted to axiomatize the laws of kinematics, let alone physics generally. This means that it is left to the specifier to render informal semantics of a scenario (such as 'a driver will not drive past red lights') outside the specification language, and nothing guarantees that these individual axioms achieve adequate coverage of the domain to uphold far more general human axioms over physics such as 'high energy collisions must not happen.' [13] Only general purpose logics and proof tools are powerful enough to axiomatize formally *everything* of importance in the application domain, including relevant fragments of physics.

Individual programming languages are mathematical structures that have similar structural universality. It makes little more sense to consider their canonical form as being their over-specified representation in some particular logic than it does to imagine that some arbitrary program is canonical of an ill-articulated set of requirements.[14] If compiler I transforms a program in language A into a program in language B, and compiler II transforms a program in language B into a program in language C, and compilers I and II were developed using different specification and refinement toolkits, how does one know that the axiomatizations of language B in both toolkits axiomatize the semantics of the same language? How does one know that the final compiler stage that produces assembly code respects the same axiomatization of the semantics of the hardware that was used in model-checking that hardware? Large numbers of people working on the same thing will inevitably use diverse toolkits. To assume artificial canonicity in any of these axiomatizations risks an over-specification problem.

A more familiar example is the over-specification of mathematical constructions by

[11]Gödel tells us that they cannot prove each other's consistency.

[12]Similarly, if we were to embed one logic in another, and embed it again in yet another, we would need to show that the axioms of these proof systems were the *same* axioms.

[13]What we mean by 'high' is a parameter.

[14]Unified theories of programming examine the common structure in all programming languages [109].

confusing their popular and sometimes even implicit formalization in ZF[15] set theory and first order logic with the phenomena that are being described. For instance, natural numbers can be formalized in sets using the usual Peano axiomatization [153], but this does not mean that they *are* sets. These are familiar problems in the philosophy of mathematics, which we do not discuss further here, but instead revisit in section 2.9.1.2. We refer the reader seeking a more general treatment of some of these philosphical issues to an introductory text such as [53].

2.5 The coordination language dichotomy

Why use coordination languages? In a loose sense, single-threaded imperative languages can express concurrent processes by using a statement to appeal to a *spawning* operation that creates an independent execution path. Semaphores, mutexes and other constructs arbitrated by an operating system are used to produce *ad hoc* restrictions on the total order of possible interleaved statement processing events. The resulting programs are awkward to reason about, because the important coordination issues become obscured among immaterial sequences of linear operations.

Coordination languages become useful when computations are expressed as functions, which map inputs to outputs without keeping any intermediate state. In this way, if imperative programs are recast in λ-calculus-based functional languages, computation is distilled into *expression languages*, while all coordination issues are dealt with by a *coordination language*, which determines the order in which these expressions execute, and how data is passed between them. This can be regarded as an extension of the pure functional paradigm of programming, such as that found in Haskell [184], where coordination is provided by linking monads with streams. As such, coordination languages are not only useful for programming in a parallel style on a single computer, but also provide a means to specify the interactions of large distributed systems. In the latter case, the objective may be not to compute the answer to a problem, but to specify how the components of a system interact so that it is possible to specify imperatives and invariants.

In the rest of this section, we first examine what physical realities are being axiomatized in coordination languages, before going on to discuss the expressivity of different types in Figure 2.1 and the emulation and animation of their structures by expression languages. We deal with the more difficult converse case in section 2.5.1.4, once we have given a more complete description of the various types of coordination language in the preceding sections. When an expression language models a coordination language, it is simulating coordination; when a coordination language models an expression language, the spatial elements can improve the way in which the corresponding expres-

[15]Zermelo–Fraenkel [78, 194]

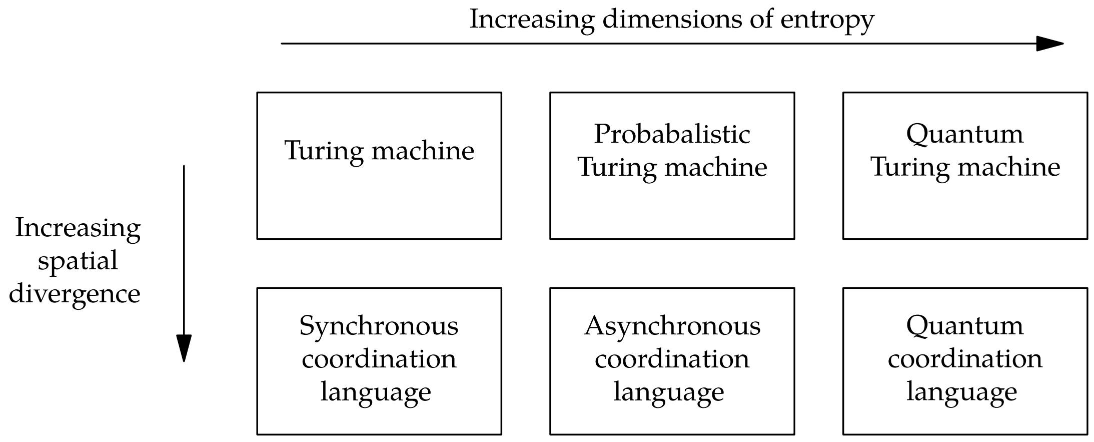

Increasing dimensions of entropy
Increasing spatial divergence
Turing machine
Probabalistic Turing machine
Quantum Turing machine
Synchronous coordination language
Asynchronous coordination language
Quantum coordination language

Figure 2.1: The spatial dichotomy between sequential and coordination languages

sion is calculated. When we look at the second row of Figure 2.1 prescriptively, we are dealing with coordination languages; when looking at it descriptively, we need process calculi, the Actor model, or similar abstractions.

2.5.1 Time and space

Since Babbage, and certainly since Turing, computation has been an activity conceived to occur in space and time, the four-dimensional fabric that is the backdrop to most of our understanding of physics. Non-physical computation can be achieved by 'oracle machines',[16] but oracles are useless if we are interested in finding effective procedures that evaluate answers to problems, since non-Delphic oracles do not exist in the physical world.

Sequential computation is one-dimensional in this four-dimensional fabric. Although in practice it has some spatial localization, that localization is not semantically accessible to the computation. Since there are no mechanisms of interaction, there are no interactions. In section 2.5.1.1.1, we briefly survey what minimal assumptions are necessary about the nature of time for the purposes of formalizing these computations.

Coordination languages and allied calculi often require an axiomatization in three-dimensional space — often a very explicit one — such as in the Ambient Calculus [40] or Milner's bigraphs [141]. If the coordination language is being used as a convenient abstraction to structure an essentially sequential computation, then communication is not spatial, but is just stored in a state variable while something else happens. In this case, an axiomatization in space is not required, and the model collapses to a sequential one, albeit with checkpoints, semaphores, mutexes and similar sub-structures. However, if

[16]The idea of oracles in this sense was briefly introduced by Turing [187] and extensively developed by others.

the coordination language is being used to express the specification of truly parallel hardware, then an axiomatization of space is unavoidable.

2.5.1.1 Axiomatizing time

The nature of time is a philosophical morrass at the limits of theoretical physics: we have no intention of becoming embroiled in this problem. We thus confine our discussion to two pragmatic issues that have to be addressed and accommodated when axiomatizing any temporal formalism: the ordering relation and the texture of time.

2.5.1.1.1 The arrow of time, causality and the ordering of events In an influential paper, Lamport [123] discussed the physical relationship between time, the ordering of events, and the implications this has for distributed systems. Lamport observed the difficulties that physical time poses to theoretical computer scientists: namely, it is impossible to say with certainty what the absolute ordering on some set of concurrent events was, or will be. However, this does not mean that it is impossible to say useful things about concurrency. When specifying a concurrent system that is supposed to exhibit convergent behaviour, it is usually partial orders that are important, and from this Lamport develops his model of event ordering in distributed systems. However, in order to *reason* about properties of a concurrent system, it is necessary to describe its entire state space, and this means that a total ordering must be found. The actual total ordering used is irrelevant, as long as it is consistent with all the partial orderings, and so a concurrent specification essentially under-specifies this total ordering, with the result that proofs have to be quantified over an equivalence class of total orderings.

In his algorithm for finding some inhabitant of this total order, Lamport uses an arbitrary order relation over all processes, over which necessary partial orders are given precedence. Although Lamport does not use the term, the physical provenance of this arbitrary ordering can be thought of as a formalized source of Shannon entropy. Further, Lamport's 'happened before' relation implies a dense or continuous (*i.e.*, not discrete) model of time, since it is always possible to bisect an interval between two events by interpolating a third event that happened after the first, but before the second. This means that, in the general case, it is impossible to know the total ordering on all events in such a system: some events may be so close to each other as to render determination of which happened first an impossible task.[17]

We wish to say little about the arrow of time and causality, save that it is implicitly axiomatized as the causative element in any prescriptive language. Lamport ob-

[17] Unknowable quantities (in some cases reducible to quantum effects) *do not* form the basis for a quantum computer. The point about Schrödinger's cat is not that the cat is possibly dead and possibly alive, but that it is both alive and dead at the same time. This is how the extra dimension in von Neumann entropy arises, as we explain in section 2.5.2.

serves [123] that to say that one event can affect another is one way of characterizing the 'happens before' relation.

Key term: **TAI**

Temps Atomique International. International time defined as that witnessed by an agreed-upon ensemble of atomic clocks, witnessing proper time in the Earth's geoid. The Earth's geoid is a coordinate system that rotates along with the planet; since the reference clocks are stationary on the Earth, they are all in the same relativistic frame.

2.5.1.1.2 The texture of time There are three main ideas characterizing the texture of time: it might be continuous, it might be dense, or it might be discrete. The key point is that we do not need to know which of these it *is*, or whether it *is* any of them. To computer scientists, concerned as they are with discrete mathematical structures, discrete models of time are adequate to deal with sequential computation. They can be logically embedded within dense models of time, which can in turn be described within a continous model of time: any of these are adequate. However, we will argue that adequate models of coordination constructs require at least dense time, since, as we observed about Lamport's reasoning above, whenever the state spaces of two systems are combined, the intervals between one's events may be bisected by virtue of a total ordering relation.[18] If we want to reason about the effects of these computations on the world using the continuous functions of classical physics, then of course we need to embed this in continuous time. In the rest of this section we briefly explain what is meant by 'continuous', 'dense' or 'discrete' time.

Continuous time is the time of classical physics, which uses real numbers and continuous functions to model reality.

Dense time is time understood as a rational number, and is discussed by Kopetz [120]. The conception of time used in the present work is very similar to that of Kopetz, who in passing mentions the use of global clocks and TAI. We take this further in incorporating the ontology of global time more explicitly in HBCL.

Discrete time, or *sparse* time, is time that can be adequately characterized by integers. It is the usual time of computation, in which a state machine advances from state to state in atomic steps.

Lamport's Temporal Logic of Actions (TLA) [125], with its infinitesimally small stut-

[18]Unless we subscribe to a theory that time is granular. Even if we did, times in the order of magnitude of the Planck time are completely useless for anything to do with practical computation, because of the quantum limits of measurement. Further, for the purposes of the kind of systems that we are likely to want to control with computation, classical physics and continuous time are adequate, even if they do not give the most comprehensive account of physical reality: this certainly seems compatible with what was envisaged by Gandy [82] in his abstraction of the minimal requirements for a physical computer.

tering steps, has a continuous notion of time. Dense time formalisms are somewhat rare but do exist, for example in the work of Cau *et al.* [43]. Individual examples in all sequential programming constructs and synchronous coordination languages are reducible to discrete time models: this is the basis of virtually all sequential computation so it would be otiose to cite exemplars. Kopetz [120] makes the point that it is the distribution of processes and the need to compose them in some way that means that a generally descriptive formalism must deal with dense time, while individual processes or sets of processes reduce to sparse, or discrete time. It is worth amplifying this by observing that it is redundant to extend this to continuous time, because in the human act of specifying a perdurant system, there is no practical reason to use irrational numbers when we can approach such numbers with arbitrary accuracy using rational numbers.

2.5.1.2 Axiomatizing space

Not all coordination languages axiomatize space: it is possible to rely on a weaker idea: that of independence of events. While the Ambient Calculus [40] and bigraphs [141], axiomatize space directly, a difficulty arises from what it means to locate a process, which has no dimensions at all in Euclidean space. Those that do, in effect, project computation into space, giving processes volume and attaching semantics to their interactions. This produces yet another non-canonical analogue of a probabalistic Turing machine. Eschewing the idea that the identity and specification of logical processes themselves have any physical materiality, we prefer the abstraction that distribution in space axiomatizes the independence of measurement events and the temporal and spatial uncertainties associated with them: it is those measurements that we wish to use to specify the input of a computable function. This does not result in reduced expressivity, as we can deduce movement and its consequences from the results of measurements relating to positioning systems. The drivers of such movement may or may not be random, but any such randomness is external to the definition of the computation.[19] Therefore, in this work, rather than localize processes, we localize only the *observations* (or inputs) that define those processes and the *manifestations* (or outputs) that signify their successful realization in reality. However, the positions of process *implementations* in space do matter if we want ever to specify and model the interaction of a computational specification with a physical system. Computational processes may be *associated* with pieces of hardware as part of configurations, so that, for example, we can conclude that the physical destruction of a piece of hardware has destroyed the physical realization of the computational process that was mapped to it.

[19]Spatial measurement independence, thus understood, needs to be redefined if the impact of quantum entanglement on quantum computation is considered.

2.5.1.3 Expressing coordination languages as sequential languages

Figure 2.1 shows sequential (or expression) languages in the top row, and coordination languages in the bottom row. We now examine which coordination languages can be executed (if the coordination language does not explicitly axiomatize *physical* space) or simulated (if it does) by which sequential languages. In general, any one can simulate any other, but there are certain subtleties to observe. It takes longer to simulate a program written in the bottom row than it takes to run it in a (physically) spatially distributed form, and it takes (exponentially) longer to simulate a language on the right of the diagram with one on the left. The latter is just another way to phrase a point originally observed by Feynman [74]. Following Deutsch [63], we express this as a quantum Turing machine in the diagram.

Synchronous coordination languages can be simulated with sequential Turing machines. In general, there are many expressions to be evaluated at a given time, and a coordination language in its spatial incarnation supposes that these are evaluated in different places. If this language is simulated by a single sequential machine, then each of these expressions is evaluated at the same logical time (by a wall-clock in the physical coordination analogue) and it cannot matter to the state space trace in which order they are actually calculated, since information travels at the (finite) speed of light; there can therefore be no causation between these co-incident expressions, and thus, by contradiction, it does not matter in which order they are evaluated. The specification expressed by a synchronous coordination language is therefore completely deterministic. Moreover, as a direct result of this proof sketch by the joint axiom (of physics) of the finite speed of light, and the axiom (of synchronous computation) of simultaneous state evolution, any two synchronous coordination programs are orthogonal and composable.[20]

Asynchronous coordination languages on the other hand, allow events to occur according to their own clocks, where the unknowability of the total order, as reasoned about by Lamport, means that to produce a trace in this language means that an arbitrary ordering of concurrent events must be found. If it is proposed to model the whole state space of these choices, then a sequential Turing-equivalent machine can be used. This is unlikely to be practical, since the number of such traces rises exponentially with the number of processes. If we further specify that we want to *pick* one of these outcomes in a *fair* simulation, so that a run of the simulation is indistinguishable in effect from a run of the physical parallel machine, then, in the absense of a source of real entropy, a deterministic machine is inadequate. To effect a simulation then, it is necessary to add a source of entropy, in order to pick an execution path at each divergent point

[20]It would be dangerous to generalize a synchronous system of quantum computers because the possibilty of quantum entanglements has exotic effects on this causation argument: although quantum entanglement cannot be used to move information faster than the speed of light, it *can* apparently produce instantaneous causation.

when there is a choice of relative order to be made. The choice of order may well affect the result of the computation. This is equivalent to the 'random' tape in a probabalistic Turing machine .

The same arguments about partial and total orders affect a quantum coordination language, which needs the same level of randomness to simulate the coordination aspects of such a construction. At the same time, quantum computers are required to perform the quantum computations specified by individual sequential processes, and if communication is generalized to the point of allowing communication of quantum information, the causation consequences of superposition have to be considered. Given that quantum computers in their present form are still highly experimental, there is not yet any practical justification for considering this in any greater detail.

In the synchronous and asynchronous cases, simulating coordination constructions has a practical use beyond modelling, in that it is a technique to simplify the resource bounds analysis of a computational problem. It may be easier to solve a sequential problem by transforming it into a coordination one using a *box calculus* [92, 93], and to simulate the resulting coordination program on a sequential machine, than it is to analyse the resource use behaviour of the original sequential program. We discuss this further in section 2.5.1.4.

2.5.1.4 The emulation of sequential languages by coordination languages

In section 2.5.1.3 we discussed how coordination languages can be emulated by sequential machines performing serially the computations that go on concurrently in a natively parallel system, but doing them in an order that preserves the same causal relationships as in the native case. We shall see in section 2.6.2 that the axiomatization of asynchronous coordination languages causes some controversy about the nature of computations that can be specified, but that these problems can be accommodated by axiomatizing a source of entropy and using *probabalistic* as opposed to *non-deterministic* Turing machines.

Reasoning about coordinated computation is always more difficult than dealing with purely sequential machines, so there has to be a pragmatic reason to make it worthwhile. Two questions need to be asked. First, can coordination languages achieve speedups compared to ordinary computation in the same entropy class illustrated in Figure 2.1? Second, can a coordination language $\mathscr{L}_C$with expression language $\mathscr{L}_E$ simulate a sequential language above and to the right of it in the diagram more efficiently than $\mathscr{L}_E$? We address each point in turn.

2.5.1.4.1 Refactoring expressions as coordination constructs for efficiency

Now that processors have effectively reached physical limits on achievable clock speeds, the computing power on a chip is being increased by adding extra cores — separate sequential processors on a chip, or on a motherboard. This has given rise to the popular notion that new ways have to be found to write programs to deal with the coordination issues this brings up. It also poses the problem of how to speed up problems which have traditionally only been thought of as being expressible as linear and cumulative computations. However, we do not wish to examine this issue here. We are more interested in explaining a less obvious but equally useful rôle for coordination languages, that of refactoring a program to help with termination analyses.

The Hume box calculus [93] is a formal tool for refactoring Hume programs where the expressive power of pure functional constructs is traded for the resource-analysis tractability of more concurrent implementations. These transformations include the introduction of nested boxes via hierarchical Hume. It provides transformations for reasoning about the correctness of such refactorings.

This use of coordination languages does not mean that the coordination-equivalent program should necessarily be regarded as the more canonical one. If the coordination incarnation of the specification obfuscates the intention of the original specificier, something analogous to Ockham's razor ought to lead us to reject it as such. In particular, if we wish to use an entropy to perform a probabalistic computation, it can be argued that it is better to specify a sequential machine with entropy, than to include it obscurely in a coordination model.

2.5.1.4.2 Additional computational power of coordination languages? The 'NC' complexity class [54] describes the complexity of a class of problems that can be handled by parallel computers with resources only polynomially scaled in comparison with the size of the problem, so it can be viewed as a parallel version of the P complexity class. Its relationship to classic sequential complexity classes is unknown, so it would be rash to claim that coordination analogues of sequential programs can always be made to run faster than the original program. It seems difficult to imagine how programs based on cumulative information like one-way hashes could be scaled to run in parallel, without changing the original algorithm.

There is one open-ended question we would like to end with. Looking at Figure 2.1, can feedback and causation in a non-deterministic coordination language amplify the dimensions of the entropy?

Process calculi are usually used to describe systems that must exhibit some convergent behaviour notwithstanding the indeterminacy involved in the ordering of events, the non-deterministic selection of possible behaviours, and non-deterministic choice operators. There is nothing that prevents them from being used to implement some sequential probabalistic algorithms in the same way that a synchronous coordination language can be used to rephrase some sequential deterministic programs.

Although quantum computers are not yet a practical reality, it is also worth noting, however, that it has been shown that classical network and graph structures can be characterized as containing von Neumann as well as Shannon entropy [16,97]. By extension of the argument for using asynchronous process calculi to model probabalistic computation, this raises the intriguing possibility that it may be possible to simulate quantum computation with classical asynchronous models more efficiently than with an ordinary Turing-equivalent computer (although less efficiently than with a native quantum computer). Such a model would require the Hilbert spaces of quantum computation to be encoded in the causalities of evolutions of non-deterministic coordination structures, whose state spaces have exponential sizes with respect to the size of the system. Given the formidable difficulties of building real quantum computers, this is a potentially interesting avenue.

2.5.2 Axiomatizing entropy

A final point is that the amount of entropy in a communicating asynchronous process is variable depending on whether the processes interact on the basis of some sort of 'perfect' internal entropy, such as radioactive decay, or whether they are part of a plesiochronous system in which the spatial divergence, thermodynamics of and physical limitations on the measurement of clocks have an effect. Such effects produce uncertainty in establishing a total order of events in a system which we would otherwise prefer to specify as perfectly synchronous. In this case, there is some entropy in the total order, but any engineered tolerances of clock skew limit the amount of entropy so that the same partial order is always respected.

Von Neumann entropy [191] generalizes Shannon entropy, measuring the information content in the complex plane of a two-dimensional Hilbert space. It is of primary interest in the way the informational analogue of the physical entropy of the quantum system can be physically represented in the superposition of two quantum states. An array of quantum bits exponentiates the amount of information that can be carried by an equivalent array of simple probabalistic bits of the same size. Von Neumann entropy can be found within non-quantum coordination structures [16,97]. No formalization of von Neumann entropy is required in the present work.

2.6 Concrete coordination languages and structures

In common parlance, a computer is a box containing a central processing unit (CPU), and until recently that CPU resembled only a single Turing-equivalent machine. In this section, the term 'computer' will be used somewhat interchangeably with a single Turing machine, functional reduction machine, or other Turing-equivalent automaton.

The semantics of Church-Turing computation directly or indirectly require machines, which are always implicitly instantiated and localized in different parts of space, giving an intrinsically physical animation to the notion of concurrency. My computer sits in a grey box next to my desk. Other computers are in other places: on another desk, or embedded, for example, in some application-specific device. Multiple processors on one chip are conceptually no different from two processors in two different interconnected computers in two different grey boxes. Coordination languages attempt to express the concurrency of a number of such automata in order to write programs that compute something more quickly using several computers, or are easier to analyze than equivalent programs written for a single automaton. Axiomatizations of concurrency assume independence of events in different processors: it is usually safe to ignore the electromagnetic or other physical interaction of two properly functioning computers sitting side-by-side, unless they are connected together with a communication mechanism. This assumption is implicitly taken to be an accurate model in most circumstances.

2.6.1 Synchronous coordination languages

The primary characteristic of synchronous languages and architectures, of which we give examples later in this section, is that their evolution is driven by time, and not by communication between system components. This makes them particularly suitable for use in real time applications. They are extensively used in safety-critical systems because they permit a tractable approach to replicating components that does not force application programmers to implement bespoke asynchronous consensus solutions. One of the earliest papers advocating this approach was Lamport's *Using Time Instead of Timeout for Fault-Tolerant Distributed Systems* [124]; the attractiveness of this solution is underlined by the discovery soon after by Fischer, Lynch and Paterson, that distributed asynchronous consensus with even one faulty process is, in fact, impossible [75].

While these languages have a deterministic evolution that can be simulated by a single thread of execution, they can nevertheless be considered to be coordination languages under our taxonomical scheme because they internally axiomatize concurrency or spatially dispersed evaluation; in a practical sense, they just stipulate that every total order of the execution of a distributed implementation preserves the partial orders described by the synchronous language.

Synchronous languages axiomatize that *there is* a clock by which state evolutions are timed, but say nothing about what that clock is. It is required to run forwards only, but may do so at a rate and with a lack of stability that bears no resemblance to wall-clock time. Usually, the implicit assumption is that a program described with the synchronous language will be instantiated in some sense in many different pieces of equipment with many different clocks: once the program is loaded, it is capable of saying how the hardware should behave, with reference to its own clock. We refer to this type of synchronous language by the term *internally timed*, to emphasize the point that the language is both dependent on the instantiation of a clock, and also that if we look at a black box realizing the program from the outside, we have no direct way of observing that clock.

To suggest another novel term, an *externally timed* language axiomatizes a *particular* clock outside the program, and the programmer or person initiating an execution has no say in choosing that clock: it is part of the language and, axiomatically, cannot be instantiated after some particular program has been written. For universally composable systems on planet Earth, the sensible choice must be humanity's canonical time standard: TAI. This is not to say that the language itself cannot be instantiated according to isolated clocks, but if it is, it becomes a qualitatively different language whose clock is not objectively defined in those parts of reality where there is no knowledge of it. The key difference with an ordinary synchronous language is that implementations of programs cannot smuggle in their own definitions of clocks: clocks are defined in a one-to-one relationship with a *language*, and any quantification over multiple clocks must therefore be done at a *meta-language* level. This is a consequence of having a language defined over an explicit ontology. We suggest that this sort of competitive instantiation is of little practical interest: what *is* of interest is that that by axiomatizing the ontology of time — proper time in the Earth's geoid[21] as witnessed by the weighted average of the clocks in the TAI clock ensemble — it becomes possible to judge any system in the language by the same consistent temporal standard of observation, in the same way that by fixing the logical semantics of a language, we start to gain confidence that all programs written in the language are semantically consistent.[22] The idea of an external clock *rate* forming part of a specification is present in a synchronous communications standard such as SONET/SDH.[23] The necessity for knowledge of the *absolute time* arises in applications such as global positioning systems, or in some modulation schemes used in digital broadcasting or wireless data networks, but it is novel that this necessity be built into a language. We develop the idea of an externally timed language in chapter 3,

[21]The restriction to the coordinate system tied to the lump of rock on which we live makes the definition relativity-proof and accords with the specification of TAI.

[22]To deal with an objective standard of observation requires a spatial axiomatization and an axiomatization of human designation. This is discussed in chapter 3. It also requires that all people and tools using a language reference equivalent axiomatizations of it. This is an issue of epistemology that, when taken to the extreme of questioning the formal system in which these axioms are stated, also runs up against the realities of Gödel's incompleteness theorems and the limitations of logic. We acknowledge and discuss these issues later in this chapter.

[23]'Synchronous Optical Networking' [179] and 'Synchronous Digital Hierarchy' [8]. Both standards do much the same thing; SONET is defined in various documents by Telcordia Technologies, and is widely used in North America, while SDH is a set of ITU standards, and is widely used elsewhere.

but for now we survey existing synchronous languages, which we regard as internally timed.

2.6.1.1 Field bus-oriented architectures

A number of systems in recent decades have been developed to address coordination amongst *nodes*, focusing on transmitting observable quantities over buses in a temporally coherent way. The principal shortcoming of the *node* abstraction is that it does not in itself offer any way of reasoning about the mutual coherence of replicated nodes.

The Time-triggered Architecture (TTA) [121] is a comprehensive high level model for describing systems. Its abstraction consists of a unified time base, observations of state information and events, computation nodes and buses. Coordination is achieved by limiting clock skews between components to be less than a silent interval, during which nothing is scheduled to happen. It defines a number of interfaces for control and data flow, and some explicit mechanisms for dealing with fault tolerance. All of this represents a useful high level abstraction over hardware, but presents difficulties in a workflow involving formal refinement from a functional specification, because there is no good way to define the behaviour of each node. Each node can contain latent state information, and this limits the transparency of the fault tolerance facilities, since the application programmer must still deal explicitly with the possible consequences of divergent state under pathological execution conditions.

Rushby provides a useful survey of other bus architectures targeting fault-tolerant applications [167]. All of these systems express relatively low-level hardware-features, and other than TTA, do not have high level timing abstractions that are relevant to the present investigation.

2.6.1.2 Other schedule-refining architectures

The Hierarchical Timing Language (HTL) [101] is a framework for formalizing the timing and scheduling properties of a concurrent control system. It is a generalization of the Giotto language [103]. Its semantics are based on a 'Logical Execution Time' ('LET') model. The LET model defines a notional time before which a piece of code ('task') may not start executing (the 'release time'), and another notional time by which it must have finished executing (the 'termination time'). There are 'communicators' which move data around, and various higher level constructs for convenience and modularity. HTL uses an 'Embedded Machine' or 'E-machine' to police these requirements, ensuring that these logical time constraints hold within wall-clock time. The model produces 'E code' which specifies the timing and schedule properties to the E-machine. The task of guaranteeing resource use is delegated to the task code.

The HTL progresses some way towards architecture-independent design, but it has a number of drawbacks:

- The LET semantics can lead to some arbitrary timing choices, based on a best guess of execution times (which is architecture and implementation dependent).
- The E machine does not schedule code: rather, it waits for timing or other hardware interrupts and determines whether a piece of code may start to run, or whether it should have terminated. The actual scheduling is done either by the operating system (if there is one) or by a dedicated scheduler. Henzinger et al. have produced a system that uses 'S code' to define a schedule [102].
- While the E code is portable, the tasks that it supervises are not, since resource usage may vary by compilation target. The HTL does not make worst case execution time (WCET) or other resource usage guarantees.
- A task may include arbitrary code in any language. This code may keep state in between the times it is enabled, which introduces a potential state divergence problem if we consider replication.
- E code introduces another specification language into the verification process that is not normative: this reduces its usefulness as a universal interface between real time program specification and implementation.

Chatterjee et al. [45] have developed a reliability analysis for fault-tolerant implementations of HTL models, but this analysis assumes that processors either function correctly or not at all, which is a significant drawback.

AutoFocus is an architecture of 'time-synchronous streams' and 'stream processing functions' [183]. These stream processing functions, or 'components', are not necessarily deterministic. Each component is connected to streams through named ports, and these structures can be nested to form composite components. The timing model is globally clocked,[24] but composite components have internal freedom to manage the evolution of their own clocks, as long as their external interfaces remain synchronized with the global clock of the enclosing component (or, at top level, global scope).

2.6.1.3 Fully descriptive languages

2.6.1.3.1 RTL Hardware description languages Hardware description languages at the Register Transfer Level ('RTL') are clocked, and can therefore be regarded as synchronous languages, although at a very low level. Classic examples are Verilog [2], VHDL [1] and more recently the RTL subset of SystemC [3] version 1.0. Like ordinary

[24]'Global' here means that each specification uses the same clock reference: it says nothing about the relationship of these clocks to each other or standardized time.

programming languages, hardware description languages can be executed (although the word 'simulated' is more often used, because the target is a physical machine rather than an abstract one). However, the dynamic semantics of hardware description languages are often less formal than those of programming languages, again, because they are intended to be physically intuitive descriptions. The semantics are those of physical electronics and very close to the semantics of physics itself. The semantics of physics are laws of Nature; from the point of view of logical systems, they are axioms.

2.6.1.3.2 Esterel Esterel [35] is a synchronous programming language that has been in development for three decades. It operates at a higher level than hardware description languages and has definite, precise semantics in the language-theoretic sense. It has a global notion of time, that functions as a single axis on which all computation events have a definite position. Events scheduled to occur at the same time execute infinitely fast and simultaneously. Communication occurs through 'signals', which are broadcast across a system. They function very like wires, and like wires, are free of the blocking behaviour that can occur in a message passing model. However, unlike physical wires that have an enduring existence, a signal can legitimately be entirely absent in a cycle. This deterministic axiomatization of concurrency is deliberately unphysical, but is ideal for specifying systems that should exhibit deterministic behaviour: specifications can be implemented by more realistic models of concurrency and then shown to be witnesses to the original Esterel specification. In order to avoid infinite loops on signals being written and read in the same instant of time, there are semantic rules to ensure that signals remain coherent. Esterel modules can contain some basic imperative programming constructions.

2.6.1.3.3 Hume coordination language (unfair subset) Hume [96] is a language in the pure functional paradigm, in which concurrency is axiomatized in the simultaneous existence of multiple *boxes*. Hume's coordination language concerns the interaction between these boxes, and its expression language concerns the computation that occurs within these boxes.[25] Hume does not have an explicit external time axis in the core semantics: this is added when assumptions about the clocking of inputs are made when the program is analyzed for execution time constraints. These constraints are introduced first *via* an intermediate language [114, 115], and are then refined to implement the correct resource behaviour on a particular hardware platform.

Hume's scheduling comes from its 'super-step' [94]. The super-step ensures that data buffers between boxes ('wires') remain coherent, by requiring all executable boxes within a super-step to be run only once, before boxes are next assessed for runnability

[25]Boxes can be nested in some versions of Hume, in which coordination constructs are allowed within boxes [92].

in the following super-step. The evolution of the super-step can be thought of as the progress of an internal global clock. If all boxes in a Hume development have unfair pattern matching (in other words, matches are evaluated in a fixed order), any overlap in patterns is resolved the same way each time a box is run, so the evolution of the whole system is completely deterministic. Given that the semantics of Hume ensure that the order in which executable boxes are run is immaterial, they can be regarded as evolving simultaneously, thus not producing an underspecified state space in which different interleavings of events would be possible. In this respect super-stepping semantics share some similarities with Esterel.

The Hume expression language has a number of levels, ranging from a hardware description language (HW Hume) to a Turing complete programming language (Full Hume). Each level has more expressive power than the level below it, but it becomes progressively more difficult to reason automatically about resource use: in the case of Full Hume, it is impossible, on account of the halting problem. There are three intermediate levels of Hume. Finite state machine Hume is similar to HW Hume, but can replace binary data types in HW Hume with more abstract (non-recursive) types. It also allows non-recursive first order functions and conditionals. Template Hume adds a set of common predefined higher-order functions, such as map and fold. PR Hume allows primitive recursion and inductive data types. Full Hume has no limits on recursion.

One of the Hume design paradigms involves refactoring a program written in a high Hume level into a lower Hume level so that the resource use analysis tools can generate upper bounds.

The centrepiece of Hume's resource use analysis is the automatic amortized cost analysis tools described by Jost *et al.* in [114]. The idea of amortized analysis was first proposed by Tarjan [176]. The idea is that the state of a computation is associated with a potential, or amount of some time or space resource available. Each step of execution then increases or decreases this potential. This type of analysis means that it is not necessary to reason about absolute potential at any point, but only the changes of potential. Tarjan's technique required ingenuity to determine suitable abstractions to make the state spaces of complex code structures tractable. A simple recursive function that, for example, summed integers with a suitable base case, would be easy to reason about. However, to determine an abstraction manually for a complicated function that uses deeply structured data and conditional constructs is very difficult. The key contribution of Jost *et al.* was to use automatically potential-annotated types to provide an automated analysis, invoking linear programming solvers to determine an abstraction that could efficiently partition the trace space. Hume source code is first translated into the lower level 'Schopenhauer' language. The upper bounds found in this way are not minimal, but they produce good maxima that are not impractically larger than empirically derived resource use data.

The automated analysis uses the resource behaviour implicit in the semantics of the Schopenhauer language. To translate this into the semantics of real machines, it is necessary to carry out a lower level analysis which takes into account the resource usage behaviour of particular platforms. This has been done for the Renesas M32C/85 board [33], using a proprietary tool. The analysis must be repeated for each targeted hardware architecture and compiler tool-chain used in order to generate an executable from Hume code. Only precision-timed architectures (*i.e.* those that do not possess effectively non-deterministic optimizations) can provide the necessary hard upper bounds on resource usage by particular instruction sequences.

2.6.2 Asynchronous languages, models and calculi

Asynchronous coordination models are characterized by assuming that all events take place at distinct times. As a result, a total order exists among events, even though that total order may not be known: this is usually the basis of an asynchronous language's implicit axiomatization of concurrency. Reasoning about such systems involves proving properties of interesting partial orders, and also showing that desired invariants are unaffected by permutations of the total order. As a total order is unpredictable and even unknowable, asynchronous models axiomatize a source of Shannon entropy. Randomness is essential to prove liveness in certain circumstances, a property that in the context of non-deterministic choice is often described as 'fairness.' The issue of fairness is thoroughly bound up with the thorny issue of unbounded non-determinism, which appears to threaten our claim that asynchronous coordination languages are, in general, reducible to what can be easily calculated with non-deterministic Turing machines.

2.6.2.1 Abstract and descriptive languages

2.6.2.1.1 Petri nets and derivatives as coordination languages Petri nets, as documented in Petri's Ph.D. thesis of 1962 [155] are one of the oldest formalisms for concurrency, and have been extensively developed and extended by Petri and others. At their most basic, they are bipartite graphs of 'places' and 'transitions' whose state is determined by the allocation of tokens to places and whose temporal evolution is determined by the semantics of enabled transitions that 'fire', consuming tokens from one place and depositing them in others. Petri nets have been enriched with data types and functions (coloured Petri nets [113]); pure functional languages such as Haskell are especially well suited for this [161]. Stochastic Petri nets [23, 132] formalize the firing delays with probability distributions. Hybrid Petri nets [59] allow continuous functions to be introduced into a system description, which is useful in modelling the control of physical processes. Formal hybrid approaches to system design have been advanced in the BASYSNET methodology [169].

Petri nets continue to be used in formalizing more traditional coordination problems in concurrency and scheduling, as in recent work by Calvert and Mycroft [39].

2.6.2.1.2 The Actor model as a coordination language

The Actor model of Hewitt *et al.* [106] was introduced in 1973 and axiomatizes everything as being an actor: an actor can, in the words of Hewitt *et al.*, be '…an active agent which plays a role on cue according to a script.' Actors may produce new actors. The sending of messages is more ambiguous than with process calculi, since there is no synchronous coordination on the sending of events (*cf.* Section 2.6.2.1.3). The Actor model emphasizes the axiomatic independence of actors, with their own local times. The sending and receiving of messages is not synchronized, but eventual delivery is guaranteed: this is the incarnation of unbounded non-determinacy. This lack of global time and uncertain status of messages in transit has caused problems in reasoning about the Actor model: these problems were addressed by the introduction of ordering laws [105] and apparently resolved by the introduction of global time and an extended powerdomain[26] model by Clinger [51]. Something akin to a global state can be found in the powerdomain, but not in the Actor model itself. There appears to have been some confusion since as to the relationship between the logical or physical existence of a global *ordering* (or setoid of global orderings), and the existence of a global *clock*. In Agha's authoritative exegesis of the Actor model [14], he again reiterates the tenet that a global clock is meaningless, but also cites the work of Clinger as being consistent with this. The implicit conflation of the logical existence of an axiomatization of a set(oid) of global orderings and the lack of knowledge of a system that makes a global clock implausible, seems to be at the root of much disagreement about the Actor model.[27]

Hewitt insists that the Actor model is inspired by physics [104], and this is a source of some problems. Without an explicit axiomatization of both the Actor model and physics in a logical system, an inchoate appeal to physics seems unconvincing as an argument for the reasonableness of the Actor model. The Actor model is not in itself a physical theory, and it is only axioms that represent physical laws of Nature that are beyond human disputation in scientific discourse, in the absence of contradictory evidence. The achievements of physics flow from its axiomatization in mathematics (itself usually axiomatized in ZF set theory) and informal inferences in the progressions between lines of proof; computational models that ignore this need for axiomatization are forced to rely

[26]Powerdomains are originally due to Plotkin [157].

[27]The sheer unknowability of total orders occurs frequently in reasoning about distributed systems, but does not imply that axiomatizing those systems with respect to total orders causes inconsistent reasoning. Lamport, in his Paxos paper [126] on the (fictionally) eponymous asynchronous consensus protocol, refers to a global state that is a 'quantity observed only by the gods': this is a metaphor that assists understanding without, of course, saying anything about theology, although it does make the usual implicit assumptions about the ontology of concurrency.

on informal axiomatizations, which are a recipe for misunderstandings, inconsistency and muddle. If we were forced to axiomatize the relevant part of a physical system *together* with the logic specifying or describing how it evolves, we might first very well instantiate the four-axis system of spacetime, and immediately there ceases to be anything mysterious about concurrency: everything to do with fixing orders falls out in a neat correspondence between entropy in Boltzmann's statistical thermodynamics and Shannon's theory of information. Hewitt has put forward [104] relational quantum mechanics [166] as a justification for not considering the idea of a global state at any one time to be a real phenomenon, extrapolating that there is no reality other than interaction. However, relational quantum mechanics is only one interpretation of quantum mechanics. The consistency of an interpretation of part of physics is not grounds for rejecting bisimilar ones, unless one wishes to invoke metaphysical dogma, and should not lead us to reject a formalization based on partial orders and the elegant and intuitive way it allows us to quantify over time.

Setting aside the physics-based controversies that tend to proliferate around it, much progress has been made in reasoning about the Actor model by axiomatizing it algebraically. Gaspari and Zavattaro introduced an algebra for reasoning about the Actor model in 1997 [83]. Agha and Thati again took an algebraic approach to the Actor model in 2004 [13], using a typed version of the π-calculus, which they call Aπ, to reason about a simple object-based language, observing that the expressivity and reconfigurability from within the semantics of the Actor model has interacted with the drive to evolve process calculi into mobile and mutable forms. This trend towards algebraic treatments of Actor-style models continues to make progress.

The Actor model is a very useful abstraction of widely distributed interacting objects that fits in well with the popular object-oriented programming paradigm. However, some of the dogma associated with what it *means* and how it can or cannot be reasoned about using algebraic techniques should be treated with caution. In the course of discussing their algebraic approach, Agha and Thati [13] draw attention to Milner's inspiration from the Actor model in designing the π-calculus [140], and it is to process calculi that the discussion now turns.

2.6.2.1.3 Process calculi and algebrae as coordination languages Process calculi provide means of formalizing and expressing concurrency, axiomatizing the independence of sequential processes that in a concrete model might be separated by time or space. The first process calculi were the Communicating Sequential Processes (CSP) of Hoare *et al.* [37, 108] and the Calculus of Communicating Systems (CCS) of Milner *et al.* [139]. The fundamental abstraction of both is the process, which is an algebraic entity. It is defined by the set of actions that it takes (CCS) or events that it accepts (CSP) in order to evolve into other processes. Events may have data attached to them. Processes can exist concurrently and their behaviour is independent, except when they communicate by synchronizing on an event or action. Where a process is able to evolve through a choice of behaviours, non-determinism is introduced. This behaviour was borrowed from Dijkstra's idea of guarded commands [65]. A process has no character apart from the behaviours it may exhibit in transforming itself into a different process concomitantly with defined actions or events. Process calculi can accurately describe localized systems where those systems can be characterized as having a number of states, and in which those states can be mapped to a set of processes whose traces form closed loops. Synchronization on events amounts to a synchronous form of message passing. This is a good description of the observable external behaviour of a piece of sequential (or nested parallel) code running on either a single or multiple processors.

Process algebræ such as the Algebra of Communicating Processes (ACP) [29] study similar structures to those specified by their 'calculus' cousins, but stress structural algebraic properties over what they are physically meant to represent.

Timed process calculi Both CSP and CCS have timed variants, timed CSP [60, 160, 168] and timed CCS [180, 181] respectively. The timed versions are able to model such things as delays and timeouts, restricting when certain processes may run with respect to a clock. However, they remain fundamentally asynchronous languages, being extensions of the core semantics of their untimed counterparts.

Reconfigurable and mobile process calculi The first process calculi were static structures. Unlike the Actor model, processes could not send or create other processes or set up new communication channels. The π-calculus [142] [143] allows links to be reconfigured between processes. The ambient calculus [40] uses a quite different set of primitives: processes contained in 'ambients', which are spatially defined localities in which computation happens. This enables processes to be moved around in a semantics of containment, closely modelling how a mobile computing device moves through administrative domains, or a remote procedure call is made on some distant machine. Yet another model with mobile agents can be found in Milner's bigraphs [141].

Quantum process calculus Communicating Quantum Processes (CQP) [84] is an extension of the π-calculus to incorporate quantum data types and quantum communication alongside classical coordination constructs. In the absence of practical quantum computers, it is currently only of theoretical interest.

2.6.2.2 Complete, concrete and prescriptive languages

2.6.2.2.1 Occam Occam [6] is a parallel programming language originally developed and introduced by May [135] for programming the INMOS 'transputer', a novel parallel processing architecture put forward in the 1980s. It has ordinary imperative features as well as constructs for specifying that code must be run sequentially or in parallel. It is directly inspired by CSP, and can be regarded as a concrete language analogue of the process calculus.[28]

2.6.2.2.2 Erlang Erlang [17] is a functional programming language that is designed for distributed execution. The main abstraction used by Erlang is the 'process' [44]. Processes are essentially repeatably executable instances of functions whose domain is the messages that can be received by other processes, and whose range is the messages that it can send to other processes. Erlang's communication and concurrency model is in essence that of the Actor model, and has in turn been modelled by D'Osualdo *et al.* [66, 67] using 'Actor Communicating Systems', from which they generate Petri nets and use a coverability checker to automate the verification of the original Erlang programs. One could translate this into a process calculus, if desired, using the execution semantics of Petri nets which are defined in terms of a relation over the state space that resembles the trace semantics of a process calculus.[29] Owing to the ease of distributing and replicating code, and migrating functions from failed nodes to active ones, Erlang can be used to make reliable implementation systems with soft real time constraints. However, it lacks generic solutions to problems of consensus and network partition, which causes problems when the persistent storage of data is required.

2.6.2.2.3 Hume coordination language Hume has been discussed in section 2.6.1.3.3 with reference to the unfair mode of pattern matching in boxes. Where fair matching is specified, Hume's semantics resemble asynchronous coordination languages, since its matching semantics then look like guarded commands.

2.6.2.2.4 Linda Linda [42] is a coordination language that is characterized by concurrent access to a global *tuplespace*. Worker processes, rather than send messages to others or push them down channels, communicate only by the intermediation of the tuplespace, into which messages are written and read. Interaction with the tuplespace is atomic. This is an elegant abstraction which is independent of how the computable functions of processes are specified, and also completely decouples expression from coordination. The idea of tuplespace put forward in Linda has been implemented in other similar languages.

[28]Although the 'transputer' is no longer developed, XMOS, also devoloped by May [136], can be regarded as a successor technology, and can be programmed using the XC language, which is essentially C but with extensions for parallelism.

[29]The opposite transformation has been much explored (most recently, in [130]).

2.6.2.2.5 Functional reactive programming languages Functional reactive programming (FRP) [70] is a programming paradigm that closely follows the monadic approach to the interaction of pure functional programs with their environments. The essential idea is that the evolution of the program's dynamic state is triggered by external events which propagate through the program as a set of causes and effects, whose computational content is abstracted into pure functions. The process is asynchronous.

2.6.2.2.6 TLM Hardware description languages Transaction level modelling is an abstraction of hardware description languages that starts to abstract away from timed implementation details. The structures it defines begin to look like those defined by concurrent programming languages. The current SystemC standard [3] includes detailed facilities for this kind of asynchronous modelling.

2.7 Evaluating agents and entropy in concrete coordination models

Agents on the one hand, and the entropy of autonomous or parallel processes on the other, can in many ways be regarded as two sides of the same coin. However, we are developing the idea that in a crucial sense they are quite different, a difference that is frequently elided by process calculi, the Actor model and many popular models of concurrency.

Save insofar as all science is corrigible by new experimental observations, the laws of physics themselves are a complete and fully adequate model of the abstract idea of concurrency; indeed, it is more accurate to say that the idea of concurrency is an abstraction of part of physics. However, Hewitt's idea that the Actor model is 'inspired' by physics [104] is odd: a model inspired by physics, but short of a formal axiomatization of physics, claiming itself to be an adequate model of an interacting physical system, looks rather like an obsfucation of physics. The word 'actor' is a clue to the confusion. It implicitly mixes the idea of human free will and agency with the entropy and uncertainty inherent in quantum and chaotic phenomena. The only way to assert consistently that human agency and physically concurrent independent events are the same thing is simultaneously to espouse some quite specific theories of human consciousness, and that strays far from the subject matter of computer science. We encounter the conflation of agency and entropy again in the summary of HBCL design in appendix A.4.1, and explain further there how HBCL does things differently.

Process calculi and algebræ are more neutral models, put forward with less vociferous and polemic ardour than the Actor model. They are perfectly good mathematical models of independent and interacting events. The abstraction fits both mechanically computational systems, and systems in which abstractions of observable human behaviour (of the same kind modelled by queueing theory) provide input events that drive the model.

The present work is differentiated from any of these models, because it neither models physics nor agents. Instead, it uses the minimal amount of physics necessary to produce a coordinate system in space-time, and then deals with specifications of how objects in this coordinate system *should* evolve deterministically in terms of their inputs. This might look like a simple synchronous language, but the approach does more because it uses a *single* unified coordinate system rooted in observations specified by international political structures, rather than a coordinate system declared by local fiat inside a single computer with its own authoritative clock. Specifications of what a system *should* do in HBCL are parametrized only in observations anchored in this coordinate system, and whether the system in fact *does* do this depends on effective engineering animating this human imperative. If we cannot control the determinism of the inputs, then the state evolution will not be deterministic *from the outside*, but it will be deterministic *given* the inputs.

2.8 Intuitionistic type theory

In this thesis, we have used the Coq proof assistant to implement a programming language. It happens, for reasons discussed later, that Coq uses an intuitionistic logic. Martin-Löf's Intuitionistic Type Theory [133] is the technical underpinning to such logics. Usually, we will only need to reference these topics indirectly through our use of Coq, but one concept from this type theory that we will use in its own right is that of a Σ-type; we explain what this is now because we will use the concept in the next section. A Σ-type is similar to a Cartesian product, but one in which the type of a component on the left affects the type of a component on the right. A very useful application of this is in constructing types which have a concrete *and* a propositional component. For example, if we have a predicate, `isPrime`, which takes a natural number argument and is true (inhabited) when that argument is a prime number, then we can construct a Σ-type representing prime numbers that is a pair of a natural number and a proof that the *particular* prime number is in fact prime. The proof has a dependent type, such that a proof that '2' is a prime number has a different type than that of the proof that '3' is a prime number. Inhabitants of this type can only be prime, since the presence of the proof term is proof of this *by construction*. This idea of propositions as types and proof as inhabitants is central to intuitionistic type theory and is closely related to the Curry-Howard isomorphism. We will have more to say about these matters when we discuss the Coq proof assistant, but for now, we state that we will use the lower-case σ-type to refer to one of these σ-types formed of a concrete and propositional component. Finally, we note that if our predicate is polyadic (it has more than one argument), then we can generate a *parametrized* σ-type, where the parameter of this σ-type is one of the arguments of the predicate, while the underlying concrete type is determined by the type of the second argument. This idea is used repeatedly in this thesis. At the simplest level, it is used to give types to the types and data of a subject type system. We go on to put the concept to work to dependently type a 'static semantic object', and ultimately to give a type to an HBCL 'reality', a coinductive type parametric in the coinductive type of possible input observations over time.

2.9 Logical systems

What are we trying to achieve with this heavy emphasis on formality? In developing reliable structures we need to do two things with a logical system: we need to show that a specification has particular properties that we identify as being necessary or desirable; and we need to be sure, with some arbitrary degree of certainty, that purported implementations of such logics actually fulfil their specifications. To have any hope of doing this consistently, we should know that our reasoning in each case concerns the *same* specification, so we must define and prove the semantic equivalence of a stack of abstraction layers using *morphisms*. This produces a strong definition of the *correctness* of an implementation. This notion of correctness expresses the correlation between the axiomatization in a logic of an intuitive specification in a human mind, and the axiomatization of its realization in the physical world according to physical laws of Nature. Logical systems have common elements that facilitate this. They possess a way of defining structures: into these structures must be encoded anything we wish to reason about, whether it be the physical world, a piece of mathematics, or any other definite structure humanly conceivable. They then provide a means of making statements about those structures, producing *formulae* or *propositions* of the logic, which may *quantify* over the possible members of definable structures. Finally, they provide some method that may allow us to establish the truth of these formulæ. Where formulæ range over finite structures, exhaustive state exploration of every possible value of such formulæ is at least theoretically possible. For infinite structures, and, in practice, most finite ones, some sort of deductive technique is required. This takes the form either of a formal deductive logic, which allows us to determine what are valid arguments of the logic through its rules of inference, or a supplementary technique over representatively exhaustive state exploration, such as a coverage analysis.

We now review logical systems in some detail, and conclude the chapter by drawing them into a unified comparison table with the type of systems we are interested in modelling in HBCL. Full HBCL, as a programming language, could be used to implement a proof assistant itself, so it is relevant to explore whether our ontology-driven approach to HBCL delivers any fresh insights to incompleteness issues.

2.9.1 Logical propositions, formulae and embeddings

In this section, we discuss how structures in the information domain — and claims about them — are formalized. We do not say anything about proof yet; that is the subject of section 2.9.2. By 'information domain', we mean the world of abstract structures that might[30] have a Platonic existence independent of any physical extension. Such structures need not have epistemic immanence.

2.9.1.1 Predicate logics

This is no place to rehearse the history of logic. We simply observe what is generally meant by predicate logic. Predicate logics are quantified extensions of propositional logics [79]. Pure propositional logics deal with structures and inferences over truth-valued variables, but cannot accommodate variables of some other type, such as concrete structures of mathematics, physics or information. Predicate logics allow just this, so statements that have a truth value can be made about types of things that are not simply propositions. For example, in arithmetic, $x+y=4$ is a statement that is true for $x=2$ and $y=2$, or $x=1$ and $y=3$, but false for $x=2$ and $y=3$. Such expressions (which are *well-formed* fomulæ) do not have truth values until their variables (in this case x and y) are bound to values, which in this case are integers. Crucially, we may then say things about all possible values that may be bound to variables, by *quantifying* over them. For example, we can say for all possible values of x(or $\forall x$), something is true. For example, if we allow our integers to be signed, we can say that for all x there exists[31] some value of y (or $\exists y$), such that $x+y=4$. To put it another way, $\forall x \exists y \; x+y=4$ (we neglect saying explicitly that x and y are signed integers). This statement happens to be true, so it is a *theorem*, given suitable axioms of arithmetic. We defer discussion about how we can be convinced this is true until section 2.9.2.1.

We have not said much about what x and y are. Implicitly, $x,\, y \in \mathbb{Z}$, and we have taken for granted that predicate logics allow constants such as 2 or $\varnothing$, and operators such as $+$, or $\{\}$, which encode relationships between structures. But herein is a potential problem. This works brilliantly if we are only interested in integer arithmetic, but it cannot deal with arbitrary structure, unless that structure is encoded in some way.[32]

In practice, mathematicians classically use set theory to describe nearly all mathematical objects, including arithmetical ones. Sets have the advantage of being very simple[33] constructions. With an empty set object, and the ideas that a set can contain any

[30]Whether they *do* or not depends on one's philosophical tastes.

[31]This quantifier can be controversial, especially in intuitionistic logic, where it is taken to mean 'it is possible to construct', where the proof is expected to give the means of construction.

[32]Attempts of this sort were brought to an abrupt end by the discovery of Gödel's first incompleteness theorem. Gödel's encodings of logic in arithmetic are known as *Gödel numbers* [88].

[33]In practice, set theory has to be more carefully defined, such as in ZF set theory, to avoid the logical problems of a set being a member of itself, which gives rise to Russell's paradox in 'naïve' set theory.

number of other (unordered) sets, arithmetic, and most of mathematics, can be encoded.

This is the perennial problem of *embedding*. In computer science, the preferred foundational system for expressing types of structure are type-theoretical or category-theoretical, although this does not remove the problem of assigning meaning to the structures we define in these structural schemata. We will return to this subject in section 2.9.1.2. For now, we talk about predicate logics quantified over an implicit placeholder for whichever type of structure we are interested in. Otherwise, it is not clear from the term 'predicate logic' whether we have some particular structure in mind (frequently arithmetic or set theory is assumed), or whether we wish to speak about it as quantified[34] over diverse structural systems.

The kind of predicate logic just described is *first order*. In other words, quantification is allowed over the concrete structures (such as integers) for which predicates may be defined. There are many other logics of 'higher' order, which allow quantification over first order predicates. We need not go into this here, but will return to the subject when we discuss proof in section 2.9.2. Such higher order logics can be compressed through suitable encodings and with suitable axioms into first order logic, so long as we do not insist on a semantically unified notion of truth between first order logic and the logic that it is encoded into it.[35]

2.9.1.2 The assignment of meaning to logical structures

We are about to talk about embedding programming languages in logical structures. The embeddings of languages are often spoken about in terms of 'deep' and 'shallow' embeddings: 'deep' means that all structures of the language (or whatever else we are embedding) are describable in a plain data structure which has no semantic notion of the import of these structures — and if such encoded structures are themselves logics, any truth-functional structures in that logic have no special status in the encoding logic; 'shallow' means that some of these structures have been borrowed from the containing logic, so it is impossible to quantify over all possible programs in the formalized language without formalizing the whole logic itself in some external structure. This understanding of the terms 'deep' and 'shallow' is widely used in the literature without reference. The first uses of the term embedding itself (without description as 'deep' or 'shallow' in this way) is given by the Oxford English Dictionary as first occurring in the fields of mathematics and linguistics [9]. Its use in computer science seems likely to have spread from here or from the mathematical idea of isometric embedding, although arguably it is the background notion of Gödel numbers that has propelled this usage.

We next discuss briefly what it means to embed *anything*, not necessarily a programming language, in a logical structure. The characteristic of an embedding is the process

[34]Further consideration of the irony of this word in fact reveals some useful insights: see section 2.9.2.
[35]This follows *a fortiori* from Tarski's undefinability theorem [177].

of modelling something, be it a language, a logic, reality or anything else in some host medium, which can also be any of these. The key problem in formalizing anything showing organized structure is that there are often many good ways to do it, so none of the formalizations *is* the thing that we are formalizing. One can adopt principles for choosing good models for things, for example, Newton's least sufficient explanation principle [148],[36] but this does not imply that a particular such model thus obtained is canonical. There is a duality between an assumption that something is *true* and the assumption that some model or other adequately and succinctly describes a phenomenon we have identified.[37] It would be perverse to axiomatize natural numbers by taking, say a Peano axiomatization [153], and then develop all of arithmetic by only counting even ordinals. It might be an adequate model, but not good enough to tempt us to elide the model with the very thing it is modelling. The difficulty of having many models of this sort was famously posed by Benacerraf [26]. This is the problem of underdetermination, which he threw into sharp relief by highlighting the contradictions that occur if one person regards Zermelo ordinals as *being* natural numbers and another regards von Neumann ordinals as *being* natural numbers. When set-theoretical questions are asked about one number being a *member* of another, different results are obtained, none of which says anything useful about arithmetic.

2.9.1.2.1 Temporal logics in predicate logics We observe that temporal logics are born of a particular ontological interpretation of modal logics. The expositions of temporal logics that we review in this section do not stress the ontology, but in talking about time, which is a physical phenomenon, they implicitly acknowledge one. A modal logic [31] is one where truth or consistency is parametrized on some external variable. That variable may be probabalistic or express some imperative or impossibility, but the modality we are interested in is that of time. When a modal logic is formalized in an ordinary logic, the modal dimension is by necessity deeply embedded, as there is no concept of time built into the formalizing logic in which to embed it shallowly. Other modalities require a formalization of necessity, uncertainty and possibility, while temporal logics require a formalization of time. Temporal logics thus have a clearer ontology and are beset by fewer epistemic difficulties than other modal logics.[38]

Time is an infinite modality. The predicates and quantifiers of temporal logics range over legitimate evolutions and invariants of a temporal system. Temporal logics are capable of asserting predicate formulæ over executions which can never be *satisfied*: in other words, there are no possible traces for which the formulæ evaluate to 'true'. For example, a formula can express the assertion that a set of coordination constraints result

[36]p.794

[37]Model theory is founded on this notion.

[38]We are assuming in saying this that we have a more intuitive grasp of what time is than what probability or necessity are.

in a system that is free from deadlock: if the system does in fact suffer from deadlock, the logic is describing a history of the system that cannot exist, even though it might be possible to construct an execution model that runs as far as the point where it deadlocks or otherwise causes the temporal formula to evaluate to 'false' .

Temporal logics are used to specify properties required of models that must hold over a number of states of the underlying model. The major temporal logics relevant to computer science follow Pnueli's key 1977 paper [158], in which developments in modal and tense logic in the preceding decades were applied in a particularly concise way, using the the modalities 'invariance' and 'eventuality': this is the essence of Linear Temporal Logic (LTL).

Pnueli's breakthrough led to a controversy in the early 1980s concerning the development of more expressive temporal logics in computer science. There are essentially two approaches: those where formulæ have an implicit universal quantification [73] over all possible executions (the linear logics, such as Pnueli's original model), and those where universal and existential claims about the properties of paths can be freely mixed (branching time logics). Neither is more expressive than the other. Linear logics such as LTL cannot make claims about multiple possible futures, so are unable to make statements such as 'There exists an execution path in which such-and-such is eventually true'. Branching logics such as Computation Tree Logic (CTL) [71], on the other hand, cannot quantify over execution paths, since they would be trying to quantify over something which is by definition never closed. They cannot make statements such as 'There exists a path on which it will at some point become the case that such-and-such will be true for evermore'. CTL* [72] is a superset of LTL and CTL: it is very expressive, but suffers from being a large language and difficult to use.

The final temporal logic system we will mention here is the Temporal Logic of Actions. TLA is a logic system devised by Lamport [125] which uses Pnueli's original modalities of invariance and eventuality, and replaces reasoning about paths and executions by considering 'actions', which express a possible next state in terms of the preceding state. Lamport's motivation was to push as much as possible of the reasoning about the specification of a system into *atemporal* reasoning and the consideration of actions, using the (two) temporal operators as little as possible. This is a paradigm we embrace in the present formalization, although we do not use TLA.

As we saw earlier in this chapter, we are observing a dichotomy between expression languages and coordination languages in developing and formalizing our coordination model. The former have no conception of time at all; it is the latter languages for which we need a temporal formalism. However, it suffices to embed the notion of time directly in a coinductive type or predicate that represents the state of a system, or the required state of a system at each discrete time slice: there is no need to introduce an extra layer by first embedding a temporal logic. This does not prevent us from reasoning about this structure using a temporal logic: we could give this coinductive type as a parameter to a formalization of such a temporal logic, where that logic was expressed as a dependently typed coinductive predicate. However, the paraphernalia of temporal quantifiers from a separate temporal logic are not *necessary* in our development, since not only are we developing a language, not a logic, but also it is a language that instantiates the idea of time itself: we do not have to borrow the axiomatization of time from a temporal logic in a 'shallow' embedding style. We have all the quantifying power we need directly in the logic we use to formalize the language. We will now apply a similar argument to axiomatizations of semantics. We will see in section 5.1.1 how the same approach is consistent with the partitioning of HBCL into Pre- and full HBCL that we introduce in chapter 3.

2.9.1.2.2 Semantics If we were to axiomatize a semantic style prior to presenting the semantic rules of HBCL, we would be axiomatizing a special-purpose specification language or programming language, albeit one envisaged for interpreting, or *simulating* other languages. This signals that we are again in the presence of the Church-Turing thesis: nesting one deeply embedded programming language inside another language, itself embedded in a logic, would serve only to obfuscate matters. After presenting HBCL in chapter 4, we therefore embed HBCL directly in the logic of a proof assistant in chapter 5. The semantics of chapter 4 have to be presented in *some* style, and so we use notations that are as evocative as possible of dependent type theory. Underlying the typography is an XML schema from which a stylesheet generates the typesetting. This relatively informal presentation could be proven to be compatible with the Coq formalization if we chose to formalize the schema within Coq. We discuss why we might conceivably want to do this in section 7.4.5, but we would definitely not want to carry out our primary formalization in a proof assistant through the intermediation of such an empty structure: its only purpose is to provide a simplified type theory for the purposes of presentation.

Following this approach, the question of embedding semantics becomes one of finding a methodology, rather than of first embedding a semantics description language. In an evaluation style, the conclusion of a rule should show how an expression gives rise to a structure which is 'the answer'. The various premises correspond to the branching logic based on the structure that is found when breaking down the implicant of the conclusion and recursively applying semantic rules. In this style, the implicand of the conclusion is constructed directly from the implicants and implicands of the various premises. This is sometimes called a 'big-step' style, since no state is apparently kept between the beginning and end of the computation. The specification of functions and structures of the embedded language are entirely dealt with by the data structures in which they are embedded, and the behaviour of the functions thus encoded is animated by the functions of the embedding language.

We also use a σ-type paradigm to combine this style with an axiomatic style, as discussed in relation to the Hayes and Jones argument in section 2.4. The use of σ-types for strong specifications in general is a paradigm that pervades Coq; it is the basis of Sozeau's 'Program' environment [173], and is the subject of a chapter in Bertot and Castéran's tutorial book [30]. We do not discuss denotational semantics: first, because sounding the Platonic overtones of the denotational approach risks confusing the ontological themes we are developing with denotational baggage (evaluation semantics, mixed with a higher order type-theoretical predicated approach, come quite close to the territory of denotational semantics); second, our example expression language has only first order functions, so we do not need to use a model for the λ-calculus and thus reach for Scott-Strachey domain theory, the centrepiece of the denotational approach. Instead, we use the reduction semantics and higher order structures of a proof assistant directly.

2.9.2 Establishment of logical formulae

If we have a proposition such as $\forall x \exists y\ x + y = 4$ or $\forall A \forall B\ A \land B \to A$, how do we know it is true? Or to be more precise, how do we know it is a valid, or consistent theorem of the logic?

Before rehearsing common methods for becoming convinced of the validity of theorems, we ought to say what we think reaching a proof in one of these methodologies might maximally hope to establish. These matters are contentious, and are on the borders of pure philosophical enquiry. In his popular book, *The Road to Reality* [154], Penrose discloses his view of issues of Platonism in mathematics and related positions as 'prejudices', and since we are presenting our views on these matters only as contextualizing conjectures, we adopt the same approach, and ask the reader to take what follows as our philosophical 'prejudices' (they differ from Penrose's).

We take the view that one of the things that makes us human seems to be our ability to *choose* what our axioms are. All logical thinking — certainly from the ancient Greeks onwards — requires at least some axioms; Gödel tells us that the study of pure logic using logic itself does not let us escape axioms either [88]. Descartes quipped that 'Good sense is the most evenly distributed thing in the world; for everyone believes himself to be so well provided with it that even those who are the hardest to please in every other way do not usually want more of it than they already have.' [62][39] It is, however, all too easy to conflate the wisdom of how axioms are chosen in any sphere with every human being's innate understanding of physical causation and the consistency of physics over space and time: an evolutionary adaptation that is vital to surviving in the physical

[39]p. 5

world.[40]

We conjecture that one way of looking at physical axioms (the laws of Nature) and logical axioms[41] is by viewing a particular kind of self-formalization of a logic in a proof assistant as a *fixpoint of Nature* with respect to time and space that mirrors a *cofixpoint* characterization of a stacked logic such as Turing's Ordinal Logic [187].[42] If one uses a proof assistant's logic to show a morphism between part of physics modelling an execution of the proof assistant and a direct deep embedding of the proof assistant's logic in itself, then one has the capacity to escape the spiralling stack of logics that one obtains by continually adding axioms and 'larger' infinities of infinitely nested logics.[43] Physical causation can be considered an analogue of logical rules of inference, in a similar way that physical geometry is related to abstract mathematical topology. One can take the view that the pair of foundational theories should be arranged the other way round, and argue that the logic is prior to physicality, and that the universe is somehow the physical extensionality of some computation or other,[44] but this is largely a matter of conviction. For practical purposes, experimentally testable scientific law is something readily graspable without needing an understanding of the nature of the consciousness that is doing the grasping. One might even argue that our willingness to accept it seems to comes from reason as a biological adaptation to the physical world, whence come 'laws of thought'. The physical analogue of a proof-checker might escape *logical* axioms through accepting that laws of physics are invariant in reality, and appealing to the plausibility of a thought experiment in which this Quine-like [111] proof checker can check its infinite *physical* self-model by running the experiment an infinite number of times, each time stripping away a layer of simulation from the embedding.

A proof assistant can therefore be used to study logic as an experimental science amenable to the scientific method like any other part of science. Logic becomes a human labelling through axiomatization of the potential of physical structure: in this way of thinking, the key to accepting the validity of a logic is through the physical realization

[40]This line of thinking suggests a critique of the heritage of Kant's *Critique of Pure Reason* would be informative, but we resist the temptation to start a discussion on the history of rationalism, scepticism, and the history of logicism.

[41]It is tempting to pair logical axioms as laws of thought, in the tradition of Aristotle, Boole (notably in [34]) and others. This does not work if the axioms are arbitrary, as in Frege's axiomatization of first order logic [79], or if they are as good as arbitrary as far as experimental testing is concerned (if, for example, they contain existential hypotheses of unconstructable transfinite universes).

[42]The locus of the non-mechanistic part of the proof of Gödel's theorems, in this way of looking at things, is in the acceptance that the universality and spatio-temporal unboundedness of the laws of physics really holds.

[43] Disjoint areas of physics appear Turing-complete in this regard: one can build a Newtonian computer such as Babbage's analytical engine without needing Maxwell's equations or quantum mechanics. This is really a matter of analyzing the mathematic shape of physical actions and is the intuition of Gandy and Sieg, which we discuss in this chapter.

[44]The Zenil compendium [193] is a modern exploration in concepts of a 'computable universe' that originated with Feynman [74] and Deutsch [63].

of a proof checker, and belief in the accuracy of the self-model and the laws of physics.[45] The physical model is *prior* to the logic, not *vice versa*. Logical axioms are replaced by physical ones and discharged through morphisms. The idea has some resonance with the structuralist argument (Benacerraf [26] *et al.*) in the philosophy of mathematics, except that we are less interested in giving an account of what mathematical objects *are*, but more concerned about what can be believed about a logical system that might use the relationship between its own axiomatization of itself and its own axiomatization of its mechanical proof checker to infer something about the consistency of its logic up to the consistency of physics. We have no more space to devote to this here, so we leave it as a conjecture; it is, however, relevant to discuss contemporary developments in *verified stacks* [145] in an introduction to proof assistants such as this, and we therefore return to this topic in section 2.10.4.

2.9.2.1 Deductive and exhaustive state analysis

Exhaustive state analysis is based on the principle that in seeking to show the veracity of a statement, whether quantified over propositional or concrete domains, one can validly conclude that the statement is true if it holds for all values of its domain. A truth table in propositional logic is the simplest example. Model-checking suffers from problems in state explosion and thus uses graph analysis and reduction techniques to achieve coverage of the state space [81]. Acceptance of the mathematical techniques is implicitly underpinned by deductive reasoning, although that deductive reasoning occurs statically at the design-time of the tool rather than during individual verification exercises. While model-checking is a widely used approach to verification, the present work is not about model-checking, so we do not consider it further.

Deductive systems use explicit rules of inference to validate each step of an argument. Exhaustive enumeration can be used in the sense of case analysis (more so in classical logics), particularly where inductive types are involved. However, the key to a deductive system is the way that complex conclusions are built up using rules of inference.

2.9.2.2 Proof assistants

The functionality of a proof assistant can be divided into two: the proof-checking kernel is the most important component; the other features help to *find* proofs. The kernel checks whether an encoding of a logical formula and a supposed proof of it are legal structures within the logic, and whether the ‘proof’ is a valid proof. It is a task of applying the rules of inference of the logic in a mechanized form: this part of a proof

[45]The distinction between a hardware or software proof checker is immaterial: the complete physical model of a proof checker is a computer with a program physically encoded into it. How much of this is on a hard drive (or for that matter punched cards) or burnt into silicon is neither here nor there.

assistant is typically a very small piece of code, which makes it surveyable by human beings, so that it is possible to become convinced of its correctness by direct inspection. A proof checking kernel that meets this condition is known as satisfying the *De Bruijn criterion* [21].

The vast majority of the code base of a proof assistant is not involved with checking proofs at all, but in helping the user to find them using tactics. Since any useful logic is undecidable, it is impossible to write a tactic that will always find a proof, if one exists. However, such tactics can still be extremely useful, since finding proofs is usually a case of deciding to apply formulaic patterns of reasoning. The rôle of insight is usually in deciding which tactics to try in an unfamiliar situation. The automatic application of tactics, particularly once they are composed into more complicated composite tactics, can save an enormous amount of time. Crucially, the impact of a bug in this part of the proof assistant is limited to the inconvenience of the assistant crashing, or producing a 'proof' which is rejected by the checking kernel. It does not risk validating faulty proofs, which would be disastrous.

The verifying kernel of proof assistants, in effect, reduces a question of whether something is a valid argument in a logic to one of running a physical experiment, whose axioms are not rules of inference, but laws of physics. The assistant is not written in terms of physical laws, but in the semantics of a programming language. However, in compiling a programming language into binary form for a particular architecture, one is reducing the problem to one of physics, albeit that the semantic model of hardware is a finite state machine model of a configuration of matter, engineered to evolve bisimilarly with a physical model up to discretization. The physical experiment involves setting up the proof checker on concrete computer hardware to verify the proof of a theorem, with the experimenter starting the program and leaving the computer to its own devices until it halts or they lose patience with it. The computer's state under these circumstances evolves according to the laws of physics. We have a hypothesis, namely that the proof script, when loaded into the machine, will cause the proof checker to issue a QED, which we test according to the scientific method. We want to be able to draw the conclusion that a successful outcome of an experiment (or verification attempt) shows either that the proposition was correct, or that the proof was faulty. This is emphatically not the same thing as proving that the proposition was false. To do that, we would need to state the negation of the proposition and produce a verifiably correct proof of that instead.

In critically evaluating such an experiment, we have to discharge the obligation to provide a convincing argument that the configuration of the physical apparatus is consistent with the semantics of the logic in which the proposition is stated. As well as concluding that the proposition encoded in the theorem is true, another possible conclusion to draw from a successful *or* unsuccessful proof verification run would be that some axioms were wrong — either that the semantic model of the hardware was wrong, because the physical model of the hardware's specification was inadequate, or the particular instance of the hardware was faulty, or that we were wrong to quantify the particular laws of physics on which we relied over all space and time and scales, or the logic used by the proof was inconsistent, or that the whole experiment was hopelessly misconceived, in that it did not in any way mirror the semantics of the logic that our intuition told us we thought we had embedded in the physical system.

To have confidence in the outcome of a verification run, therefore, we must become subjectively convinced that none of these explanations was justified. This is the physical subjective analogue of becoming convinced that axioms that are ultimately based on laws of thought are true, notwithstanding that Gödel's incompleteness theorems show that we can never prove them in the logical sphere by using their own apparatus. Still, laws of thought seem safer when they can be put in a structure morphism with a physically deductive system that is axiomatized on what are apparently laws of Nature.[46]

In this experimental formulation of a proof-checker, one therefore cannot ignore what Dijkstra dismissed as the 'pleasantness problem' of formal methods [64]. The idea of pleasantness is that of whether the formal specification of a system accords with our intuition of what we wanted it to say. We have already considered how such problems might be overcome by the use of parallel axiomatizations shown to be equivalent under structure morphisms. However, even if we choose to eschew such issues as outside the scope of formal methods for general specification work, we cannot avoid the matter where gaining confidence in a proof-checker is concerned. Even with a proof checker that satisfies the De Bruijn criterion, the complexity of the reasoning that allows us to map the result of a physical experiment to a logical deduction draws us inexorably to formalizing the computer and every tool we used in designing it into a logic that we can check with computers themselves.

In considering the validity of projecting a computer's model of itself into the logic that we are checking in that very computer, we cannot avoid the need to become convinced that what we physically have conforms to our logical model of it. This is similar to an engineer reaching a judgement that the bridge standing in front of them is a reflection of the mathematical model that claims that it will not fall down, except that a bridge does not calculate things about itself, other than its own structural integrity, or lack of it. That mathematical model is a fusion of a set of physical axioms and a particular conformation of matter, and the engineer must be convinced that both are justified in relation to the bridge. If we accept the indispensability of considering the pleasantness of a computer's model of itself, we have some philosophy to do, but we have a way of approaching Gödel that identifies the problem of known self-consistency with a human

[46]While the mere idea of a structure is an abstract one that only exists in a *logical* system, it seems plausible that the injection of *physical axioms* might be enough to bootstrap the formalization of a logic with the most plausible external axioms possible.

belief in the fitness of models and the quantifiability of physical laws over space-time,[47] and does not leave us appealing to the Platonic existence of mathematical objects and inscrutable laws of the human mind.

2.9.2.2.1 Particular proof assistants We now come to discuss individual proof assistants. There is a bewildering array of tools available: a representative but very non-exhaustive list includes those based on classical higher order logic such as PVS [151], HOL [90], and the major logics of Isabelle [149]; dependent type-theory-based tools including Coq [68] and Agda [150]; and environments geared to pure-mathematics such as Mizar [185] . Again, this is not the place to write an encylopædia of the history of proof assistants: such synoptic works exist, those by Geuvers *et al.* are particularly informative [85, 86]. Rather, we will focus our discussion on two of the most well-developed and popular tools, which together are representative of a wide class of available systems: these are Isabelle/HOL and Coq.

Isabelle is a meta-proof assistant. It uses LCF [89] as an abstraction that can capture the structures and inference rules of various different logics. The most widely used of Isabelle's logics is Higher Order Logic (HOL), which is based around the typed λ-calculus, and has a polymorphic type system. Its logic over these structures is classical, in that it admits the excluded middle as a valid rule of inference. It has predicate types, but these cannot be used for explicit subtyping.

Coq, by contrast, contains one logic, based on dependently typed logic and constructive type theory. It implements an extended version of the Calculus of Constructions [56]. It has an intuitionistic logic, in which all proofs must be constructive: it does not permit classical axioms unless these are explicitly admitted by individual developments. It has a dependently typed type system which encompasses both concrete and propositional types. Proofs are functions with the type of proposition that they prove; arguments are implicants and conclusions are implicands. It is impossible to construct objects of the type '`False`', an empty propositional type, since it has no constructors. False propositions are ones with no inhabitants. This view of propositional functions as proofs of their types is rigorously based in the theory of the *Curry-Howard isomorphism* [172].

[47]In other words, the laws of physics here, now, are the same as the laws of physics somewhere else, next week.

2.10 Verified compilers and the preservation of semantics under transformation

Both Isabelle and Coq possess facilities to export verified compilable code, which is produced from the executable subsets of the logics.

For simplicity, in the following discussion, the subject of compilation is not a syntactic entity, but a member of a semantic domain of static semantic objects. Membership of this (predicate-qualified) static semantic object guarantees that it can be animated in an interpreter function of the operational semantics from within the formalizing logic that exports compilers. The reason for discussing the topic like this, removing any syntax that could violate static semantic rules, is that it allows us to define compilation neatly as a monomorphism.

The process of compilation is one of translating one structure, which is a member of the set of valid programs in a high level language, to another, which is typically a member of the set of valid programs of a lower level language. The dynamic semantics (a function) of the lower level language should be a monomorphism of the dynamic semantics of the higher level language (another function) with respect to equivalence relations of the inputs and outputs as prepared for, and output by, each set of semantics. Ultimately, the task of a compiler is to produce binary code that can be processed by a piece of hardware (which itself has semantics). The task of the verified compiler developer is to prove that the compiler function is indeed a monomorphism. It is better to show this monomorphism using the predicate characterizations of the semantics than to show it using their operational counterparts, since this separates correctness of compilation from the implementations of the virtual or physical machines on which the static semantic objects are run.

Unfortunately, the big snag with this approach to verified compilation is that there are no chains of common axiomatizations that allow this to be done from code all the way to hardware and deductive hardware verification, although there has been some recent resurgence of interest in Moore's idea of *verified stacks* [145], which is an exciting area of research. The same snag applies *a fortiori* to finding a *verified* proof assistant: at the moment, no comprehensive solution is available. We therefore use the code export facilities of a proof assistant in the present work, even though this leaves verification gaps. These logical lacunæ are problematic in making a safety case to use the results in real safety-critical systems: they leave much scope for further work in proof tool development.

2.10.1 POPLMARK

More concretely, the PoplMark [19] challenge invited submissions of formalizations of a variant of System F [41], with specific challenges being given. The challenges specified the meta-theoretical properties that should be provable, and the animation of the language that should be producible in a compiler or interpreter implementation. The interesting part of the PoplMark challenge is not so much the particular challenge that the authors devised, but that it attracted a number of responses, which in sum substantiate the wider point that large scale formalization and production of verified tools is indeed practicable.

2.10.2 Isabelle-based

Isabelle has been used to verify compilers for some time, for example, in Strecker's verification of a compiler for a subset of Java in 2002 [175]. There has also been a more general effort by Berghofer *et al.* to produce tools that provide executable HOL code from predicate definitions [27,28]. Hol code maps relatively easily to Haskell by code extraction, and can then be executed outside the Isabelle environment [95] (this requires a chain of trust with the Haskell tools themselves).

2.10.3 Coq-based

Compcert [128,129] and its sister projects use Coq to build certified compilers, whose input languages are generally subsets of C. Consequently, a certified compiler could be produced for the present language by providing a certified transformation into the existing Coq formalization of the semantics of one of these C dialects. Chlipala has used Coq to produce a compiler for a higher level functional language using Coq [47]. While not a compiler as such, Tollitte *et al.* [182] have produced recent work on automatically producing executable Coq from predicates. This is relevant to the method advocated in this thesis because we have emphasized an approach based on developing predicates and parametrized σ-types in tandem. The method of Tollitte *et al.* has the capacity to take some of the drudgery out of developing a reference interpreter, but does not help if the finding of predicates is based on the intuition gained by developing operational semantics, and hence a reference interpreter, by hand.

2.10.4 Verifiability of proof assistants

We observed in section 2.4.1.2 how specification toolkits that check logical claims about specifications raise the problem of what verifies this verifier. Extended to general purpose logics and their implementing proof assistants, this is clearly an infinite loop, which touches again on the problems of verified stacks. Deductive verification has been applied to hardware for some time, too, (*e.g.* [137]), even though model-checking is still the dominant approach.

The most difficult element, however, is verifying one proof assistant with another: this is, strictly, impossible, owing to Gödel's second incompleteness theorem. However, various successful attempts have been made to produce self-verifying tools that get close to this limiting point. See, for example, the work by Harrison [98], Myreen [146] and Davis [61] for HOL-flavoured developments in this field. The main stratagem used for producing a development of this kind is to make the external axiom that is inevitably relied upon as small, transparent and readily convincing as possible. Harrison, in his study of self-formalizing HOL Light, uses a slightly stronger theory to prove that HOL Light is consistent, and employs it to prove that a slightly weaker theory is also consistent.

Barras produced a formalization of the Calculus of Constructions in Coq in 1996 [22], but little work on this has been done since. The infinite type universe hierarchy of Coq is in many ways more appealing for self-verification, since, according to our earlier conjecture, it is conceivable that it could enable the location of an experimental fixpoint by a Quine construction that is not dependent on ever more exotic infinitary axioms.[48]

Code exporters are useful development tools, and can be helpful for bootstrapping a proof assistant: however, to rely on their use suggests that we should verify them, which means another tool is needed to do this (this cannot be done by a proof-checking kernel — it is not within the remit of the De Bruijn criterion).

A final important but overlooked consideration in proof assistant development is that of multiple axiomatizations. The presence of multiple repositories of formal mathematics formalizing the same theories leads to the problem of captive dependency systems. Alama *et al.* review Public Mizar and Coq libraries [15], and examine how different systems deal with dependencies in these environments. In particular, they examine large formal WIKIS, which only admit theorems with verifiable proof scripts in some logic or other. This immediately suggests the Benacerraf problem we discussed above is at work again, since no formalization of what intuitively seems to be the same mathematics is necessarily canonical in any meaningful way. The challenge of relating such systems to each other in a usable way is an enormous and, so far, relatively unexplored problem.

In a more 'applied' field, multiple formal axiomatizations of the C programming language are being devloped by Krebbers *et al.* [122]. These multiple axiomatizations do not address the problem of proving that they are equivalent. This would require embedding heterogeneous logics within each other. We contend that the best confidence in such axiomatizations would be gained by deeply embedding each axiomatization within each logic, itself deeply embedded in a comparison logic, and then showing morphisms between axiomatizations in the host logic and each such embedded logic. If the logics are then permuted so that another logic is the comparator logic framing the morphism, then complete confidence in the consistency of these axiomatizations with human intuition and with each other could be approached in a limit, as the number of

[48]We would like to investigate how such an approach might relate to Wittgenstein's problematical critiques of Gödel [24, 76, 77, 165, 192], but this lies utterly outside the scope of the present task.

such axiomatizations and logics is increased. This is an idealized possibility in the spirit of a grand challenge of the kind described by Hoare [110] or Moore [145], possibly one that is rather too grand.

2.11 A remark on fault tolerance

A simple strategy for synchronously removing single points of failure can be implemented by applying overlapping replication transformations, following the original observations of von Neumann in constructing arbitrarily reliable systems [189]. More modern and sophisticated fault tolerance techniques can be used at an asynchronous level of abstraction, but simple synchronous techniques are very robust and suit a synchronous language.

2.12 Assembling the parts

We have now reviewed a variety of formalisms, including programming languages, specification languages and proof frameworks. We now explain the interaction between these kinds of systems, using Table 2.1. Each formalism in Table 2.1 is compared using the following five headings:

- **Abstracta** In this column we give a natural-language description of what entities the formalism is trying to describe, such as a type system, time, computational processes, or communication channels. The connection between the symbollic treatment of these phenomena in ASCII or underlying mathematical structure and the thing in reality which they are trying to describe is almost never formalized, with the exception of the TTA and Pre-HBCL.

- **Concreta** In this column we describe the type of things that instantiate the abstracta. This is usually some sort of text file, but, except in the case of HBCL, the text and the mathematical structures it respresents have only an informal relationship to identifiable entities in reality. 'Abstracta' and 'concreta' are standard philosophical terms. In our formalization of HBCL, we use dependant typing to introduce pairwise relationships of abstracta and concreta, where a more concrete form is obtained by supplying a Curried argument to a functional term.

- **Executable functions** In this column we answer whether the formalism in question provides a semantics for evaluating functions that are equivalent to (or can express a subset of) one of the standard axiomatizations of Turing-completeness, such as the λ-calculus.

- **Predicates and propositions** This column categorizes the formalisms according to whether they have a semantic notion of truth and an ability to quantify over some (sub)sets of its abstracta.
- **Deductive apparatus** This column asks whether the formalism in question has a semantics of valid arguments for establishing the truth of predicates over the abstracta. For intuitionistic formalisms such as Coq exploiting the Curry-Howard isomorphism, the semantics are the same as for executable functions. It is, however, impossible to have Turing-complete executable functions with a sound deductive apparatus, as this would imply an (impossible) solution to the halting problem. Coq's executable functions are thus limited to being primitive-recursive, and extra axioms over coinductive structures, which are unexecutable in native Coq, are needed to deeply embed Turing-complete languages.

Formalism	Abstracta	Concreta	Executable functions	Predicates and propositions	Deductive apparatus
B method and Event-B	Abstract machines (*ad hoc* states and events)	Statically compliant text	Yes	Some, first order	Limited
Time-triggered architecture	*ad hoc* observations with respect to GPS time; coordination protocol	Compliant specifications	No	No	No
Hierarchical timing language	*Ad hoc* types and time; program; module; mode; task	Compliant specifications	No	No	No

Formalism	Abstracta	Concreta	Executable functions	Predicates and propositions	Deductive apparatus
AutoFocus	Components; ports; channels; streams. Discrete clocks related by stream compression factors	Compliant specifications	State transition function expected in *ad hoc* logic	No	No
Verilog	Modules, wires. Event-driven	Statically compliant text	Limited	No	No
VHDL	Ports, processes, entities, architectures, libraries. Event-driven	Statically compliant text	Limited	No	No
Esterel	Axiomatization of time allows simultaneity; signals; modules	Statically compliant text	Limited	No	No
Actor model	Actors; messages	Compliant specifications	No	No	No
Hume	Boxes, wires	Statically compliant programs	Yes	No	No
Petri nets	Places, transitions, tokens	Compliant graphs	In basic nets, not directly; in some extensions, yes	No	No

Formalism	**Abstracta**	**Concreta**	**Executable functions**	**Predicates and propositions**	**Deductive apparatus**
Communicating sequential processes (and relatives such as CCS, ACP, π-calculus)	Events, processes	CSP models	No	No	No
SystemC	Ports, modules, processes, arbitrary C++	Valid C++ programs using SystemC libraries	Yes	No	No
Occam	Processes, channels, sequence and parallel operators, guarded commands	Statically compliant text	Yes	No	No
Erlang	Data types, function closures, processes	Statically compliant text	Yes	No	No
Linda	Tuplespace and processes operating on it	Program in a compliant implementation	Dependent on implementing language	No	No
Linear temporal logic	Propositional variables and time	Compliant expressions	No	Yes	No
Temporal logic of actions	ZF sets, time	Statically compliant text	No	Yes	With TLA+ and an implementation, yes

Formalism	**Abstracta**	**Concreta**	**Executable functions**	**Predicates and propositions**	**Deductive apparatus**
Isabelle/HOL	Higher Order Logic	Statically compliant text	Not directly	Yes	Yes
Coq	Calculus of (co-)inductive constructions	Statically compliant text	Yes, but primitive recursive	Yes	Yes
Pre-HBCL	Spacetime coordinate system; timed types; object identifiers; memories of timed types	Specific measurements or measurable quantities in physical reality; fixing values of timed type memories	No	No	No
Full HBCL	Pre-HBCL concreta plus box specifications and FIFOS	Instances of full HBCL structures and their mapping to Pre-HBCL concreta	Yes, boxes and FIFOS	No	No

Table 2.1: Specification formalism comparison matrix

The Curry-Howard isomorphism is implicitly present in this table: concreta and predicates are related in the same way as executable functions and logical deductions.

We can see that formalisms in Table 2.1 spread across many columns, but none achieves completely satisfactory coverage. At one extreme, programming languages specify computational procedures, while at the other, the formal logics of some proof assistants have no, or very limited, executable functions. Specification toolkits, such as the B-method, satisfy most of the columns, but such monolithic solutions lack generality, and cannot easily be extended to axiomatize application domains or more general reasoning. In every case, the concreta of the formalisms lack an ontologically sound anchor to reality, relying instead on informal associations between specifications and particular physical systems. In the rest of this thesis, we go on to carve a distinctive place in this schema for HBCL. We do this by focusing a deeply embededded ontology layer in a proof assistant, which maximises logical power whilst minimizing the complexity of what HBCL is. We will go on to see how this deep embedding provides a fast route to establishing language soundness, up to the soundness of the embedding logic. It is this apparent sleight of hand that has obliged us to examine in particular detail what a proof assistant is, and why it, and its logic, can be relied upon.

2.13 Conclusion

We have identified two strong dualities in this chapter: the duality of expression and coordination constructions and the duality of physical and logical processes. We have seen a variety of synchronous programming languages and asynchronous models. However, we have not seen either a general synchronous model which is sufficiently parametrizable to describe simple and nested expressions and coordination structures, with potential language heterogeneity, or a unified synchronous formalism for high and low level structures. Further, we have noted that there are certain philosophical ambiguities and shortcomings among some asynchronous models that flow from a conflation of physical entropy with unpredictable human agency. The language we present in further chapters is a response to these observations. In particular, it uses a synchronous model that can accommodate expressions from NAND gates to very high level functional languages, but which retains the possibility of using simple hardware fault tolerance techniques to be applied at the macro-scale.

We have explained what can be achieved by both specifying and realizing resulting specifications using formal tools. We have discussed the limits on these processes, surveyed some of the tools available for the task, and brought this study together in synthesizing an approach to formalizing the semantics of a language.

Chapter 3

The Harmonic Box Coordination Language I: Motivation, examples and semantic preliminaries

On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' … I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage — *Passages from the Life of a Philosopher*

Reason is, and ought only to be the slave of the passions, and can never pretend to any other office than to serve and obey them.

David Hume — *A Treatise of Human Nature*

We now present the Harmonic Box Coordination Language. First, we explain why the language is *harmonic*. This is the essential feature that makes the language unusual and particularly suited to real-time applications. The choice of this temporal structure is very closely related to the other novelty in our approach: HBCL's *ontological* commitment.

3.1 The case for a harmonic and ontologically aware language

Ontology is a branch of philosophy that is concerned with describing reality. An ontologically aware specification language is not one that tries to make deep statements about reality, or becomes involved in the controversies of ontology, such as the existential status of mathematical structures. Rather, we suggest that designing an ontologically aware language is about choosing a mathematical structure that refers unambiguously to what is being specified. This is important in safety-critical engineering, since if multiple safety theorems need to be proved about a physical or data object, then the basic axes of understanding of the identity and properties of that object must be fixed relative to the logical system.

A five-axis system is sufficient: three-dimensional space, plus time and an identifier space. Once we fix a base coordinate system in space defined by these axes, we have a sufficiently rich fabric to specify complex systems with significant precision. The natural scales to choose are those of the proper time in Earth's geoid as witnessed by TAI: this fixes the first four axes. We suggest that the strongest form of human designation is that of object identifiers, which are rooted in international political structures (ISO/ITU OIDs). This identifier space has an infinite tree structure, so it is possible to graft a language's conception of object identity onto some suitable part of the tree (and in fact, to develop the semantics with the particular node to which it is grafted as a parameter). This way of looking at things leads directly to how we set up the harmonic properties of the language, which we now discuss.

Key term: **OID**

Object identifier. A standardized tree of natural numbers with a single, internationally recognized, root.

The idea of a harmonic language comes from two observations. First, that the specification of all computer hardware has somewhere in it a clock-conditioned oscillator. Second, that most of these crucial low-level clocks (as distinct from the rather inaccurate real-time clock in most computers) are mutually incoherent: this means that computers have to model communications between themselves asynchronously, which presents formidable difficuties to engineering fault tolerance in distributed systems.

The processes of a low-level computer design are clock-driven, and typically observe the electrical potential of some conductor at a time determined by the main oscillator. If the specifications of what computers should be doing are to be universally composed, it follows that some single clock is required. This means we have to ask questions about what that clock is, and what we can know about observations that are timed according to it, which generates the need for the temporal part of our ontological approach. It is also necessary to uniquely identify the parts of the computer in a way which is dealt with by the scope of a local identifier space in the hardware specification language of a single computer. Finally, we justify the apparent problems with clock skew over long distances by observing that all we need to do is find *sufficiently* accurate *witnesses* of executions according to the global clock; the coarseness in the time scale compared to a local clock is dealt with using a pure functional programming approach, in which the separation of expression and coordination constructions makes for a system which is indifferent to such coarseness.

This approach also facilitates an extremely strong idea of 'correctness' for the purposes of engineering fault tolerance, because the idea of what is prescribed as being correct is enshrined in a solid ontological formalism, rather than in the implicit assumptions of a particular tool.

We now describe how we meta-instantiate HBCL ('meta-', since it is the instantiation of the language, not the things that might be described in it), tying it to a coordinate system in space-time that bears a static relationship to the Earth's geoid at the beginning of standard international atomic time in the 1950s. From a computer science point of view, the unusual step here is to specify the time base of the computer relative to a canonical clock ensemble, without the need to build a computer at all. The definition of a language to program a *single* computer according to a clock we do not own or control gives rise to what we would like to call an *externally timed* language. This gives an extremely strong conception of system correctness, because the logic is described relative to a very commonly accepted handle on physical reality, and the reliability of all computer systems we might build to implement a specification is judged by its ability to manifest its prescriptions of reality in a way that bisimulates this specification. We already have a potential computer — the Earth — and physics gives us its semantics. We animate the computer by arranging matter into components that represent and manipulate information efficiently, by labelling those components and by attaching significance to the information flowing into and out of them. As a bonus, by having a single clock in this planetary computer (a property it shares with most desk-bound computers), with managed skew, we can eliminate many problems in the reliability of asynchronous communication, since all communication is time-triggered. Even desk-bound computers have to deal with clock propagation considerations because of the very high frequencies at which their clocks run. A planetary computer programmed using an expression/coordination dichotomy can make do with much slower clock speeds. The computers evaluating the expressions run at normal speeds, but are tasked only with finding witnesses to Church-style functions, so their internal clocking is irrelevant to the HBCL specification.

However, if we are to regard our planet as a computer (a purely human designation and axiomatization), we must have a system for uniquely discussing its parts that is similarly ontologically well-rooted in international politics as — at least since the preeminence of Greenwich — physical coordinate systems are in space and time. Fortunately, such mechanisms already exist, and it is easy to graft a language's semantic conception of these parts onto this robust reference system, enabling HBCL to refer to individual interfaces and instances of logic in a temporal modality.

Oids are a tree of natural numbers with a single root recognized by the iso and itu.[1] It is one of the firmest referential anchors possible in the morass of humananity's common intentionality. An individual oid is just a list of natural numbers, and thus very easy to deal with in machine-assisted reasoning. One could imagine a mapping to a more meaningful structure, such as ldap[2], but this would add needless complication at this stage, and would introduce a great deal more arbitrary choices over the deep structure, over-specifying the model and obscuring any canonicity we might hope for, insofar as *any* axiomatization can ever approach canonicity. For demonstration purposes, we sandbox some oids with an arbitrary root. We give more details of the semantics of oid usage within hbcl in appendix A.2.

3.2 Informal exposition of a coordination language by conformationally representative examples

The static semantics of any language give the rules by which a model of arbitrary complexity can be built up from simple steps. A compiler accepts the next component if it is well-formed and its introduction is compatible in a specified way with existing components in the specification. The source-code static semantics of a language are relatively uninteresting: the behaviour of the model that the language represents depends on the *structure* of the object that can be built by applying the static semantics to some particular abstract syntax tree (ast), rather that *how* that object was built: the same object could be built by many different sets of abstract syntax trees and static semantic rules.

In the case of a coordination language, the most basic scenario is a single expression (Figure 3.1(a)). Next, is the idea of the parallel introduction of a box which has no interactions with any existing boxes: a parallel composition (b). The three remaining scenarios we will use examine how communications between boxes may be introduced. First, there is the pipelining of two boxes (c); second, there is feedback from a box directly to itself (d); and third, there is indirect feedback *via* a pipeline (e). These three structures appear very similar, but the structural difference is in whether the directed graph that describes the communication channels between boxes is cyclic at all. The simple feedback case allows cycles of one link, and the unrestricted case allows any directed cyclic graphs. The presence of unrestricted cyclic structures of these sorts is essential to allow a coordination language to represent unrestricted recursion; it therefore by implication introduces the possibility of non-terminating computations in some semantic domain, even though expressions within boxes are restricted to provably terminating

[1]These acronyms stand for the International Standards Organization and the International Telecommunication Union respectively.

[2]Lightweight Directory Access Protocol.

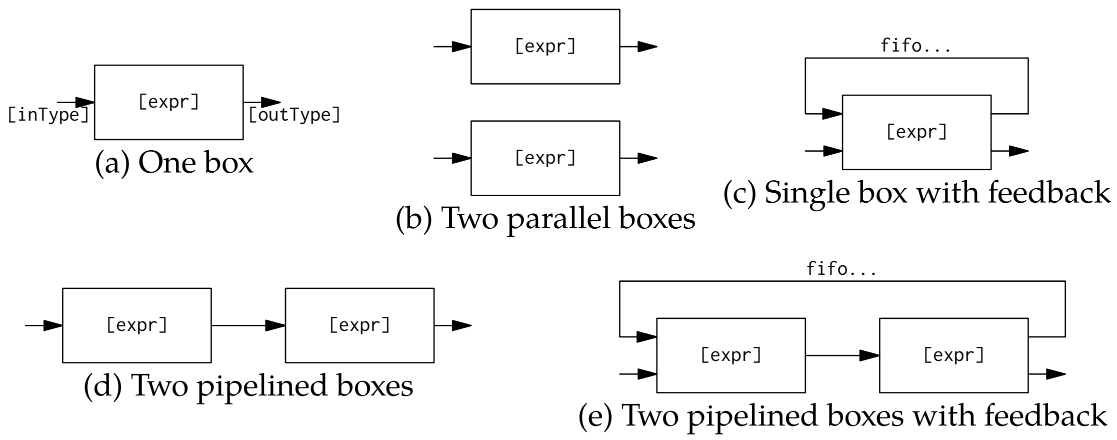


Figure 3.1: Case study summary

computations.

So in summary, our five, in an informal sense, 'canonical' examples are as follows:

1. A single box
2. A parallel composition of two single boxes
3. A pipelined composition of two single boxes
4. A box with feedback
5. A pipelined composition with feedback

These fundamental primitives of parallel and pipelined composition emerge from coordination mechanisms intended to adapt sequential functional programs to parallel implementations such as Caliban for Haskell [118, 178, 184].

3.3 Introduction to the structure of HBCL

The basic coordination structures of HBCL are exceptionally simple. There are two types of memories. Memories are just lists of values with some particular temporal validity. They maintain internal buffers which do not necessarily empty themselves every time they are read. The relationship of the values in the lists to the current time are determined by static parameters of the memory: the times to or from live (TTL/TFL). There are two different sorts of memories that execute at different times in the execution cycle. There are memories that stand between boxes (that contain the specifications of computations) and FIFOS, and there are memories that stand between FIFOS and boxes. The two are different because there is a unidirectional data flow in each case. The dynamic semantics are built up from an endless four-fold execution cycle which, by definition, never blocks on unavailable input, but by default produces empty output data in such circumstances. Boxes execute, then box-FIFO memories, then FIFOs, then FIFO-box memories, then boxes again. The types and every component in the system have a rational number frequency. The frequency of the top-level cycle is determined by the lowest common multiple of all the system components, so not every component executes on every cycle.

Definition: **Timed type**
A data type that has a frequency; its inhabitants have an absolute time that is an integer number of ticks having a duration of the reciprocal of that frequency.

Definition: **Memory**
A point in the identifier/time coordinate space with which is associated a timed type and a frequency.

Definition: **TTL/TFL**
Time-to-live or time-from-live. This gives an offset from the current time in whole numbers of cycles at the frequency of a memory. In the case of a time-from-live, the value of the memory is the given number of cycles after it was considered current, and in the case of a time-to-live, it is the number of cycles until it will be considered current. In the case of a sampled analogue quantity, these have natural meanings. In the case of intermediate quantities, their main purpose is to define the lengths of FIFOs, which have an implicit length of the time to or from live at either end. Internally, the time to or from live is a single signed integer, with times-to-live being negative and times-from-live positive.

The state trace is a coinductive structure that is completely determined by the coinductive input stream *observable* at the system boundary. This is expressed as a dependent type in the formalization, which gives a 'type' of reality, which only has one meaningful inhabitant in the real-world analogue: the way in which reality actually unfolds over time. This inhabitant provides what is meant by a *correct* execution, and is defined as long as the axiomatization of what is observable fits with reality (this is a human designation). The manifestations on the edge of the model are the way reality must evolve if the program, as an agent of causation, is *realized*. Realizing the specification thus involves constructing an argument that the semantic trace of a particular program has a *witness* in reality, up to some arbitrary degree of confidence. All of these programs are composable, since they axiomatically refer to different observations and manifestations in reality. We elaborate on the subtleties of this in the examples that follow, and in the formalizations described in the following chapters. Further commentary on the design features of HBCL can be found in appendix A.

Key term: **FIFO**

First in first out. A queue of data in which items are removed in the same order as that in which they are inserted. In HBCL, FIFOs are timed, so they consume and produce data at the same clocked rate, and have a deterministic length.

The structures that we have mentioned so far describe the Pre-HBCL coordinate system. Its embedding in a logic allows an application specifier to give a predicate specification of some system without being bound by the specific features of an executable language. Full HBCL is such a language, whose instances provide computational procedures that specify an operational causative link between FIFO-box memories and box-FIFO memories.

Definition: **Instance signature**

A set of Pre-HBCL structures over which a logical predicate or a full HBCL program may be specified.

3.4 Instance and library closure semantics

The way in which HBCL deals with code organization and repeating structure is novel and born of the same ontology-focused philosophy that occasioned HBCL's observation semantics. The key to giving effect to an HBCL program is not to compile it,[3] but to catalyse its accession to a suitable epistemology of engineering.[4] Therefore, when we write `linst` in global or `linst` scope, we are saying that here is a definition of a *unique* piece of logic that has a *unique* history. It might not yet be connected to observation points (this is done with `hinsts` and `cinsts`), but nevertheless by issuing this `linst` as an instruction, we are giving it an existence in some human frame of reference.

This is much like creating a car registration plate that has yet to be assigned to a car. The plate has a conceptual existence before, during and after being affixed to a car in some material form, and continues to exist in a historical sense long after the car has

[3]Although we have done so in this thesis, using an interpreter to simulate evolutions of state specified by the language.

[4]We could call it a 'database', but we avoid this because databases have become synonymous with the RDMS implementations that record the data and relationships between them, as opposed to the data themselves.

been destroyed — even when the database recording the plate has been closed down, forgotten, or destroyed.

This becomes more interesting when we add libraries, or `llibs` to the mix. Libraries have the effect of altering the scope, so that `linsts` declared within them have deferred instantiation: they must be referred to by some other `linst` which does not have deferred instantiation (or if it does, the instantiation will be deferred to the same extent as the instantiator). There are further scoping subtleties, which we discuss as we encounter them. Using just these primitive components, we can adjust the visibility of libraries depending on their placement, produce nested instances, and write down instance functors.

It is unusual to do without module and namespace systems, but we have chosen to eschew them in order to defeat a more serious problem: that of competing root namespaces. Module resolution systems usually rely on particular filesystem layouts or environment variables, but this requires subtle common assumptions between supposedly compatible system components that eventually fail: an unacceptable situation for robust systems. Java and some other languages piggyback on the domain name system in order to try to ensure uniqueness, but these systems rely on every contributor respecting conventions that struggle to deal with versioning and namespace re-use. At the bottom of this problem is that, just as competing system clocks cause synchronization problems, competing filesystem roots cause referential failures. The filesystem itself provides relative file paths and breadcrumb syntax (`../`), but programming languages seldom make use of this because it looks so confusing. Hosts can be configured to try first to resolve domain names as if they were subdomains of a local namespace, but to anchor the consistency of a system in such a subtle network configuration detail would hardly be robust.

The solution we adopt is novel, in that it need not reference a root, nor rely explicitly on exploration towards the root of a tree by the use of `../` or similar syntax. Rather, any name not resolvable at local scope is taken to be declared somewhere between the current scope and the root, and any compound name is one whose root component similarly occurs somewhere between the current scope and the root. More remote clashes with the the same names are thus masked. A reference to a library that is referenced but not declared at the current scope instantiates a functor: nothing in the library can be safely instantiated until it is put into a context that supplies another library containing instances with the correct input/output signatures.

A consequence of this is that any fully defined library or instance (*i.e.*, not a functor) can be grafted anywhere onto another library tree and will have exactly the same semantics. It also opens the possibility of supplying bisimilar implementations for missing definitions that satisfy some predicate we might associate with a signature. There is a very practical motivation behind this lack of an explicit root: bootstrapping a standard. While in its pure form, the ontology of HBCL is predicated on being grafted into the tree of ITU/ISO OIDS, it is likely that practical systems that might use the approach we have advocated would use different roots and different versions of HBCL. It is impractical to suppose that one could start with a canonical version of the system that would be widely adopted, and that a suitably robust organization framework be set up to administer the tree structure. By allowing rebasing that does not affect the standard, we allow one system to graft another system into its descendant OIDS by means of a mapping, if necessary with explicit conversion logic. This avoids the kind of internetworking standards disagreements that arose in the creation of the internet. Roots and their subtrees become a setoid structure susceptible to structure-preserving morphisms.

We return to this theme again in a different guise when we discuss multiple axiomatizations in heterogeneous deductive logics. In the extreme case, conversion logic and grafting of heterogeneous ontological schemata permit any hierarcical identifier system (not just OIDS or namespaces administered by supranational organizations). This extends to our spatio-temporal ontology too: any spatio-temporal coordinate system with known relationships to all other such systems in a transitive closure with proper time in the Earth's geoid are effectively equivalent and inter-operable.

3.5 Examples

3.5.1 One box

We introduce our first example by means of the schematic diagram in Figure 3.2. The whole configuration is contained in the dotted-line '`cinst`'box, called `negConfig`. It instantiates the configuration instance called a `negCfInst`, in the configuration library `negCf`. This is shown by the label in the top right of the configuration box. This type of configuration specifies that a logical instance be instantiated called '`negInst`', defined by `negatorInst` in the `negator` library. The label '`linst`' in the label on the upper left of the negator instantiation means that it is a piece of logic that comes into existence by virtue of the existence of the configuration. It exists for precisely as long as the configuration exists. The logical instance `negInst` declares an input memory (*i.e.* a FIFO-box memory, hence the subscript 'fb') called `posIn`. This is an observable bit of information with a fixed frequency that will be bit-flipped by the harmonic box '`bitNeg`' and manifested on the output memory `negOut`. Both types of the timed type `tDat` are from the `common` library. The hardware box '`negHwInst`' is an *hinst reference*. This means that it has an existence that is independent of the existence of the configuration. The hardware instance is a placeholder for a physical description of a piece of hardware that is incompletely specified: it does not 'know' what logic it should implement. The mapping of the logic to the hardware closes this open specification and the configuration demands that the logic be implemented (by refinement) in the particular piece of hardware. The configuration is only realized if this prescription is adhered to. The logical instance is also an open specification, in that it does not ‘know’ what values to observe, or where to manifest the bit-flipped result. Again, the mapping arrows close this specification.

We now discuss how this model is rendered in HBCL source code. First, we introduce a library containing only a trivial datatype.

Listing 3.1: A harmonic datatype

```
 1  llib commonDat {
 2
 3    type uDatTripleProt (bool * bool * bool);
 4
 5    oid type uDatTripleOid uDatTripleProt;
 6
 7    htype tDatTriple : uDatTripleOid [64];
 8
 9    type doublePairProt (bool * (bool * bool));
10
11    oid type uDatDoublePairOidT doublePairProt;
12
13    htype tDatDoublePair : uDatDoublePairOidT [64];
14
15  }
```

An ‘`llib`’ is a logical library. The one given here on line 1 is called ‘`commonDat`’. There are two points that need explaining here. The first is what is meant by a ‘library’ in this context; the second is why it is ‘logical’. Being a library means that instantiation of the structures it contains is deferred until a point at which an instance outside the scope of the present library instantiates it. The effect of deferred instantiation is transitive on scope enclosure, so any instance directly or indirectly enclosed by the library is deferred. This applies both to type definitions and to ‘logical instances’, which will be discussed shortly. Any contents of the library and any nested libraries can be instantiated from outside the scope of the library, as long as there is no intervening instance in the enclosure scope of the inner library to which reference is being made. The ‘l’ in‘`llib`’ stands for ‘logical’ and indicates that the contents of the library are data types or instances of causative logic.

Inside the ‘`llib`’ can be seen the declaration of the two types necessary for the negator box below: at line 3 there is a tuple of three Boolean values, while at line 9 there appears a pair of one Boolean with another pair of nested Booleans. The ‘`oid`’ and ‘`htype`’ keywords elevate this type to a full harmonic type as understood by the coordination language. The OID type declarations of lines 5 and 11 elevate these respective types to OID uniqueness, while the `htype`s of lines 7 and 13 elevate these to timed types, assigning

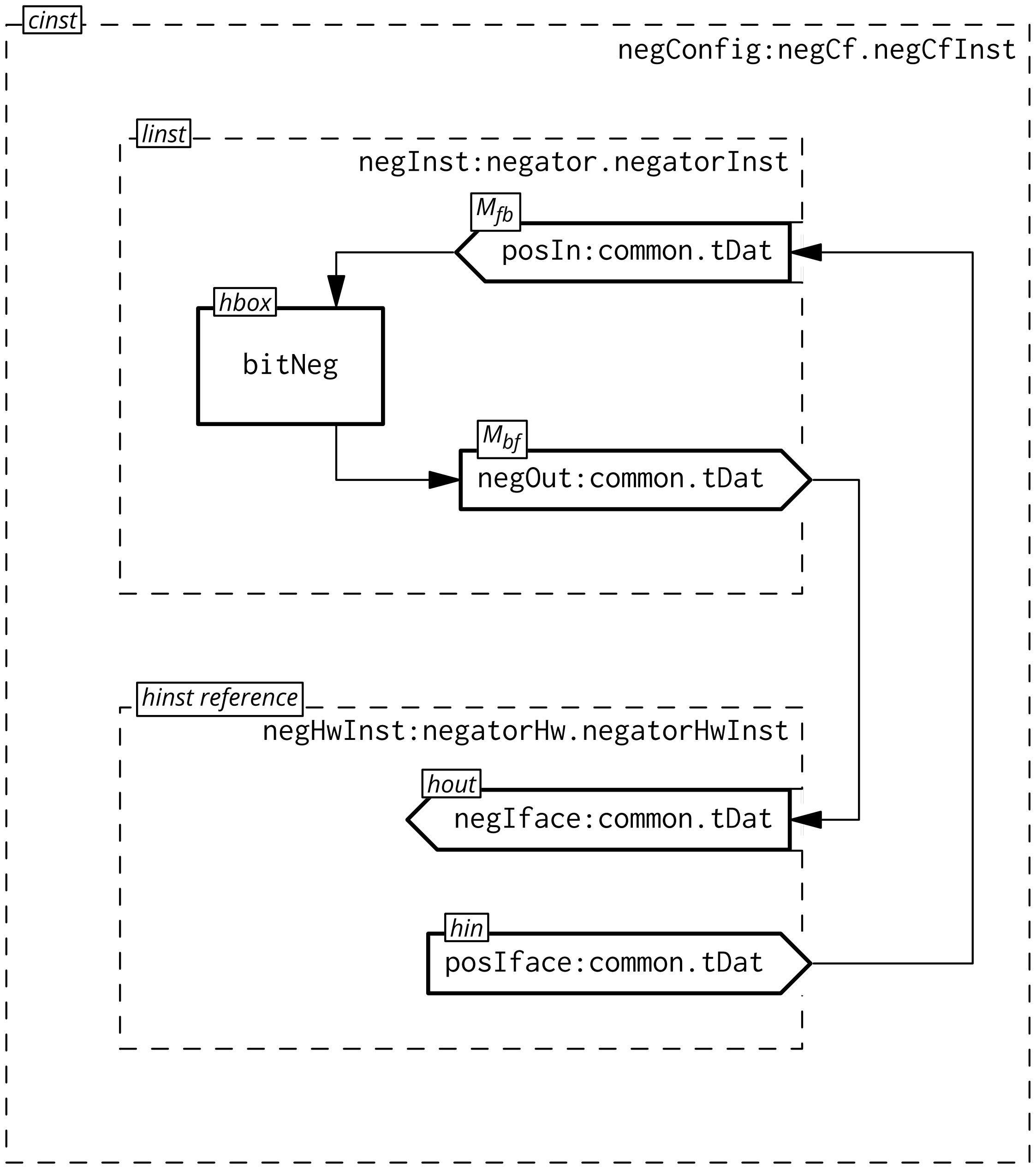


Figure 3.2: A negator instance scenario

a frequency. Frequencies appear in square brackets.

Next, we introduce a negation box itself.

Listing 3.2: A negating box

```
 1  llib negator {
 2
 3    linst negatorInst {
 4
 5      mem(fb) posIn : commonDat.tDatDoublePair [64, tfl(0)];
 6      mem(bf) negOut : commonDat.tDatTriple [64, ttl(2)];
 7
 8      observe {
 9        posIn;
10      }
11
12      manifest {
13        negOut;
14      }
15
16      hbox bitNeg : posIn -> negOut [64]
17      {
18        main : { posIn : ((bool * (bool * bool))) }
19         -> { negOut :  (bool * bool * bool) } :=
20           { negOut = ((not posIn.0.0, not posIn.0.1.0, not posIn.0.1.1)) };
21      }
22    }
23  }
```

The negator listing contains a negator library, called ‘`negator`’ (line 1), and a negator instance, called ‘`negatorInst`’ (line 3). This requires some explanation. When an HBCL program is *deployed* or *simulated*, it is done at the implicit scope of an anonymous instance. If the program is being deployed in the real world, it acquires a prepended OID dependent on the fiat of some external agency. We do not concern ourselves with this here, because we will always *simulate* (or, equivalently, *interpret*) HBCL programs in an ontological sandbox: a synthesized reality of our imagination with the root OID arc associated with this root instance, rather like the way that an instance of a *NIX kernel exists in its own reality, associating itself with the root of its file system namespace. Such distinctions are a worthwhile hazard of an ontologically aware language.

The library of types `commonDat` is declared at the same root instance scope, allowing the resolution of `commonDat.tDatDoublePair` and `commonDat.tDatTriple` to behave as expected in lines 5 and 6 respectively. This accords with the lookup semantics in which declarations of types in another library called `commonDat` would have been preferred, had they existed in the scope of `negatorInst` or `negator`. All such identifiers are thus made relative, so the semantics of an instance at a given scope are invariant, no matter onto which OID arc they may be grafted.

Line 5 declares a memory of the negator instance, which becomes qualified by the prepended arc of the instance OID. '`fb`' indicates that it is an input memory: that is, one that stands between a data flow from FIFOs to boxes. '`posIn`' is the raw name without this implicit OID appendage. As discussed above, the text after the colon is the timed type of each of the values in the memory. `64` is the memory's frequency in Hz. The text '`tfl(0)`' indicates that the most recent value that emerges on a memory execution (every 64th of a second) has a time stamp 2 cycles after the time of the memory execution (in this case also at 64Hz).

Line 6 defines the output memory (or box-FIFO memory), and follows the same scheme. The names of memories that are exposed as the instance interface (the Pre-HBCL *instance signature*) are enclosed by the curly braces following the keywords '`observe`' and '`manifest`' for inputs and outputs respectively. We see that the input memory `posIn` 'observes' the environment (at line 9) and `negOut` stipulates that a value be manifested in the environment if the program specification is satisfied (at line 13).

Finally, the definition of a harmonic box is signalled by the keyword '`hbox`' at line 16. In this case, it has a frequency of `64` Hz, as indicated by the number in square brackets. The `hbox` '`bitNeg`' is defined to observe one memory, `posIn`, and output values to one other memory, `negOut`. If we had more than one harmonic box binding, or more than one untimed box language, or more than one expression language, we would need concrete syntax here to indicate which language or binding was being used, so that the appropriate parser and compiler or verifier component could be invoked. Given that for each of these bindings and languages, we have at present only one exemplar, this syntactic clutter is unjustified and we default to discussing the box language that we do possess.

Definition: **Box language**

A computational structure within a coordination language that specifies the functional relationship of a set of inputs to outputs.

The `hbox` definition contains a program in the bit field expression language. There must be at least one function called `main` (at line 18), according to the rules of the binding, and this function must have an argument with a structure that matches the inputs it refers to, and an output that matches the outputs it refers to. Both of these arguments are records. The identifiers that the record uses to map values correspond to variable identifiers which in this case are syntactically equal to the OID identifier of the coordination language memories. The values corresponding to each memory are tuples of values of the memory's underlying plain type. The size of this tuple is determined by the relationship of the frequency of the memory to the frequency of the timed type.[5] In

[5]With the present box language, the box execution frequency must be the same as the memory execution frequency, to comply with the restriction to fixed size types in the underlying type system.

this case both are 64Hz, so the tuple contains one value on each execution of the box. This is why there is apparently an extra set of brackets around the raw data tuple in the type signature of the function. Inside this set of brackets is the raw type of the input and ouput. We use an explicit representation of the type here, which can nonetheless be unified with the declared `uDatTripleProt` type. The actual definition of the function follows the `:=` symbol, and consists of an expression. The expression here is the constructor of an appropriate output value, where values are obtained by an application of the '`not`' function to a pattern that dereferences a component of the input value. The patterns consist of a list of record member identifiers or tuple positional parameters as they descend through the structure of the input type. The built-in '`not`' function can be seen to be applied to each of the input Booleans to produce an output triple at line 20.

3.5.2 Two parallel boxes

We now consider a parallel composition of a one input I/O negation box and a parity calculator.

Figure 3.3 shows the parity box in an identical conformation to the negator box of Figure 3.2. The labels have been changed to refer to parity boxes. `posIn` has become `dataIn`, while `negOut` has become `parityOut`. We therefore proceed straight to present the HBCL code for the parity scenario.

Listing 3.3: A parity box library

```
llib parity {

  type uDatParity ((bool, bool, bool), bool); -- final boolean is parity bit

  oid type uDatParity;

  htype tDatParity : uDatParity [128];

  linst parityInst {

    mem(fb) datIn : commonDat.tDatTriple [64, tfl(0)];
    mem(bf) parityOut : tDatParity [64, ttl(2)];

    observe {
      dataIn;
    }

    manifest {
      parityOut;
    }

    hbox parCalc : datIn -> parityOut [64]
    {
      main : { datIn : (commonDat.tDatTriple) }
       -> { parOut : (tDatParity, tDatParity) } :=
```

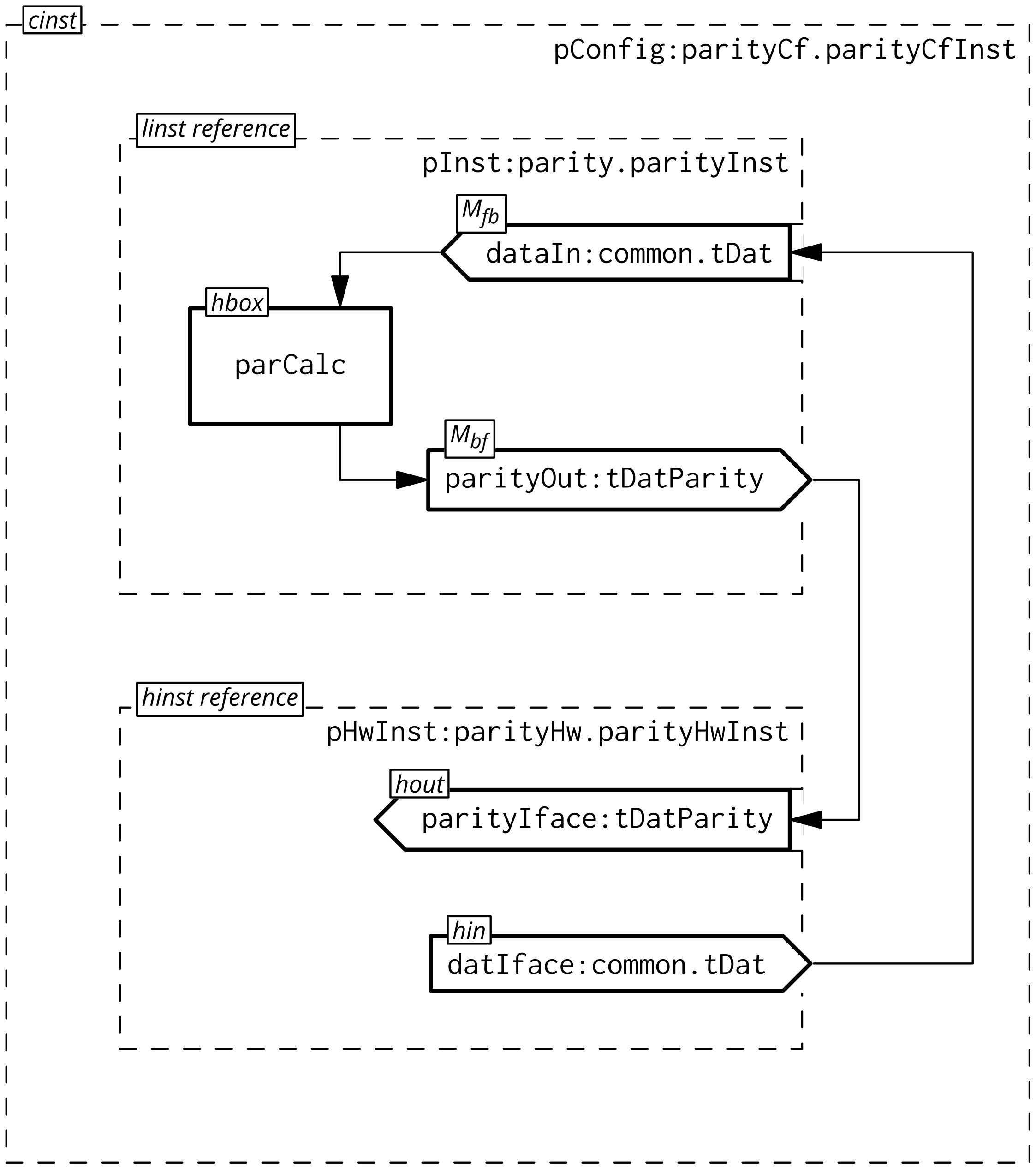


Figure 3.3: A parity box scenario

```
26              { parOut = ((datIn.0, xor datIn.0.0 (xor datIn.0.1 datIn.0.2)),
27                          (datIn.0, xor datIn.0.0 (xor datIn.0.1 datIn.0.2)))
                                };
28      }
29
30    }
31
32  }
```

In comparison with the negator example, the only structural difference to notice is that the data types specific to the parity example are declared at the parity library scope. If they were declared inside the instance they would not be accessible to code instantiating the parity library, and the output would have an unknown type (this would result in an inconsistent HBCL program). This data type also has double the frequency of its memory. The consequence of this is that the output must occur twice in the the output tuple for the '`parOut`' field of the record. Accordingly, the function '`main`' now produces a pair of values, at lines 26 and 27. Each element of the pair is identical. Each of these elements is a pair, containing the original input value and a parity bit. The parity bit is calculated by applying the built-in exclusive OR function `xor` to two of the bits in the input triple, and again applying the OR operation to the result of this and the remaining input bit.

We now give a parallel composition with a negator box, shown in Figure 3.4. It can be seen, if we disregard the nesting, that the structure is exactly the same as two non-communicating negator and parity implementations instantiated side by side. The `linst` is not given in a library context. This means that, in the case of a simulated/interpreted HBCL execution, this is a command to the interpreter to instantiate the instance there and then; in the case of a non-simulated instantiation in reality, it is a statement that the instance exists, and starts to accrue a real history (which may consist of null data if the input memories are not yet bound to some real inputs). The existence of the trace is therefore fully defined, regardless of whether an implementation successfully realizes the program by causing a history of output manifestations consistent with this trace.

Listing 3.4: A parallel composition

```
linst parityPar {

  linst negInst : negator.negatorInst;
  linst parityInst : parity.parityInst;

  observe {
    negInst.tDatIn as tDatInNeg;
    negInst.tDatOut as tDatOutNeg;
  }

  manifest {
```

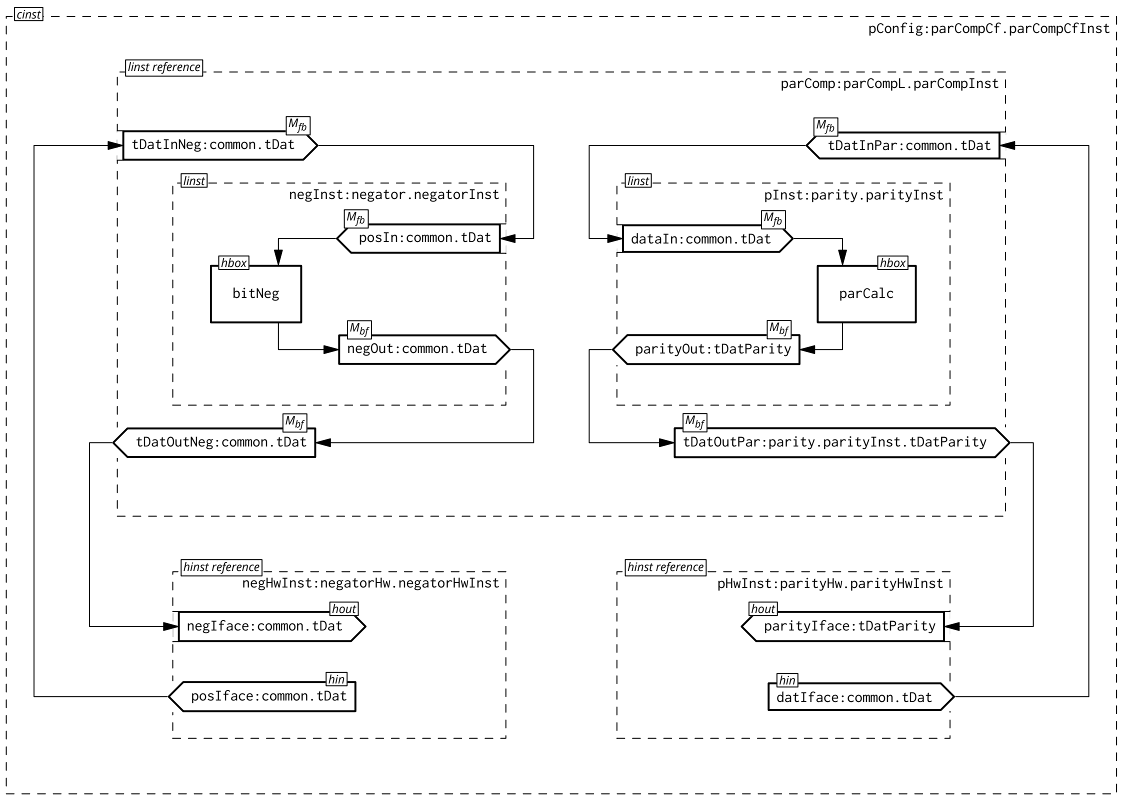


Figure 3.4: A parallel composition

```
    parityInst.dataIn as tDatInPar;
    parityInst.parityOut as tDatOutPar;
  }

}
```

3.5.3 One box with feedback

To illustrate a simple example, we develop a basic rolling checksum. The structure of this example can be seen in Figure 3.5. This time, the logic underlying instance `cInst` has two inputs and two outputs. It is nested inside another instance called `cInstFIFO`, which connects one of these outputs to one of the inputs, so that the new enclosing instance has only one input and output. These are mapped to hardware in the normal way.

First, we define a checksum instance in a library without connecting up the FIFO that achieves the feedback. We do this because we want to reuse this instance in a later checksum pipeline example where we pass the feedback through the negator box.

Listing 3.5: A checksum box library

```
lib checksum {

  inst checksumInst {

    mem(fb) datIn : tDatTriple[64, tfl(0)];
    mem(fb) checkIn : tDatTriple[64, tfl(0)];
    mem(bf) checkOut : tDatTriple[64, ttl(4)];
    mem(bf) checkOutCopy : tDatTriple[64, ttl(4)];

    observe {
      dataIn;
      checkIn;
    }

    manifest {
      checkOut;
      checkOutCopy;
    }

    hbox checkBox : (datIn, checkIn) -> (checkOut, checkOutCopy) [1 / 64]
    {
      main : { datIn : (uDatTripleProt); checkIn : (uDatTripleProt) }
       -> { checkOut : (uDatTripleProt); checkOutCopy : (uDatTripleProt) } :=
       { checkOut = ((xor datIn.0.0 checkIn.0.1),
                     (xor datIn.0.1 checkIn.0.2),
                     (xor datIn.0.2 checkIn.0.0));
         checkOutCopy = ((xor datIn.0.0 checkIn.0.1),
                         (xor datIn.0.1 checkIn.0.2),
                         (xor datIn.0.2 checkIn.0.0))
       };
```

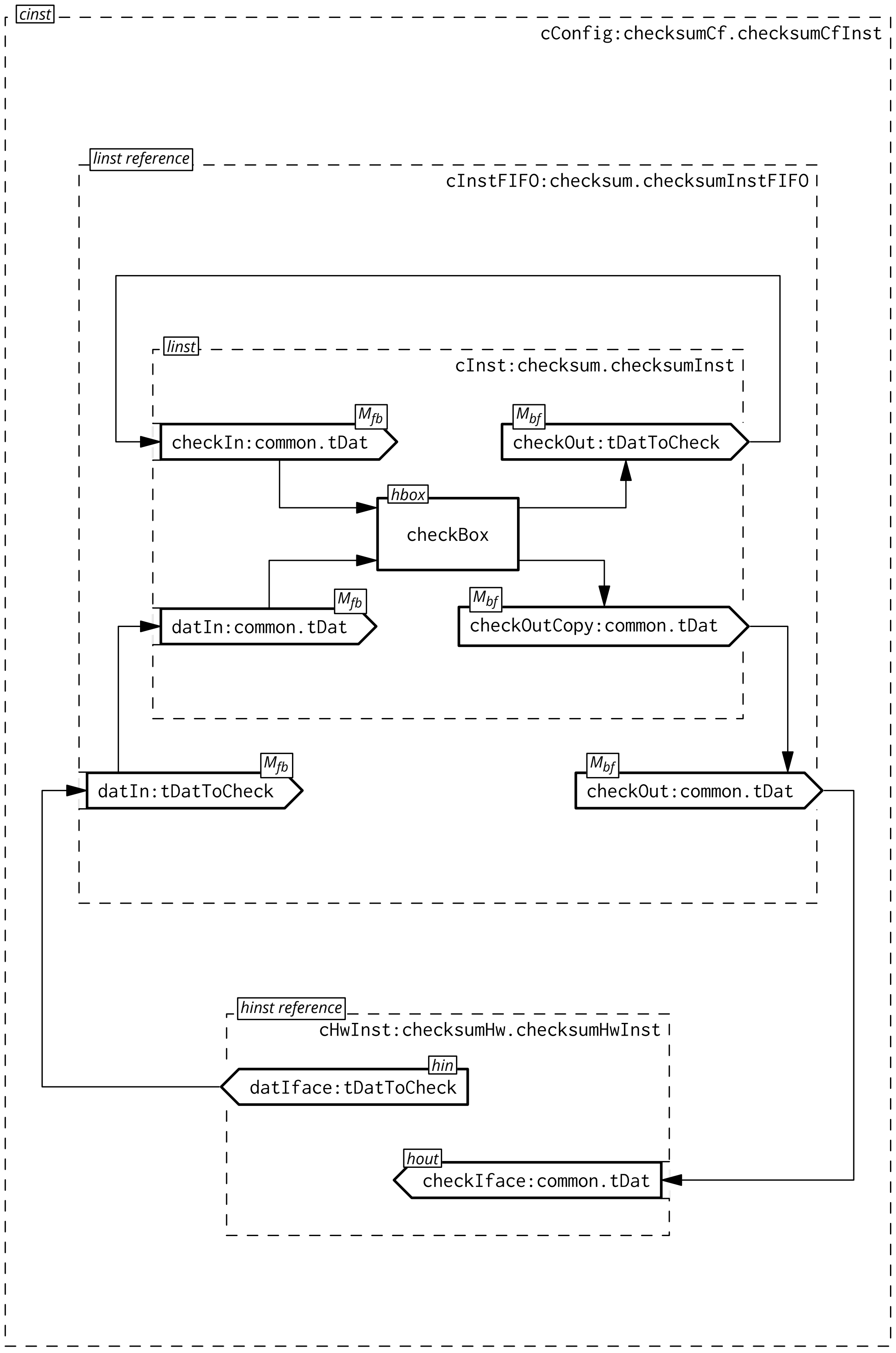


Figure 3.5: A checksum configuration

```
31
32     }
33
34   }
35
36 }
```

Of particular interest (at line 20) is that, reflecting Figure 3.5, the hbox has a signature of *two* input and output memories. We reuse the Boolean triple data type from `commonDat`. This time, the function `main` computes a checksum (lines 24 to 26) using the `xor` function. This is a triple that is assigned in the output tuple to the identifier `checkOut`. At lines 27 to 29, the logic is repeated for the identical `checkOutCopy`. We now instantiate this logical instance, introducing a feedback FIFO:

Listing 3.6: A fully specified single box checksum

```
linst checkTest {

  linst checkInst : checksum.checksumInst;

  observe {
    datIn as checkInst.datIn;
  }

  manifest {
    checkInst.checkOutCopy as datOut;
  }

  fifo checkInst.CheckOut to checkInst.CheckIn;

}
```

This example introduces some new syntax. First, the colon in '`linst checkInst : checksum.checksumInst`' indicates that the instance is being instantiated from a library, in this case the instance `checksumInst` in library `checksum`. If the `checkTest` instance had been specified in a library scope, this instantiation would have been deferred until `checkTest` itself had been instantiated, but given that `checkTest` is instantiated directly in the anonymous global instance scope, it is instantiated immediately. Now that we have a nested instance, the specification of the nested observations and manifestations changes slightly to include the keyword '`as`'. This is syntactic glue between the instance signature of the enclosing instance and the nested instance. '`datIn as checkInst.datIn`' means that the handle `datIn` can be used when using a `checkTest` instance to refer to the `datIn` input memory of the nested instance `checkInst`. The manifested output memories reverse the position of the local and nested identifiers so that the way the statement reads is closer to an equivalent description in natural English. Also, we see the first

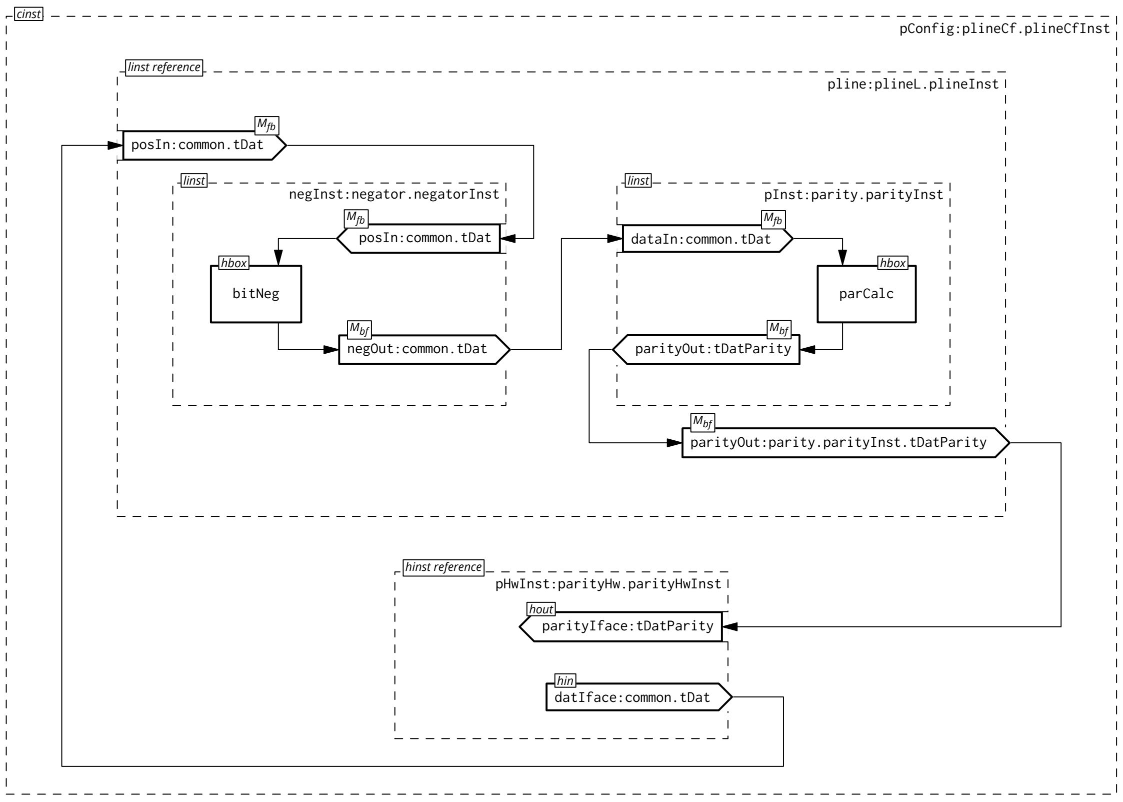


Figure 3.6: A pipeline scenario

example of the keyword '`fifo`', denoting the specification of an eponymous FIFO.

3.5.4 Two boxes in a pipeline

The next example involves two boxes in a pipeline. The structure can be seen in Figure 3.6. As was the case in Figure 3.5, the nesting and connection of a FIFO in the enclosing instance `pLine` has exposed only one input and output to be mapped to hardware I/O.

Listing 3.7: Pipelined boxes

```
linst pLine {

  inst negInst : negator.negatorInst;
  inst pInst : parity.parityInst;

  observe {
    negInst.posIn as posIn;
  }

  manifest {
    pInst.parityOut as parityOut;
  }

  fifo negInst.negOut to pInst.dataIn;

}
```

First in the pipeline is a negator box; it is followed by a parity box. There is a FIFO in between them. This arrangement instantiates instances of each from the relevant libraries and does so at global instance scope. This program computes the same parity function as the plain parity instance, but does so with an inverted input.

3.5.5 Two boxes in a pipeline with feedback from second to first

The final example is a rolling checksum with negation. It can be seen in Figure 3.7. There are no new structures here: the novelty is in the less restricted form of directed cyclic graph described by the communication flows. There are now two FIFOs in the program. Only one input and one output are left and need to be mapped to hardware. The effect of the program is the same as the plain checksum box, except that it computes a different checksum as a result of the negation instance interposed in the checksum feedback.

Listing 3.8: Negated checksum

```
linst checkTestNeg {

  linst checkInst : checksum.checksumInst;
  linst negInst : negator.negatorInst;

  observe {
    datIn as checkInst.datIn;
  }

  manifest {
    checkInst.checkOutCopy as datOut;
  }

  fifo checkInst.CheckOut to negInst.posIn;
  fifo negInst.negOut to checkInst.datIn;

}
```

3.6 More involved examples

In chapter 6, we develop a set of progressive examples which culminate in a replicated multiplier in section 6.8.5; we preview its illustration in Figure 3.8, as it is convenient to illustrate what we mean by an *interpretation function*.

The replication example of Figure 3.8 shows three parallel replicas of the same code:

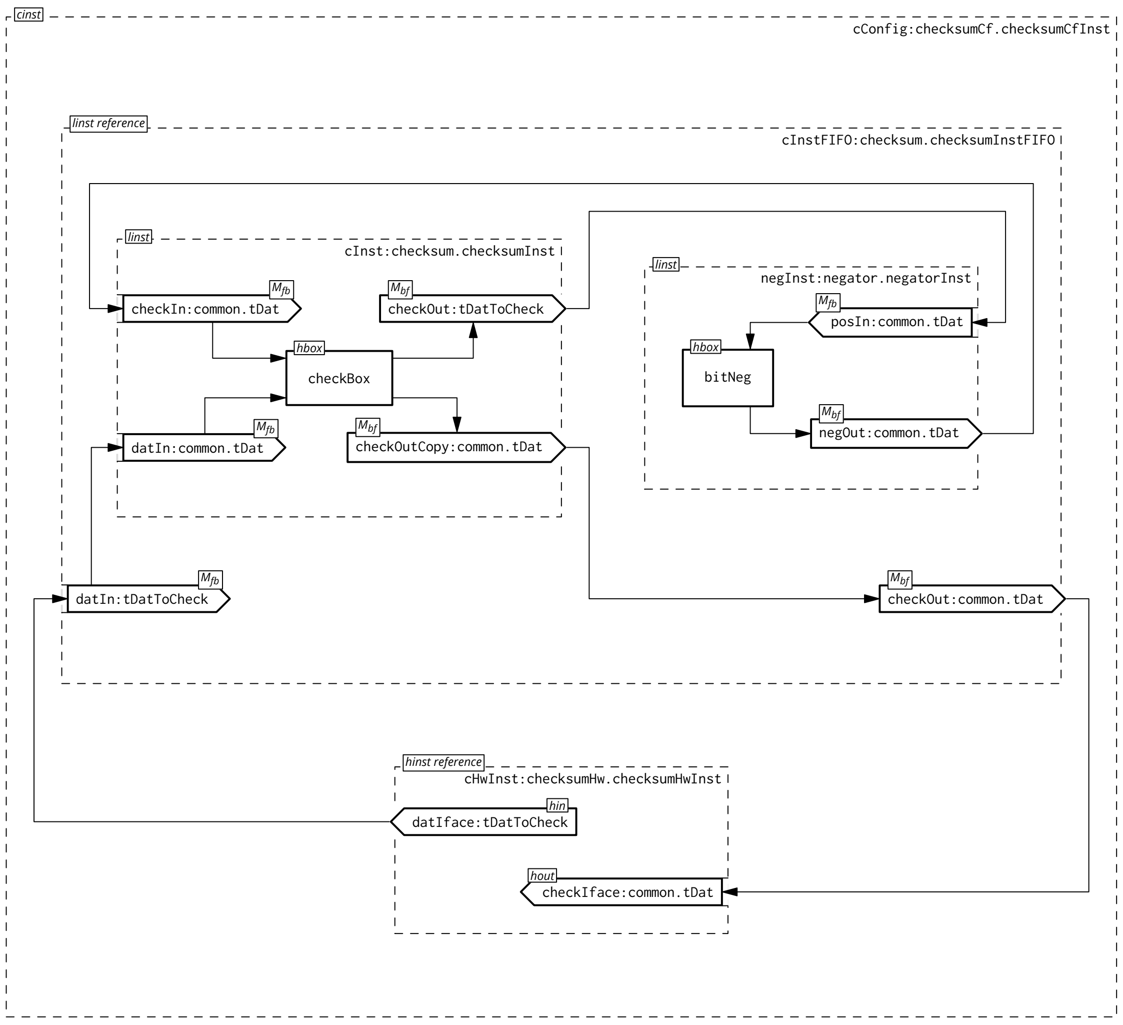


Figure 3.7: A checksum calculated with negation

given the fanout and voter logic, we can construct a bisimulation predicate that witnesses the non-replicated equivalent: the interpretation function in question is a two-out-of-three voting function. Unlike asynchronous consensus systems, each replica never needs to 'know' what the state of the other replicas is. If a replication transformation is applied by drawing a boundary around some processes and replicating what is inside, bounded by fanouts and voters, then if another transformation is drawn so as to overlap these fanouts and voters, a restoring effect is carried forward through a data flow without ever any single replica knowing whether it is correct. This is the old idea of restoring organs, introduced by von Neumann in 1956 [189], and we should be careful to point out that the apparently Byzantine faults that it can tolerate have nothing to do with the axiomatization of Byzantine processes communicating by asynchronous message passing of Lamport *et al.* [127]. The Byzantine Generals' Problem is about resistance to a potentially duplicitous general inconsistently disseminating his decisions to other unreliable generals who may (mis)communicate between themselves; the synchronous, von Neumann-style, model is about a diffuse aggregate truth as comprehended across a clique of processes that do not directly communicate with each other being passed on in the same diffuse, probabalistic form through another discrete set of replica generals. The will of these generals is a synthetic property following from the definition of which generals are in the ruling clique; it is not and need not be accurately known by every general involved, and might be better described as a 'Byzantine polyarchy'.

3.7 HBCL timing principles and properties

Hbcl's timing model is axiomatic. We will now explain in more detail what we mean by this, and why it is a useful approach. An hbcl specification exists in a single coordinate system in which events only occur at rational-number instants in time. This is not physically plausible, as it neglects dispersion, jitter and wandering clocks. Dispersion occurs because of instabilities and uncertainties in measuring the time of flight of a particular signal. Jitter occurs because of the instability in any physical clock (usually characterized by the Allan variance), thermodynamic effects in information transit times and quantum effects in measurement. The further that two supposedly synchronized clocks or clock signals are from one another in space, the further they may wander from each other due to accumulated random walk effects in clock propagation. The key to transforming an idealized hbcl specification that captures the logical requirements of the application developer into a realistic transformation is to use two ideas that implement the same concept in discrete and continuous domains respectively. The first idea is to use bisimulation and morhpisms to transform hbcl programs into more precisely timed hbcl programs. The second, related, idea is to use homeomorphisms when projecting the actual, continuous, timing properties of a physical system onto the rational-number

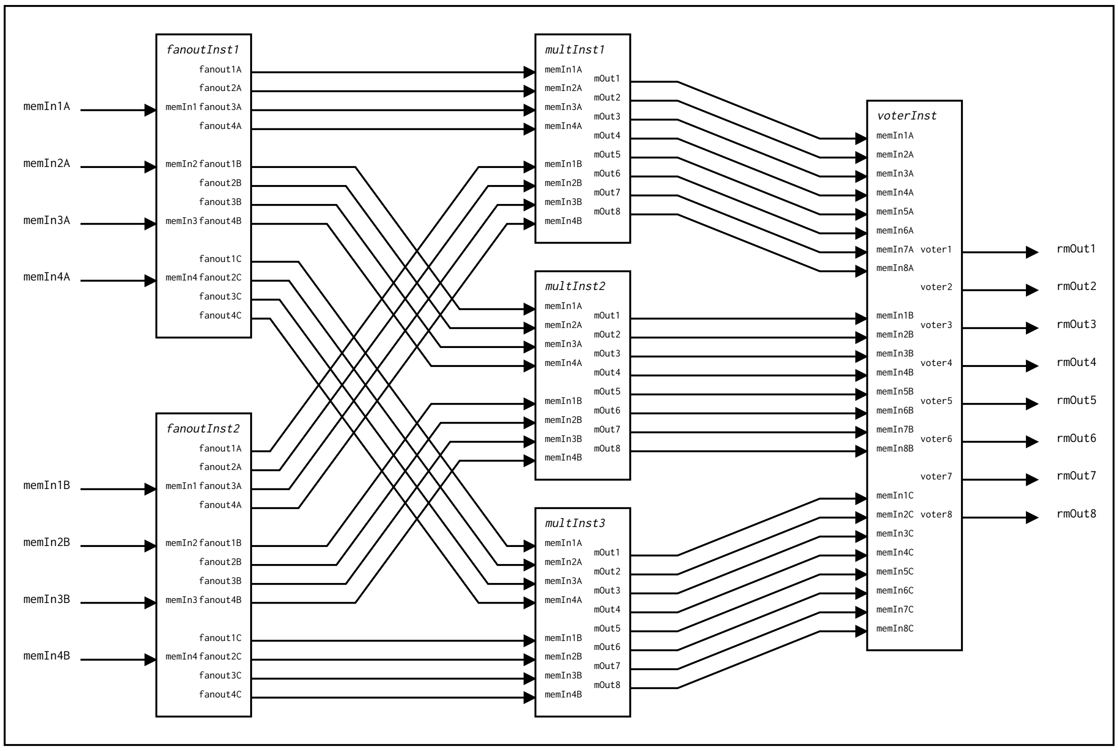


Figure 3.8: Structure of replicated multiplier

grid. The point can be made by considering a ‘rubber sheet’ graph of memories (discrete) against time (rational or continuous) that can be arbitrarily manipulated within certain boundary conditions. Transformations from one HBCL program to another allow only discrete manipulations of points of interest on this sheet, while hardware transformations would allow continuous manipulations on a real-numbered backdrop.

To illustrate this point, we adapt Lamport’s graphical approach, specifically, that found in Figure 3 of his seminal paper on time, clocks, and the ordering of events [123]. Figure 3.9 shows the negator box of section 3.5.1 using this kind of notation. The two-dimensional surface does not show two dimensions of space-time (3 + 1 dimensions), but instead memory identifier-time (1 + 1 dimensions): there are therefore no hidden or collapsed axes. The distinction between physical space and identifier space is important because the apparently un-physical features of the diagram, such as instantaneous communication, do not violate the speed-of-light restriction on the rate of information transfer in *metres* per second: the horizontal axis of the diagram is concerned only with units of discrete-links-between-abstract-objects per second, which lacks dimensions of physical distance.

In this kind of diagram, vertical red lines show FIFO-box memories. Such a memory is a point in one-dimensional identifier space, smeared into a line along the time

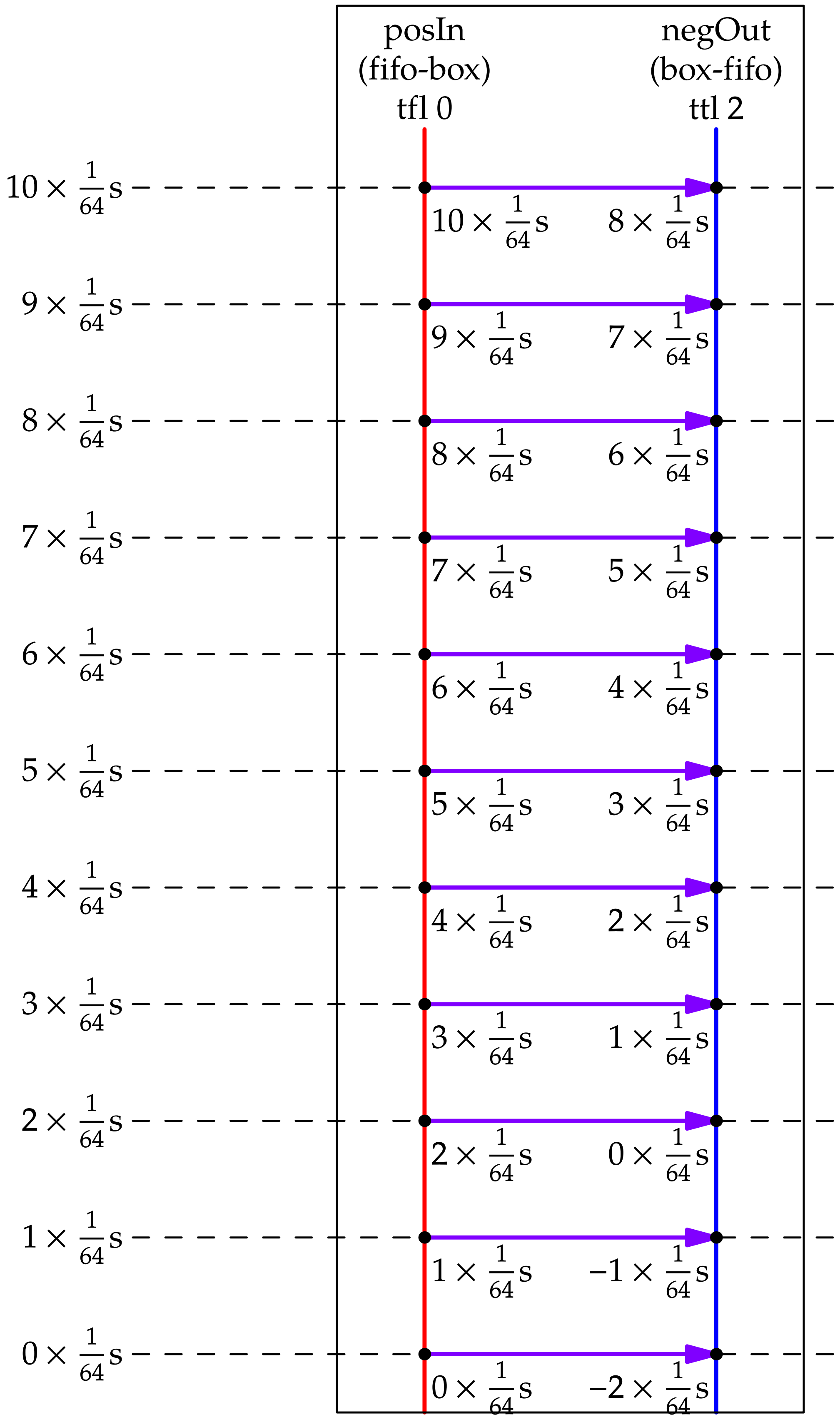


Figure 3.9: Lamport-like timing diagram for negator box

axis, analogously to a world line in the physical terminology of relativity. Blue lines do the same for box-FIFO memories. Purple lines show causative relationships stipulated between two memories, which are specified by instances of HBCL box languages. Green lines show the causative relationships between two ends of a FIFO. In order for HBCL programs to be deterministic and realizable when mapped to hardware, there are restrictions on the gradients of these lines. Each FIFO has a direction that is implicit, given that a FIFO only ever transmits information from a box-FIFO memory to a FIFO-box memory. For extra clarity, an arrow head is included in each execution of a FIFO shown on the diagram. If the arrow is considered as a vector with identifier-space and time components, then the time component must be *greater than or equal to zero* for a box arrow, and *strictly greater than* for a FIFO. Given that a box cannot be connected directly to another box, the combination of these two conditions ensures that any particular box output cannot affect its own corresponding input, which would create a contradiction, violate our model of causality, and render the language non-deterministic.

To return to the rubber sheet model, in refining the timing within HBCL or finding a physical realization (projection onto spacetime), we oberve that if the background fabric of reality is inelastic, then we must impose some boundary conditions on permissible distortions to prevent causation lines running backwards or at superluminal speed in the (at least, relatively) inelastic space onto which the rubber sheet is projected. Finally, we note that the causation lines of boxes and FIFOs cannot be 'drawn' onto the sheet if we consider continuous deformations of a rubber sheet, because even a small number of boxes and FIFOs projected onto two dimensions in this way would be a cat's cradle, the topology of which would change as the lines were crossed and uncrossed as various sections of memories were deflected up or down. If we did draw the lines on the sheet, we would still be able to consider the relationship between two geometries as causation-preserving homeomorphisms, but not continuous deformations.

The presence of FIFOs can be seen in Figure 3.10. The FIFO vectors obey the requirement of having a strictly positive time component. We also see another feature in this example: the `parityOut` memory has a timed type that has twice the frequency of the memory. The dot for the `parityOut` memory, which ticks at *instants* in time, has been stretched into a lozenge, spreading the box execution arrow that connects to it into a triangle. This represents a kind of 'dispersion' that occurs as a result of the mismatch of frequencies. In general, we can stretch either kind of memory like this, turning each arrow into a trapezium with parallel sides along the time axis. Each non-parallel side must obey the same gradient conditions as when we were considering this shape as collapsed into a line, with the additional requirement that a line drawn between the top left and bottom right corners must also obey the same condition.

In Figure 3.11 we see the result of a rubber-sheet manipulation in which the disparity of timed type frequency, compared to that of the memory, is elided. This transformation

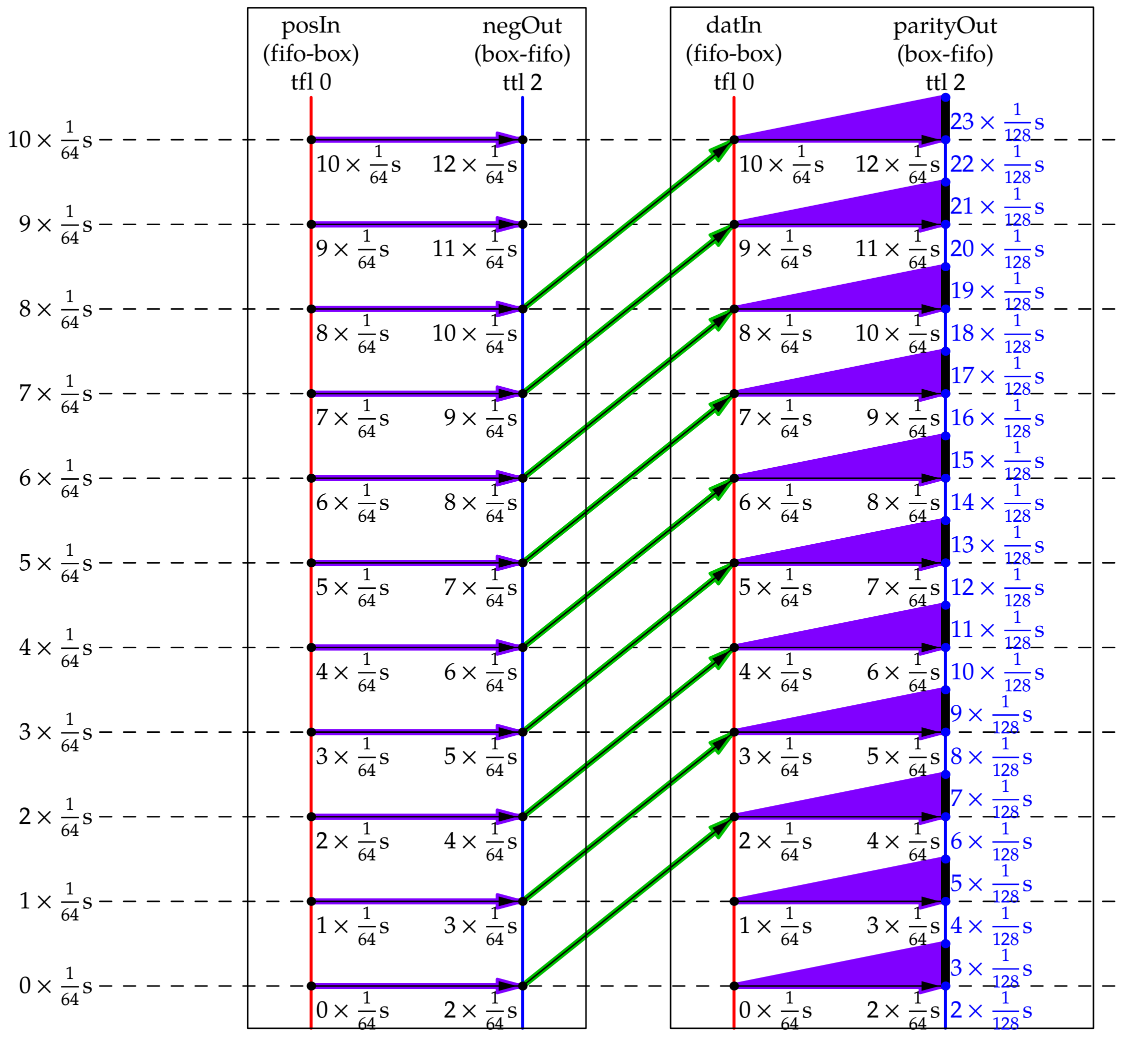


Figure 3.10: Timing diagram for pipeline example

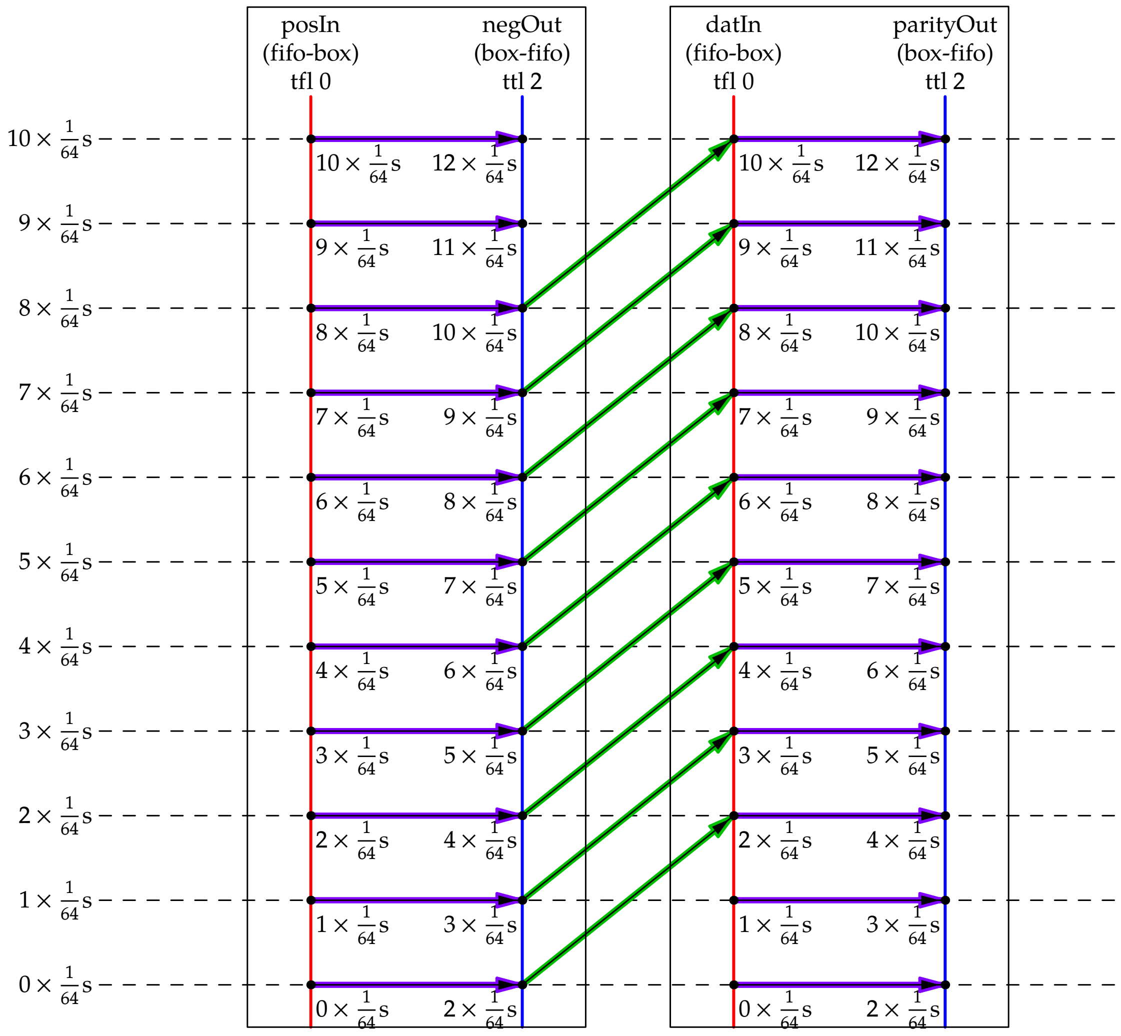


Figure 3.11: Timing diagram for pipeline example with compressed timed type timeline relative to memory timeline

has not violated the causality rules or changed the topology of the spatial relationship of memories; the memories represented by the black lozenges have been compressed back into circles of nominally infinitessimally small size.

Universal compositionality of HBCL programs follows axiomatically from this style of specification when components are linked with FIFOS. Since each end of the FIFO is defined relative to the same clock (the same temporal coordinate system) that is instantiated with the *language*, not with the language *instance*, then the resulting FIFO has a fixed and deterministic length. Whether or not two *implementations* are composable or not is quite a different matter, because dispersion, jitter and wandering clocks must be taken into account in this case. We now elucidate how this might occur in the context of our simple pipeline example. The power of this approach is not that it excuses us from engineering acceptable temporal tolerances, but that transformations from one HBCL program to another, such as that illustrated in Figure 3.8, can take place without considering physical constraints. The overall correctness of the implementation of an HBCL program, transformed via other HBCL programs, and eventually rendered into a physical implementation, is provided by the *transitivity* of the transformations, of which only the final one needs to be concerned with details such as dispersion, jitter and wandering clocks. The boundary conditions might be expressed algebraically using interval arithmetic. In appendix A.4.1, we think of this transformation design space as an 'entropy sandwich'.

One potential criticism that can be levelled at the 'rubber sheet' model is that it is very restrictive of allowable clock divergence if universal composability is to work, as the necessary guard interval to cope with the 'blurring' of the vibrating rubber sheet within some probabalistic bounds is small; it gets smaller with every higher-frequency component that is added, and is also constrained by the length of the shortest FIFO. This is a powerful motivator for programming a model into hierarchical units, as is allowed by full HBCL. The lowest common multiple (henceforth LCM) of the memories that are visible from *outside* an HBCL instance is independent of all of the potentially higher frequencies inside the instance. This is compatible with our intuition about most computers, where clocks become more and more tightly synchronized with decreasing spatial dimensions, with the most tightly controlled clocking environment being that of a central processing unit.

We can imagine attaching a semi-rigid boundary to the rubber sheet enclosing an HBCL instance, whose internal clock tolerance (or rubber sheet deformation tolerance) might have a higher LCM frequency, and thus a lower permissible deviation in the time domain. If that instance communicates with its neighbours over longer FIFOs, and in some ensemble with a lower LCM, then the tolerance between instances may be larger. We might intuit that larger clock deviations due to vibrations in the measured time occur between clocks with a larger spatial separation. Or, to think of it another way, the instantaneous displacement uncertainties of clock divergence between two spatially separated clocks would probabalistically correlate with the spatial separation.[6] Carrying out these implementations is outside the scope of the current thesis, but we consider them because these matters have an effect on the design of HBCL, and especially its hierarchical nested structure. We stress that nothing in this 'rubber sheet' way of thinking about implementations requires that physics is like that: it is just a way of axiomatizing unavoidable measurement uncertainties.[7]

[6]Thought of in terms of wavelengths, these quantities start to look something like a superposition of waves.

[7]To look at the other side of the coin, if real, physical spacetime (as opposed to a construction that helps us to reason about the uncertainties of metrology) *could physically* be consistently axiomatized to have this kind of minutely non-deterministic vibrating geometry on quantum scales, then we suppose it might conceivably make an interesting way of smoothing some aspects of the impedance mismatch between quantum mechanics and general relativity — but such thoughts about theoretical physics are far outside our present

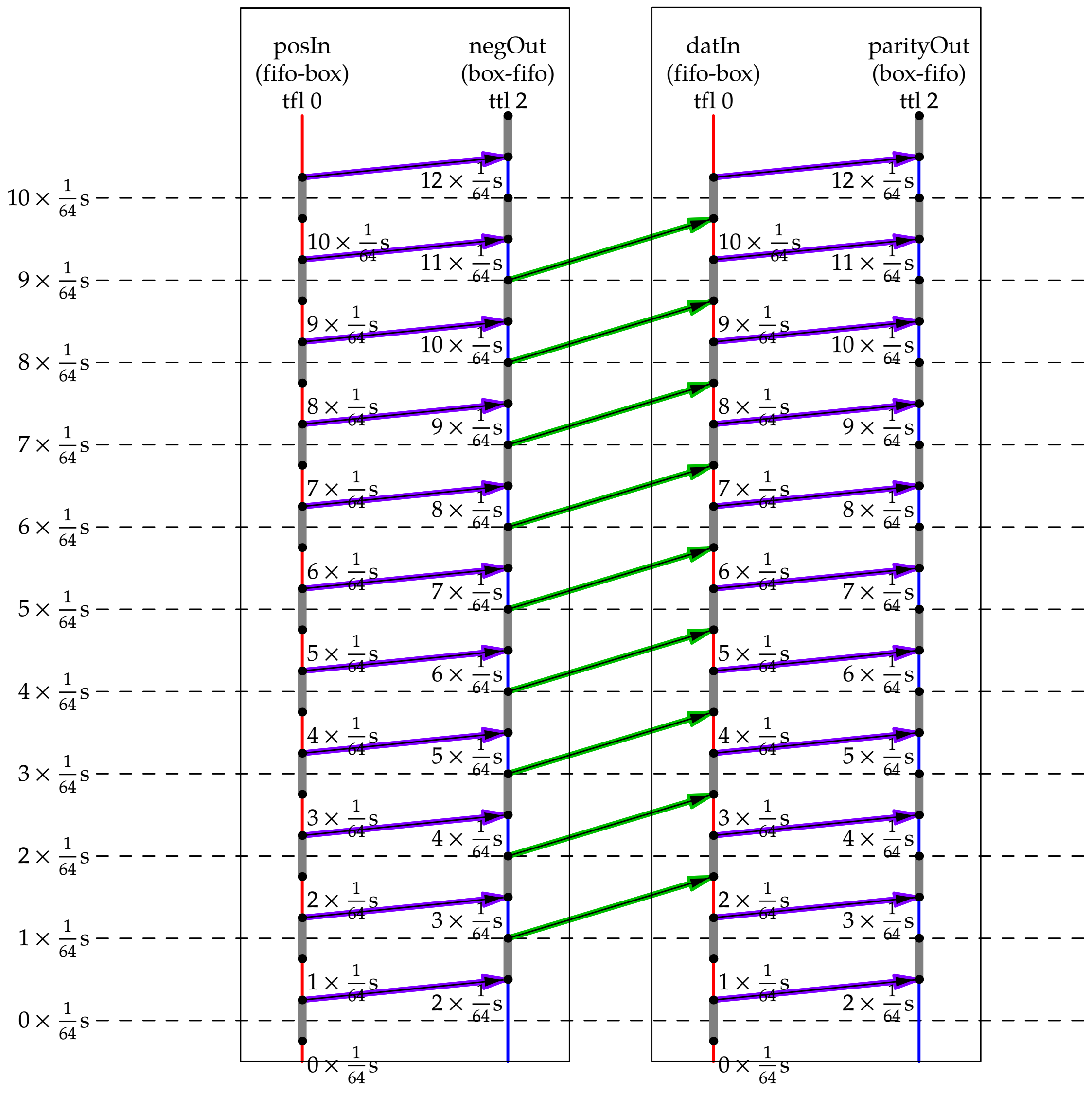


Figure 3.12: Timing diagram for pipeline example with read-write memory phases dissociated

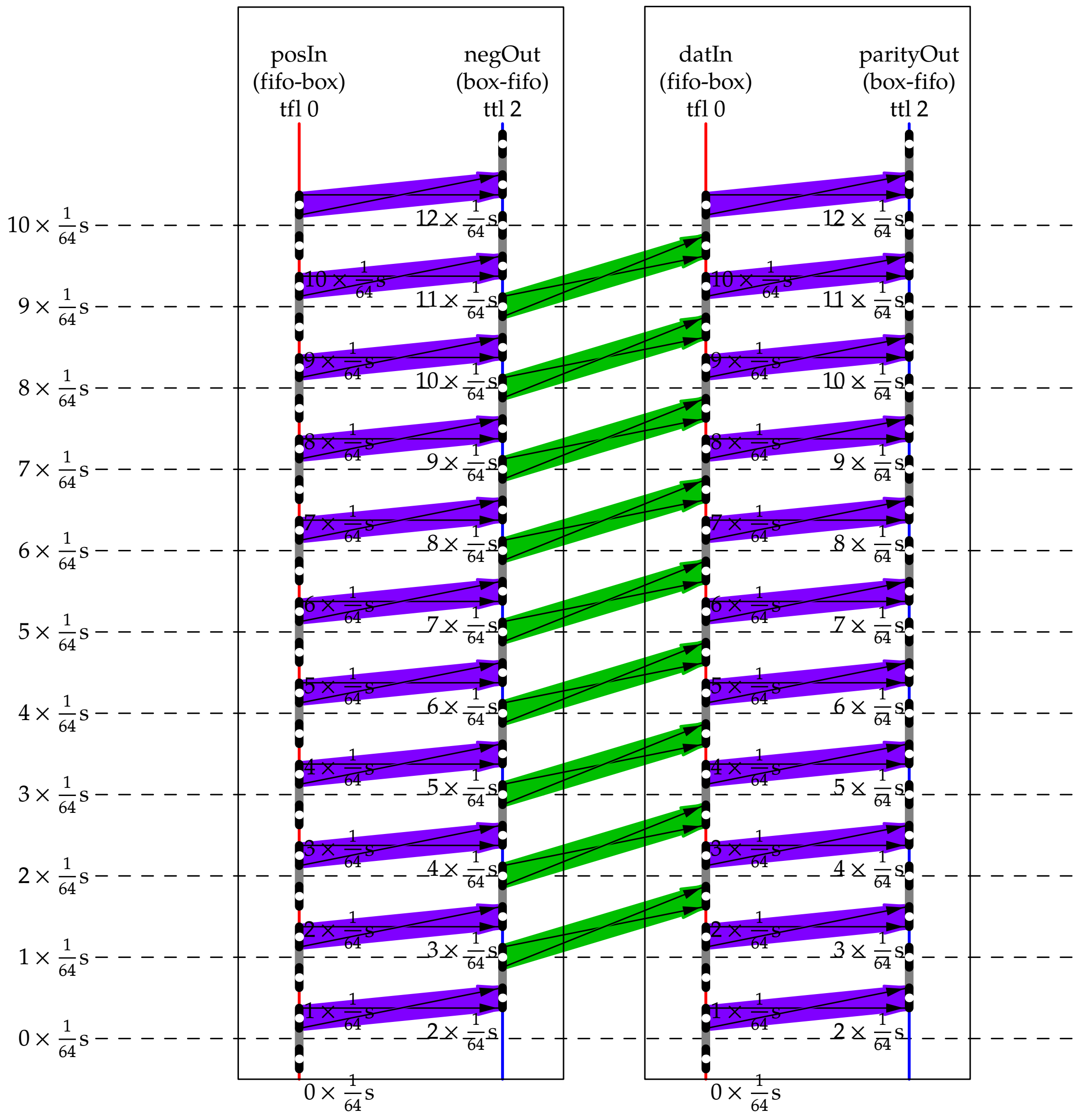


Figure 3.13: Timing diagram for pipeline example with intervals

In Figure 3.13 we see the permissible smearing due to vibrations of the rubber-sheet coordinate system. The lozenge has been further divided up, with the writing and reading phases of a memory separated by the grey guard interval, and the instants for reading and writing then being spread into more realistic intervals. The gradient top-left to bottom-right diagonal arrow in each purple and green causation link shows the maximum 'speed' of the reading at one end affecting the writing at the other; the bottom left to top-right arrow shows the minimum speed.

Definition: **Multi-manifest**

Of a quantity, to be represented, by definition, by some function over spatio-temporally dispersed measurable quantities.

Before ending this section, we make a final remark on the engineering of clock trees. Clocks in HBCL are multi-manifest. We define the clocks we are interested in and some function over them (such as the average number of ticks, weighted by the stability of the clock, since some particular reference event), without ever knowing the quantity to which this function evaluates; it is instead known to a particular degree of certainty within defined bounds. The *actual* measurements of some subset of these clocks can then be transmitted in a clock tree to replicated slave clocks, which are singly not authoritative, but which are collectively defined to be a particular slave clock, whose unknowable average is *knowably* inside some bound of the master clock ensemble, up to a given level of certainty. The master clock indexes the top level coordinate system, but the clock trees are distributed spatially in the same way that HBCL instances are distributed spatially. As a result, local ensembles relying on some local slave clock ensemble may have some large error compared to the clocking of an arbitrary HBCL instance with an implementation more distantly localized in space. However, these distant instances are unlikely to need as high a level of coordination with respect to *each other* as compared to a closely coupled local system, so the inevitably longer FIFOs can be allowed to have larger differences between their minimum and maximum tolerated transmission speeds, allowing greater relative blurring of temporal coordinate systems. The statistics of engineering precisely what amounts to an acceptable clock tree divergence and guard interval is an implementation matter that would need to be addressed, were HBCL to be developed into a system of practical tools.

3.8 Summary

We have now seen how HBCL programs are constructed. We have met the basic coordination features and seen how elements can be composed by the use of nesting. We

subject, so let us leave them there.

have explained the timing properties of HBCL and the issues that must be navigated in implementations, presenting diagrams that can be regarded as a scheme for graphical proofs. However, we have not yet seen a formal account of what is and is not a valid HBCL program. Neither have we said how the coordination state evolves from one discrete time slice to the next. We address these issues in the next chapter, presenting formal syntax, before elaborating on the static and dynamic semantics and discussing formal properties of the language.

Chapter 4

The Harmonic Box Coordination Language II: Formal syntax and semantics

The practice of tuning organs by equal temperament is, in my humble opinion, most erroneous.

Samuel Sebastian Wesley — *Musical Standard, 1863*

This chapter concerns the formal specification of the concrete and abstract syntax of HBCL, its semantic domain and dynamic semantics. To aid familiarity, we have tried, in developing the syntax, to adhere to conventions in the semantics of Standard ML [144] and Hume [96] wherever possible.

4.1 Structure of the formalization of HBCL

We have adopted the following methodology in developing the semantics:

1. Specify the underlying datatypes of the static semantic domain.

2. Wrap these datatypes in predicates in order to arrive at a strongly typed static semantic domain in which only meaningful and executable subsets of the underlying data structures can be constructed. These predicates are parametric in the input and output types (or more generally, bounds) of the program. Leave the predicates undefined until the operational semantics have been developed.

3. Develop the dynamic semantic domain. Stub out predicates confining it to sane states for the program being executed. These predicates take a static semantic object as a parameter and generate a σ-type giving the type of the domain. In the

case of the coordination language, this is the coordination state object. In the case of a functional expression language, the domain of this is the type of expressions.[1]

4. Provide the operational semantics over the semantic domain.

5. Strengthen the operational semantic rules using predicates on the dynamic semantic domain states before and after the application of each rule, where inhabitants of the strong semantic domain parametrize those predicates.

6. Refine all predicates referred to above until the point at which those specifying the semantics always describe an injective relation, so that reductions of the operational semantics always find the single inhabitant of the relevant σ-type. This σ-type is the result of the evaluation of a program as specified by the overal injective relation of the top-level semantic rule: the input parametrizes the σ-type, restricting its inhabitant to the allowed output.

7. Develop static semantics that specify how a compiler/verifier may construct the static semantic object.

This approach is a compromise between the ease of developing operational semantics, and the importance of finding a type-theoretical meaning of programs. A type-theoretical meaning is desirable in order to enable extensions to the work in which the validity of refinement steps are shown by finding witnesses to the trace inhabiting the type, or proving morphisms between different expressions of the same semantics that have also been reduced to a type. We do not follow this methodology to the end, since it requires an amount of work out of scale with the current enterprise, but we follow it as far as the specification of the operational semantics.

The semantics are therefore strongly specified using predicate subtyping, *modulo* completion of all empty predicates and admitted lemmas. Many of these predicates are dependent on parameters, giving rise to a form of dependent types. The operational semantics are given in an evaluation style, and are closely related to the construction of a reference intepreter that proceeds using the reduction rules of a host logic based on a typed λ-calculus.

The approach followed by the *static* semantics is to specify completely all possible abstract syntax trees as either producing a semantic object that satisfies the semantic rules, or an invalid semantic object. If a valid static semantic object is produced, it has its dependent type fixed by the type of its run-time argument and return type.

The dynamic semantics bind an argument to this static semantic object and guaran-

[1]For expression execution, we use a big-step semantic model, using the types of a host logic to define the type of expressions, and outsourcing our reduction semantics through a direct appeal to the host logic's reduction rules. We thus avoid dealing directly with Scott-Strachey domain theory, or asking what reduction actually *is* in denotational or other form.

tee to evaluate a result: this implies that the static semantic object must carry constructive proof of termination. In a pure functional language, an interpreter in this paradigm is a function that takes this static semantic object to produce an evaluation function of the correct type. The static semantic object maps directly to the concept of a piece of executable code, where the interpreter function, prior to the application of the static semantic object and input argument, embodies the semantics of the machine that executes the code. The functions in the reference interpreter developed for this thesis essentially form an executable specification.

There is, of course, an infinitely large set of static semantic objects, semantic formulations and data encodings that populate a category of isomorphisms with respect to state evolutions of executions for different encodings. The more equivalent semantics that are given in heterogeneous styles, the more inhabitants of this isomorphic category can be found, and the more confidence we can have that the language being formalized is well understood. For the purposes of this thesis, however, we confine ourselves to concrete reduction rules.

A subset of isomorphisms of the type just discussed corresponds to stages of compilation through generally injective transformations between intermediate stages. This differentiates compilers from transformations into equivalent axiomatizations of the language, which should be bijective. This approach is similar to that used in the context of imperative languages by the Compcert project [128,129]. Compcert and its sister projects use Coq to build certified compilers, whose input languages are generally subsets of C. Consequently, a quick route to a certified compiler could be produced for the present language by providing a certified transformation into the existing Compcert formalization of the semantics of one of these C dialects.

It is worth stressing that the semantics specify that only valid program objects carry proofs of correctness. A verified compiler (which we leave to further work) must construct only objects in this type, but the existence of an object in this type does not necessarily mean that the compiler can construct it. In other words, there is always a possibility of there being undetectable errors in the static semantics. However, this is much less worrisome than the prospect of constructing invalid program objects that are then used as if they were valid. If it were felt necessary to prove that any compilation function did not produce false negatives, it would require a more informative and strongly specified inconsistent object. These are not priorities, so they are not pursued further here.

4.2 Abstract syntax

We now present an abstract syntax for HBCL. The notation is conventional; it is discussed in the context of a simple propositional calculus example in appendix B.

4.2.1 Sets and primitive categories from the meta-logic

$$\textit{intconst} \in \left\{n \in \mathbb{Z} : \text{-}2^{64} \leq n \leq 2^{64}\text{-}1\right\} \tag{4.1}$$

The set *intconst* is the set of signed 2's complement 64-bit integers.

$$\textit{intconstpos} \in \textit{intconst} \cap \{n \in \mathbb{Z} : n \geq 0\} \tag{4.2}$$

The set *intconstpos* consists of those integers that are members of *intconst* and are greater or equal to zero.

$$\textit{boolconst} \in \{\top, \bot\} \tag{4.3}$$

The set *boolconst* is the set of the constants 'true' and 'false'.

$$\textit{romanletter} \in \{\texttt{a},\ldots,\texttt{z}\} \cup \{\texttt{A},\ldots,\texttt{Z}\} \tag{4.4}$$

The set *romanletter* is the set of upper and lower case roman letters, of which there are 52.

$$\textit{arabicnumeral} \in \{\texttt{0},\ldots,\texttt{9}\} \tag{4.5}$$

The set *arabicnumeral* is the set of decimal figures zero to nine.

$$\textit{idprepend} \in \textit{romanletter} \cup \{\texttt{\_}\} \tag{4.6}$$

The set *idprepend* is the set of characters that may begin an identifier, being the set of roman letters and the underscore character.

$$\textit{idappend} \in \textit{romanletter} \cup \textit{arabicnumeral} \cup \{'\} \tag{4.7}$$

The set *idappend* is the set of characters that may be present in any other position in an identifier name: the roman letters, the arabic numerals and the 'prime' symbol.

4.2.2 Abstract syntax construction rules

4.2.2.1 Abstract syntax common to coordination and expression languages

$$\begin{array}{lcl} \textit{type} & ::= & \texttt{typeBasetype}\ \textit{basetype} \\ & | & \texttt{typeTupletype}\ \textit{types} \\ & | & \texttt{typeRecordtype}\ \textit{assoctypes} \\ & | & \texttt{typeTypeid}\ \textit{typeid} \end{array} \tag{4.8}$$

The category *type* is the disjoint union of four kinds of type: a base case type, a tuple of types, an associative array (record) of types, and an invocation of a type previously defined by name.

$$basetype \quad ::= \quad \texttt{basetypeBool} \tag{4.9}$$

The category *basetype* is a singleton category containing only one base type. It could be extended to include numerical types to enrich the language.

$$\begin{aligned} types \quad ::= \quad & \texttt{typesInd}\ types \quad type \\ | \quad & \texttt{typesBase} \end{aligned} \tag{4.10}$$

The *types* category is a tuple of types, built up inductively from an empty base case, with each recursive construction using `typesInd` adding a new *type*.

$$\begin{aligned} assoctypes \quad ::= \quad & \texttt{assoctypesInd}\ assoctypes \quad vardeclprim \\ | \quad & \texttt{typesBase} \end{aligned} \tag{4.11}$$

The inductive structure of *assoctypes* works in the same way as *types*, except that this time each new type is associated with a variable identifier, using the *vardeclprim* category.

$$vardeclprim \quad ::= \quad \texttt{vardeclprim}\ varid \quad type \tag{4.12}$$

vardeclprim is the Cartesian product of a variable identifier and a type. It is used in declaring variable types and defining the permissible contents of particular record types.

$$varid \quad ::= \quad \texttt{varidId}\ id \tag{4.13}$$

varid is a singleton category wrapping the plain identifier type, retained in case we were later to want to introduce a further syntactic rule for this kind of identifier (for example, we might limit the initial letter to lower case characters).

$$\begin{aligned} id \quad ::= \quad & \texttt{idHead}\ idprepend \\ | \quad & \texttt{idTail}\ idprepend \quad idappends \end{aligned} \tag{4.14}$$

An identifier is either a single character (one of the set of *idprepend*), or such a character combined with a list of characters formed from the permitted successor characters defined by *idappend*.

$$\begin{aligned} idappends \quad ::= \quad & \texttt{idappends}\ idappends \quad idappend \\ | \quad & \texttt{idappendsIdappend}\ idappend \end{aligned} \tag{4.15}$$

The category *idappends* is a list of *idappend* characters, formed in the same way that we defined *types*. The abstract syntax notation does not faciliate higher-order lists, so we repeat this design pattern a number of times.

$$typeid \quad ::= \quad \texttt{typeidId}\ id \tag{4.16}$$

A *typeid* is a specialization of the *id* type, in the same way that *varid* was.

$$typedef \quad ::= \quad \texttt{typedef}\ typeid \quad type \tag{4.17}$$

The category *typedef* associates a type identifier with a plain type.

$$utypedef \quad ::= \quad \texttt{utype}\ typeid \quad typeid \tag{4.18}$$

The category *utypedef* declares an untimed OID type. The first *typeid* is the new type identifier for this new OID type, while the second is the plain type out of which the new type is constructed.

$$htypedef \quad ::= \quad \texttt{htype}\ typeid \quad typeid \quad freq \tag{4.19}$$

The category *htypedef* follows the same pattern as *utypedef*, except this time a new harmonic type is formed from an untimed OID type. A frequency is added to the definition.

$$freq \quad ::= \quad \texttt{freq}\ intconstpos \quad intconstpos \tag{4.20}$$

An object of the category *freq* consists of two positive integers. The first is the numerator of a rational-numbered frequency; the second is the denominator. The implicit units are SI seconds.

4.2.2.2 Coordination language top level

$$\begin{aligned} linst \quad ::= \quad & \texttt{linst}\ id \quad linstdecls \\ | \ & \texttt{linstlib}\ id \quad linstref \end{aligned} \tag{4.21}$$

A logical instance is either an association between a new logical instance identifier and a set of instance declarations (an in-place instance made with the `linst` constructor), or it is an association of a new logical instance identifier with a logical instance from a library (constructed using the `linstlib` constructor).

$$\begin{aligned} linstdecls \quad ::= \quad & \texttt{linstdeclsInd}\ linstdecl \quad linstdeclsInd \\ | \ & \texttt{linstdeclBase} \end{aligned} \tag{4.22}$$

The category *linstdecls* is a list of objects that may appear in a logical instance definition.

$$\begin{aligned} linstdecl \quad ::= \quad & \texttt{linstdeclLinst}\ linst \\ | \ & \texttt{linstdeclLlib}\ llib \\ | \ & \texttt{linstdeclHbox}\ boxid \quad memfbids \quad membfids \quad freq \quad uprogram \\ | \ & \texttt{linstdeclMemfb}\ memfbid \quad htyperef \quad ttfl \\ | \ & \texttt{linstdeclMembf}\ membfid \quad htyperef \quad ttfl \\ | \ & \texttt{linstdeclType}\ typeany \\ | \ & \texttt{linstdeclObs}\ memfbidrefs \\ | \ & \texttt{linstdeclManif}\ membfidrefs \\ | \ & \texttt{linstdeclFIFO}\ membfidref \quad memfbidref \end{aligned} \tag{4.23}$$

A logical instance's declarations and definitions may be one of nine kinds, as given by the disjoint union defined by *linstdecl*. The `linstdeclLinst` constructor signifies a nested logical instance, while `linstdeclLlib` introduces a library that can only be referenced at the scope of this logical instance or a contained scope. The `linstdeclHbox` constructor builds a harmonic box definition: its name is contained in *boxid*, its (local) input and output memories are referred to by the lists *memfbids* and *membfids* respectively, the frequency of the harmonic box is described by a *freq* object, and the code executed on each invocation of the box by *uprogram*. `linstdeclMemfb` defines the input memories of the box, associating an appropriate identifier with a harmonic datatype and a time to or from live. `linstdeclMemfb` does the same for output memories. `linstdeclType` introduces a type, which can be a plain, untimed or harmomic OID type. `linstdeclObs` refers to those input memories that observe the environment of the logical instance. This can include memories from enclosed instances that are directed to observe the environment without interference from the present logical instance. `linstdeclManif` does the same as `linstdeclObs` but applies to output memories to be manifested in the environment. `linstdeclFIFO` defines a FIFO connecting an output memory to an input memory. It may link local memories as well as the exposed interfaces of nested memories.

$$llib \quad ::= \quad \texttt{llib}\ id \quad llibdecls \tag{4.24}$$

An *llib* associates an identifier with a logical library definition, which is a list of logical library definitions.

$$\begin{aligned} llibdecls \quad ::= &\quad \texttt{libdeclsInd}\ libdecl \quad libdeclsInd \\ \mid &\quad \texttt{libdeclBase} \end{aligned} \tag{4.25}$$

The category *llibdecls* is a list of the types of definitions permitted in a logical library.

$$\begin{aligned} libdecl \quad ::= &\quad \texttt{libdeclLinst}\ linst \\ \mid &\quad \texttt{libdeclLlib}\ llib \end{aligned} \tag{4.26}$$

The *libdecl* category is a disjoint union of the two types of definitions permitted in a logical library: further nested libraries or logical instances. These structures are mutually inductive.

$$boxid \quad ::= \quad \texttt{boxId}\ id \tag{4.27}$$

A *boxid* is the identifier type of harmonic boxes. This singleton is retained for the same reason given for *varid* and *typeid*.

$$\begin{aligned} memfbids \quad ::= &\quad \texttt{memfbidsInd}\ memfbid \quad memfbids \\ \mid &\quad \texttt{memfbidsBase} \end{aligned} \tag{4.28}$$

The category *memfbids* is a list of input memories.

$$memfbid \quad ::= \quad \texttt{memfbId}\ id \tag{4.29}$$

The type of input identifiers is given by *memfbid*. It is another singleton type wrapping the base identifier definition.

$$\begin{array}{lcl} membfids & ::= & \texttt{membfidsInd}\ membfid\quad membfids \\ & | & \texttt{membfidsBase} \end{array} \tag{4.30}$$

The category *membfids* is a list of output memories.

$$membfid \quad ::= \quad \texttt{membfId}\ id \tag{4.31}$$

The type of output identifiers is given by *membfid*.

$$htyperef \quad ::= \quad \texttt{htyperef}\ libdecls \quad typeid \tag{4.32}$$

An *htyperef* is a harmonic type (*typeid*) qualified by a (possibly empty) library reference (*libdecls*).

$$ttfl \quad ::= \quad \texttt{ttfl}\ intconst \tag{4.33}$$

A time from live is given by a positive *intconst*, while a time to live is given by a negative one. It becomes a member of the *ttfl* category when the `ttfl` constructor is applied to this integer.

$$\begin{array}{lcl} typeany & ::= & \texttt{typeanyType}\ typedef \\ & | & \texttt{typeanyUtype}\ utypedef \\ & | & \texttt{typeanyHtype}\ htypedef \end{array} \tag{4.34}$$

The disjoint union of any of the kinds of HBCL data type is given by *typeany*

$$\begin{array}{lcl} memfbidrefs & ::= & \texttt{memfbidrefsInd}\ memfbidref\quad memfbidrefs \\ & | & \texttt{memfbidrefsBase} \end{array} \tag{4.35}$$

The list *memfbidrefs* holds references to input memories that may have been declared in a nested instance.

$$\begin{array}{lcl} memfbidref & ::= & \texttt{memfbidrefLocal}\ memfbid \\ & | & \texttt{memfbidrefLInst}\ id\quad memfbid \end{array} \tag{4.36}$$

The category *memfbidref* is the disjoint union of local and nested input memory references. The constructor `memfbidrefLInst` signifies the nested version, qualified by a reference to a nested instance.

$$\begin{array}{lcl} membfidrefs & ::= & \texttt{membfidrefsInd}\ membfidref\quad membfidrefs \\ & | & \texttt{membfidrefsBase} \end{array} \tag{4.37}$$

The category *membfidrefs* is the same as *memfbidrefs*, except that it is for output memories rather than inputs.

$$\begin{array}{lcl} membfidref & ::= & \texttt{membfidrefLocal}\ membfid \\ & | & \texttt{membfidrefLInst}\ id\quad membfid \end{array} \tag{4.38}$$

The category *membfidref* is the same as *memfbidref*, except that it is for output memories rather than inputs.

$$linstref \quad ::= \quad \texttt{linstref}\ llibrefcomp \quad id \tag{4.39}$$

The category *linstref* qualifies a logical instance reference (*id*) with a logical library. The logical library may be inside another library, hence the composite list of library identifiers of *llibrefcomp*

$$\begin{aligned} llibrefcomp \quad ::= \quad & \texttt{llibrefcompInd}\ id \quad llibrefcomp \\ | \ & \texttt{llibrefcompBase} \end{aligned} \tag{4.40}$$

The category *llibrefcomp* is a composite library identifier, used for dereferencing nested logical libraries.

4.2.2.3 Expression (untimed box) language top level

$$uprogram \quad ::= \quad \texttt{uProgDecls}\ udecls \tag{4.41}$$

The *uprogram* category is the syntactic container of instances of the expression language.

$$\begin{aligned} udecls \quad ::= \quad & \texttt{udeclsInd}\ udecls \quad udecl \\ | \ & \texttt{udeclsBase} \end{aligned} \tag{4.42}$$

The category *udecls* is a list of the permitted declarations and definitions of the expression language.

$$\begin{aligned} udecl \quad ::= \quad & \texttt{declVardecl}\ vardecl \\ | \ & \texttt{declVardef}\ vardef \end{aligned} \tag{4.43}$$

The *udecl* category is the disjoint union of variable declarations and definitions. For simplicity, we do not allow type declarations here.

$$\begin{aligned} vardecl \quad ::= \quad & \texttt{vardeclVar}\ vardeclprim \\ | \ & \texttt{vardeclFun}\ fundeclprim \end{aligned} \tag{4.44}$$

A variable declaration can either be a primitive (data-valued) variable or a function-valued variable.

$$fundeclprim \quad ::= \quad \texttt{fundeclprim}\ varid \quad type \quad type \tag{4.45}$$

The category *fundeclprim* associates a variable identifier with the argument and return type of the function respectively.

$$\begin{aligned} vardef \quad ::= \quad & \texttt{vardefVar}\ vardeclprim \quad expr \\ | \ & \texttt{vardefFun}\ fundeclprim \quad varid \quad expr \end{aligned} \tag{4.46}$$

A *vardef* object is the disjoint union of a plain variable or function definition. In the first case, under the constructor `vardefVar`, an expression is associated with the variable name and type of the *vardeclprim*. In the second case, under the constructor `vardefFun`, an expression is associated with the variable name, argument and return type of the *fundeclprim*. The expression may refer to an argument *varid*, bound on invocation to the argument type given by *fundeclprim*.

$$\begin{array}{lcl} \textit{expr} & ::= & \texttt{exprPatt}\ \textit{patt} \\ & | & \texttt{exprConstr}\ \textit{constr} \\ & | & \texttt{exprFunapp}\ \textit{varid}\quad \textit{expr} \end{array} \tag{4.47}$$

An *expr* is the disjoint union of a pattern expression, a constructor expression or an application expression. In the last case, the *varid* is the name of the function to which the evaluated inner *expr* is to be applied.

$$\textit{patt} \ ::= \ \texttt{pattIndex}\ \textit{varid}\quad \textit{datresolvelist} \tag{4.48}$$

A pattern consists of a *varid*, which identifies a piece of data in the environment, and a list of record members and positional parameters (a *datresolvelist*), which disassembles a composite type of mutually nested records and tuples.

$$\begin{array}{lcl} \textit{datresolvelist} & ::= & \texttt{datResolveInd}\ \textit{datresolve}\quad \textit{datresolvelist} \\ & | & \texttt{datResolveBase} \end{array} \tag{4.49}$$

The *datresolvelist* is a list of *datresolve* components.

$$\begin{array}{lcl} \textit{datresolve} & ::= & \texttt{datResolvePos}\ \textit{intconst} \\ & | & \texttt{datResolveVarid}\ \textit{varid} \end{array} \tag{4.50}$$

Each *datresolve* is either a positional parameter of a tuple (constructed under `datResolvePos`) or a named record field constructed under `datResolveVarid`.

$$\begin{array}{lcl} \textit{constr} & ::= & \texttt{constrBase}\ \textit{boolconst} \\ & | & \texttt{constrTup}\ \textit{exprtup} \\ & | & \texttt{constrRec}\ \textit{exprrec} \end{array} \tag{4.51}$$

The category *constr* is the disjoint union of a Boolean (base type) constant, a tuple formed from a tuple of expressions of the *exprtup* category, and a record formed from a record of expressions of the *exprrec* category.

$$\begin{array}{lcl} \textit{exprtup} & ::= & \texttt{exprtupInd}\ \textit{exprtup}\quad \textit{expr} \\ & | & \texttt{exprtupBase} \end{array} \tag{4.52}$$

The *exprtup* category specifies a list of *expr* expressions.

$$\begin{array}{lcl} \textit{exprrec} & ::= & \texttt{exprrecInd}\ \textit{exprrec}\quad \textit{varid}\quad \textit{expr} \\ & | & \texttt{exprrecBase} \end{array} \tag{4.53}$$

The *exprrec* category specifies a list of expressions, each *expr* being indexed by a *varid* label. The order of the record components in the list is not material.

4.3 Approach to the semantics

In approaching the static semantics, we use predicate subtyping constructions in generating a static environment. This ensures that a compiler, when traversing the abstract syntax tree, produces either:

- An object which, when interpreted by the dynamic semantics, only gives terminating and correct results; or
- The inconsistent environment object. Obtaining this as a result of the compilation process indicates that a static semantic rule was violated. Obtaining it as a result of looking up an identifier in an environment indicates that the environment in question did not contain a mapping for that identifier. This is the expected result when making a new *declaration* (the environment should not already contain a contradictory one), but it indicates an error when the identifier is expected to have been previously *defined*.

  These inconsistent environment elements of semantic objects are loosely equivalent to exceptions in an operational formulation, or something analogous to 'bottom' in a denotational style. In the present formulation, this 'bottom' value in each semantic variable is a constant: if one wanted explicitly to specify a functional specification of compilation error reporting, one would want to convert it into a category that could carry useful information about semantic errors.

The static semantics therefore give, in effect, a direct specification of a strongly specified (although very inefficient) compiler. The inefficiency is not important, since one can always later optimize and prove equivalence. The dynamic semantics provide a similar route to an interpreter.

Proposition types in the premises of a rule (which are the result of testing equality after pattern matching on the return result of a function) are available to be recast as proof terms in the conclusion implicand. Usually proof should be straightforward in these cases, because we arrange that the inductive structure of predicates matches the recursive structure of concrete functions.

4.4 Semantic domain

The semantic domain serves two purposes. The first is to provide concrete data types over which the static and dynamic semantics can operate. The second is to contain the semantics themselves through the use of predicates and dependent types. For example, the environment of *declared* variables is a parameter to the environment of *defined* variables. This is expressed in the predicate specification of the definition environment by giving the declaration environment as an argument to the this polyadic predicate. This predicate is used to form a parametrized σ-type, in which the concrete structure of the definition environment is absorbed into the σ-type, while the declaration environment (and others) form parameters. This ensures that it is impossible *by construction* to build nonsensical concrete structures, and this in turn allows primitive recursive operational rules to be constructed (as given in appendix C). These operational rules are existential proof that the predicates describe computable functions.

In giving the example of variable declaration and definition environments, we have just described how the semantic domain can be made to contain the static semantics. The same can be done for the dynamic semantics. Here, in understanding the dependent typing, it is useful to think of the physical idea of an action. A physical 'action' is some relation that can predict the future of a system given its state at a particular instant. If, instead of considering the position and velocities of solid bodies (in a Newtonian system, say), we substitute the global state variable of a computational system, then we use the system state at time t as a dependent *type* argument to the state at time $t + \delta t$. In HBCL, that δt is the lowest common multiple of all harmonic components in an instance, that is, the frequency of that instance. If the system evolves deterministically without external input, then this type of coordination state at time $t + \delta t$ has only one inhabitant; otherwise, the particular value that is taken at $t + \delta t$ is a function of an input state that is external to the HBCL instance in question. These concepts generalize into a succession of coordination states, each one being added to a trace object that has dependent type at the time t, and which generates a new trace object with the coordination object for $t + \delta t$ forming its final dependent argument. An input *stream* supplies enough extra information at each step to make each transition deterministic. The semantics of state transition are contained in the predicate that must be supplied in the *constructor* of the trace object, which axiomatizes the relation between the old and new coordination states. The type of the trace is dependent in the type of the stream, and this is what gives rise to our notion of typed reality, in which, for each distinct input stream, there is one inhabitant of the trace.

The following sections detail the types used in the semantic domain. In the discussion that follows we will frequently refer to 'types' in different contexts, sometimes in the same sentence. This is unavoidable when using types to define type systems: the meaning of each use can be inferred from the context; to keep referring to 'object' and 'subject' types or similar terminology would necessitate an increase in verbosity vitiating any hoped-for increase in clarity.

4.4.1 Typography

We adopt the following conventions for semantic objects and environments (the latter is a subset of the former containing mappings of identifiers to other semantic objects):

1. The same letter is used for every font to represent types, variables and enclosing categories of the same definition (subscripts render the letter to which they are attached a different letter for these purposes). Where a bare syntactic category is used in a definition of the semantic domain (shown in an italic typeface), it only appears with a quantifier if an instance of that category must be referred to in the same definition. In semantic rules, we achieve the same effect by picking convenient letters for syntactic categories and subscripting them with the name of the category in order to make their type clear.

2. `Monospace` font is used for singleton types.

3. *Italic Roman* font is used for variables that range over consistent types, and for the names of objects from the abstract syntax.

4. **Bold Roman** font is used for variables that range over a consistent type and the opposite inconsistent type.

5. $\mathscr{Script}$ font is used for the type of consistent or inconsistent objects, except $\mathscr{P}$, which is reserved for indicating a power set type.

6. $\mathfrak{Fraktur}$ font is used for the union type of consistent and inconsistent objects corresponding to the same semantic entity.

7. $\mathsf{Sans\ serif}$ font is used for the names of predicates. The subscript 'Prop' indicates that it has the type of logical propositions.

8. $\top$ denotes truth, $\bot$ denotes falsity. Both together indicate that the object under consideration can take either consistent or inconsistent values.

9. $\bigsqcup$ defines a union of types.

10. $\varnothing$ defines or denotes an empty type; $\in$ denotes type membership analogously to the usual set-theoretic usage.

11. The construction $\prod \left| \begin{array}{l} \cdots \\ \cdots \end{array} \right.$ denotes a product type, in which entries higher in the pile of types may be referenced by entries lower in the pile as dependent arguments in parametrized σ-types. For convenience and clarity, the members of these product types are sometimes given names.

12. The construction $\left\{\begin{array}{l} x \in \ldots : \\ \mathsf{X}_{\mathsf{Prop}}(x) \end{array}\right\}$ denotes a predicate sub-type, or σ-type.

13. $\rightarrow$ denotes a total mapping from a domain on the left to a co-domain on the right; $\leftarrow$ denotes a dependent type. Such types are restricted to type-parametrized σ-types that are only allowed to be dependent in the arguments of their defining predicates. This restriction ensures that the semantics do not have to be implemented in a dependently typed logic: one with predicates (and preferably predicate sub-types) will do. Types given to the left of leftwards (type) arrows may be dependent in any number of arguments to the right. The parameters can be inferred from the previous definitions of those types and the meta-variable name inference rules. An exception to this occurs when the dependent type is dependent in two quantities of the same type (whose variables are differentiated by prime symbols), in which case the ambiguity is resolved by a statement in parentheses of which one is meant: the variable on the right of the equality symbol shows which of these variables of the same type is to be assigned to the variable on the left.

14. Instances of dependent types can be explicitly instantiated in their type parameters. This is indicated by giving the base dependent type with the instantiating arguments supplied in parantheses, with the rightmost type parameter of the dependent type being given first.

15. Concrete instances of dependent types are given as fractions, with the variable name given as the 'numerator' (its type can be inferred by looking up the script typeface version of the character), and the 'denominator' providing the dependent type arguments, with the leftmost argument appearing first. Rather than have a stack of 'fractions' for dependent arguments which are themselves dependent, all dependent arguments are flattened into a single denominator, with the dependency of these arguments following the right-to-left rule described above.

16. A full stop (.) indicates dereferencing of a field of a product type. If an explict name for the field is not given in that product type, it can be inferred from the standard meta-variable equivalent of the type by changing the font as described above.

17. The coordination language can accommodate arbitrary expression languages. This means that higher order types become first order entities in the semantics of the coordination language: these higher order types *are* the semantics of a particular expression language that is being used in a particular box in the coordination

language. We use the construction $X_\top$: $\mathscr{X}$ in the meta-variable column of the semantic domain tables to indicate that the script letter is one of these higher types giving the encoding of the semantics of the expression language; the ordinary typeface equivalent is the meta-variable used for a member of this type. The interpretation of the expression language semantics encoding is given by function parameters in other coordination language structures. This becomes clear as the coordination language is presented.

18. The parametrized expression language semantics requires two final pieces of notation. We use τ to denote a higher order type giving an expression language encoding in definitions, and we subscript it with the script letter we use to represent that type as a meta-variable when we need a label to define that meta-variable's type. This sounds confusing, but becomes clear when the coordination language semantic domain table is examined. Similarly, we use κ to stand for a predicate on a higher-order τ type. The π operator is subscripted by a type to show that an instance of a σ-type following it in parentheses is being stripped of its outer predicate, thus yielding the underlying concrete (as subscripted) type of its argument.

Although the typography is intricate, it permits a reasonably conventional presentation whilst allowing the type-theoretical information to be conveyed in compact notation. Also to this end, to keep the semantics as concise as possible, we have adopted further conventions on omitting arguments, either because they can be inferred from dependent types, or because the type and variable have the same label *modulo* the font. We explain these as we encounter them.

4.4.2 Semantic domain common to coordination and expression languages

The coordination language and expression language communicate through a common type system, and this type system is reflected in objects in the semantic domain that are common to both.

4.4.2.1 Simple type environment static semantic object

Meta-variable or constant	**Type name**	**Type definition**
$U_\top, \mathtt{U}_{\top\varnothing}$	$\mathscr{U}_\top$	$\left\{ \begin{array}{l} u \in \prod \left\lvert \begin{array}{l} t \in \mathscr{P}(typeid) : \\ t \to type \end{array} \right. : \\ \mathsf{TEnvWF}_{\mathsf{Prop}}(u) \end{array} \right\}$
$\mathtt{U}_\bot$	$\mathscr{U}_\bot$	$\varnothing$

Meta-variable or constant	Type name	Type definition
$\mathbf{U}_{\top\bot}$	$\mathfrak{U}_{\top\bot}$	$\mathscr{U}_{\top} \bigsqcup \mathscr{U}_{\bot}$

This object is a semantic wrapper to the syntactic pre-type object *type*. This pre-type on its own would not be semantically usable as its meaning is implicitly dependent on what other declarations have been made. In the construction below, we first express a mapping of type identifiers to pre-type objects as the product of the power set of type identifiers and a total function of elements of this power set to pre-types. The explicit use of a power set here ensures that we can recover a set of keys from the mapping, which we could not do if we used a bare partial function. The σ-type we create using this map object specifies a predicate ($\mathsf{TEnvWF}_{\mathsf{Prop}}(u)$) ensuring that references to type identifiers within a type object refer only to types that are in the mapping. There is a meta-variable shown for this whole structure; there is also a constant referring to the empty mapping. The empty mapping is a consistent structure, and it is subscripted as such according to the conventions described in section 4.4.1.

4.4.2.2 Simple type static semantic object

Meta-variable or constant	Type name	Type definition
$T_{\top}$	$\mathscr{T}_{\top} \leftarrow \mathscr{U}_{\top}$	$\left\{ \begin{array}{l} t \in \mathit{type} : \\ \mathsf{tWFInEnv}_{\mathsf{Prop}}(U_{\top}, t) \end{array} \right\}$
$\mathtt{T}_{\bot}$	$\mathscr{T}_{\bot}$	$\varnothing$
$\mathbf{T}_{\top\bot}$	$\mathfrak{T}_{\top\bot} \leftarrow \mathscr{U}_{\top}$	$\mathscr{T}_{\top}(U_{\top}) \bigsqcup \mathscr{T}_{\bot}$

This is the full type object, which is dependent on a pre-type environment. If this object can be formed, it asserts that the pre-type of the σ-type is valid in the environment given, and therefore the corresponding pre-type could be consistently added to the type environment provided that a non-duplicate identifier be chosen. This proposition is specified by $\mathsf{tWFInEnv}_{\mathsf{Prop}}(U_{\top}, t)$.

4.4.2.3 Simple type pre-data static semantic object

Meta-variable or constant	Type name	Type definition
$D_{\text{-}}$	$\mathscr{D}_{\text{-}}$	$\coprod \left\| \begin{array}{l} \mathit{boolconst} \\ \mathscr{D}_{\text{-}tup} \\ \mathscr{D}_{\text{-}rec} \end{array} \right.$

Meta-variable or constant	**Type name**	**Type definition**
$D_{\text{-}tup}$	$\mathscr{D}_{\text{-}tup}$	$\prod_{i=0}^{n} \mathscr{D}_{\text{-}i}$
$D_{\text{-}rec}$	$\mathscr{D}_{\text{-}rec}$	$\prod \left\lvert \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{D}_{\text{-}} \end{array} \right.$

This is the raw data object corresponding to the full type object, lacking the intrinsic typing that is added in the type of the full data object. The negative subscript differentiates it from the full type, indicating that there is something missing from the full specification. The three definitions are mutually inductive. The first is the ordinary type object, ranging over base types, tuples and records. The second and third definitions are for tuples and records respectively. We use the same power-set and total function approach as before in defining the mappings of records.

4.4.2.4 Simple type data static semantic object

Meta-variable or constant	**Type name**	**Type definition**
D	$\mathscr{D} \leftarrow \mathscr{T}_{\top} \leftarrow \mathscr{U}_{\top}$	$\left\{ \begin{array}{l} D_{\text{-}} : \\ \mathsf{DataWF}_{\mathsf{Prop}}\left(U_{\top}, \frac{T_{\top}}{U_{\top}}, D_{\text{-}}\right) \end{array} \right\}$

This is the data object corresponding to the full type object, as qualified by the necessary types and dependent arguments. The σ-type is dependent in both type environment and full type object (which is itself dependent in the type environment). The dependent parameters are implicit in the type definition, since the meaning of their names corresponds with the type signature. According to our typographic conventions, we use the fraction construction to stress that the type object is dependent in the type environment. This is necessary, since in later examples, multiple variables of the same type, differentiated by their 'prime' symbols, will be present in the same rule. The proposition $\mathsf{DataWF}_{\mathsf{Prop}}\left(U_{\top}, \frac{T_{\top}}{U_{\top}}, D_{\text{-}}\right)$, used in constructing the σ-type, asserts that the concrete pre-data object is well formed given its asserted type, which is correct in its type environment by construction, since it is dependent in it.

4.4.2.5 Untimed oid type environment static semantic object

Meta-variable or constant	**Type name**	**Type definition**
$U_{U\top}, \mathrm{U}_{U\top\emptyset}$	$\mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\left\{ \begin{array}{l} u \in \prod \left\| \begin{array}{l} t \in \mathscr{P}\big(typeid\big): \\ t \to \mathscr{T}_{\top}(U_{\top}) \end{array} \right. : \\ \mathsf{UOidEnvWF}_{\mathsf{Prop}}(U_{\top}, u) \end{array} \right\}$
$\mathrm{U}_{U\bot}$	$\mathscr{U}_{U\bot}$	$\emptyset$
$\mathbf{U}_{U\top\bot}$	$\mathfrak{U}_{U\top\bot} \leftarrow \mathscr{U}_{\top}$	$\mathscr{U}_{U\top}(U_{\top}) \bigsqcup U_{U\bot}$

This object is the environment for untimed OID types. The underlying product type is a mapping of type identifiers to complete types dependent in a pre-type environment. Its predicate ($\mathsf{UOidEnvWF}_{\mathsf{Prop}}(U_{\top}, u)$) ensures that the types to which the mapping is made are restricted to types that are defined by instantiating a type identifier, that these type identifiers are present in the pre-type environment, and all of the types in that environment can be elevated to full simple types if their self-mapping is removed from that environment.

4.4.2.6 Untimed type static semantic object

Meta-variable or constant	**Type name**	**Type definition**
$T_{U\top}$	$\mathscr{T}_{U\top} \leftarrow \mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\left\{ \begin{array}{l} t \in typeid: \\ \mathsf{utWFInEnv}_{\mathsf{Prop}}\Big(U_{\top}, \frac{U_{U\top}}{U_{\top}}, t\Big) \end{array} \right\}$
$\mathrm{T}_{U\bot}$	$\mathscr{T}_{\bot}$	$\emptyset$
$\mathbf{T}_{U\top\bot}$	$\mathfrak{T}_{U\top\bot} \leftarrow \mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\mathscr{T}_{U\top}\Big(U_{\top}, \frac{U_{U\top}}{U_{\top}}\Big) \bigsqcup \mathscr{T}_{U\bot}$

This is the full untimed OID-qualified type object. It states that an untimed OID with the given type identifier is well formed in the environment of the type and oid type environments. Both this fully qualified object, and its simple type analogue that we have already introduced, are objects whose existence is implied if an object can be inserted or has just been removed from a corresponding environment. The environment provides enough consistency guarantees itself, and the predicate of these stand-alone full types is in fact implied by such membership of the environment. We cannot, however, embed the full type directly in the environment because it would cause a type to be a parameter of itself: such structures are contradictory because they lack a base case . The predicate $\mathsf{utWFInEnv}_{\mathsf{Prop}}\Big(U_{\top}, \frac{U_{U\top}}{U_{\top}}, t\Big)$ asserts that the untimed OID type identifier t actually corresponds to an OID type defined in the untimed OID type environment (itself dependent on the plain type environment). We can tell the difference between untimed OID and plain type environments by the presence and absence of the subscript 'U' respectively.

4.4.2.7 Harmonic oid type environment static semantic object

Meta-variable or constant	**Type name**	**Type definition**
$U_{H\top}, \mathtt{U}_{H\top\varnothing}$	$\mathscr{U}_{H\top} \leftarrow \mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\left\{ \begin{array}{l} h \in \prod \left\lvert \begin{array}{l} t \in \mathscr{P}\left(typeid\right) : \\ t \rightarrow \mathscr{T}_{U\top}\left(U_{\top}, \frac{U_{U\top}}{U_{\top}}\right) \times freq \end{array} \right. \quad : \\ \mathsf{HOidEnvWF}_{\mathsf{Prop}}\left(U_{\top}, \frac{U_{U\top}}{U_{\top}}, h\right) \end{array} \right\}$
$\mathtt{U}_{H\bot}$	$\mathscr{U}_{H\bot}$	$\varnothing$
$\mathbf{U}_{H\top\bot}$	$\mathfrak{U}_{H\top\bot} \leftarrow \mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\mathscr{U}_{H\top}\left(U_{\top}, \frac{U_{U\top}}{U_{\top}}\right) \sqcup \mathscr{U}_{H\bot}$

This object defines harmonically timed OID types. These consist of untimed OID types combined with a frequency, thus defining an infinite set of equally spaced instances in time at which the untimed value may be instantiated. This could be thought of as a kind of dependent typing in which the time value of each slice of data constitutes another parameter to the timed type. However, this would complicate the formalization, and so we do not introduce time-dependent typing. The pattern of the harmonic type environment otherwise follows that of its untimed cousin exactly. The predicate $\mathsf{HOidEnvWF}_{\mathsf{Prop}}\left(U_{\top}, \frac{U_{U\top}}{U_{\top}}, h\right)$ states that the harmonic type h is well-formed in its OID type environment: that is, the untimed OID type from which it is built is present in the untimed OID type environment.

4.4.2.8 Harmonic type static semantic object

Meta-variable or constant	**Type name**	**Type definition**
$T_{H\top}$	$\mathscr{T}_{H\top} \leftarrow \mathscr{U}_{H\top} \leftarrow \mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\left\{ \begin{array}{l} t \in typeid : \\ \mathsf{htWFInEnv}_{\mathsf{Prop}} \\ \left(U_{\top}, \frac{U_{U\top}}{U_{\top}}, \frac{U_{H\top}}{(U_{U\top}, U_{\top})}, t\right) \end{array} \right\}$
$\mathtt{T}_{H\bot}$	$\mathscr{T}_{H\bot}$	$\varnothing$
$\mathbf{T}_{H\top\bot}$	$\mathfrak{T}_{H\top\bot} \leftarrow \mathscr{U}_{H\top} \leftarrow \mathscr{U}_{U\top} \leftarrow \mathscr{U}_{\top}$	$\mathscr{T}_{H\top}\left(U_{\top}, \frac{U_{U\top}}{U_{\top}}, \frac{U_{H\top}}{(U_{U\top}, U_{\top})}\right) \sqcup \mathscr{T}_{H\bot}$

This is the full timed OID-qualified type object, as qualified both by a predicate on the type identifier of interest and by a function that maps timed type indentifiers to untimed types (as resolved from the untimed type and simple type environments, which are also arguments). Again, this follows exactly the same pattern as the untimed cousin, except with an extra dependent parameter. The predicate $\mathsf{htWFInEnv}_{\mathsf{Prop}}\left(U_{\top}, \frac{u_{U\top}}{U_{\top}}, \frac{u_{H\top}}{(U_{U\top},U_{\top})}, t\right)$ provides that the type identifier t dereferences a harmonic type in the harmonic type environment.

4.4.2.9 The type environment

Meta-variable or constant	**Type name**	**Type definition**
$E_{T\top}$	$\mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} \mathscr{U}_{\top} \\ \mathscr{U}_{U\top}(U_{\top}) \\ \mathscr{U}_{H\top}\left(U_{\top}, \frac{u_{U\top}}{U_{\top}}\right) \end{array} \right.$
$\mathrm{E}_{T\bot}$	$\mathscr{E}_{T\bot}$	$\varnothing$
$\mathbf{E}_{T\top\bot}$	$\mathfrak{E}_{T\top\bot}$	$\mathscr{E}_{\top} \bigsqcup \mathscr{E}_{\bot}$

The type environment contains all of the plain, untimed OID and harmonic OID type mappings. The less dependent components index the more dependent components, but the overall type of the type environment has no type dependencies.

4.4.3 Coordination language

4.4.3.1 Untimed box variable mapping

Meta-variable or constant	**Type name**	**Type definition**
A_B	$\mathscr{A}_B \leftarrow \mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{T}_{U\top}(E_{T\top}.U_{\top}, E_{T\top}.U_{U\top}) \times \mathbb{N} \times \mathbb{N} \end{array} \right.$

This semantic category associates a variable name in the environment of the expression language instance (within a box) with an untimed OID type, along with two natural numbers that designate the minimum and maximum length of a tuple of this type that can be processed by the box language during one invocation of the expression. In all cases in this development, the minimum and maximum are further constrained to be equal, since the example expression language used only deals with fixed size data types. The environment objects use ‘dot’ notation to dereference the components of the overall type environment object; we omit the explicit indication of type dependency, since this can be inferred from the structure of the overall object.

4.4.3.2 Expression language static semantic object meta-type

Meta-variable or constant	Type name	Type definition
$X_{\mathsf{T}}\colon \mathscr{X}$	$\tau_{\mathscr{X}}$	$\tau \leftarrow \mathscr{A}_B{}' \leftarrow \mathscr{A}_B \leftarrow \mathscr{E}_{T\mathsf{T}}$

We now give the type for an expression language static semantic object for the first time, using the notation we have described for higher-order semantic types. This is the type of expression languages, not the type of a particular expression language, whose meta-variable is written $\mathscr{X}$. τ is the type of types, and the subscript $\mathscr{X}$ in the type name shows that the complete type name refers to the type of expression language types. In the type definition there are two variable type mappings (dependent on a type environment $\mathscr{E}_{T\mathsf{T}}$). One is for input ($\mathscr{A}_B$), the other for output ($\mathscr{A}_B{}'$).

4.4.3.3 Expression type box binding

Meta-variable or constant	Type name	Type definition
D_{UM}	$\mathscr{D}_{\mathrm{UM}} \leftarrow \mathscr{A}_B \leftarrow \mathscr{E}_{T\mathsf{T}}$	$\left\{ \begin{array}{l} r \in \prod \left\vert \begin{array}{l} v \in \mathscr{P}(\mathit{varid}) : \\ v \to D_{\_} \end{array} \right. \quad : \\ \mathsf{exprDat}_{\mathsf{Prop}}\left(E_{T\mathsf{T}}, \frac{A_B}{E_{T\mathsf{T}}}, r\right) \end{array} \right\}$

The static semantic object of the expression language is required to have a single record for its input type and a single record for its output type. These are ordinary records over and belonging to the simple type system, while the signature of an untimed box is given in terms of a set of sequences of OID-qualified values associated with it. Each such sequence is related to a variable identifier using an associative mapping. This requires a binding that establishes a bijective relationship between this set of elevated types in the coordination environment and the function over record types provided by the expression language. This relationship is described by the predicate $\mathsf{exprDat}_{\mathsf{Prop}}\left(E_{T\mathsf{T}}, \frac{A_B}{E_{T\mathsf{T}}}, r\right)$, which generates a σ-type over maps in the basic type system values as parametrized by the map of OID associations (the 'UM' subscript is for 'untimed map'): we give this enriched type of base data below, prior to defining the form of the untimed expression language object. A refinement of this approach, which we leave to further work, might include allowing a state variable to be kept between invocations. This would enable us to use an arbitrary coordination language in place of an expression language, running it in bursts and keeping its state between bursts in the state variable.

4.4.3.4 Untimed box expression language semantic predicate

Meta-variable or constant	Type name	Type definition
$\mathsf{exprSem}_{\mathsf{Prop}}$	$\kappa_{\mathsf{exprSem}} \leftarrow \tau_{\mathscr{X}} \leftarrow \mathscr{A}_B{}' \leftarrow \mathscr{A}_B \leftarrow \mathscr{E}_{T\top}$	$\kappa \leftarrow \mathscr{D}_{\mathrm{UM}}\left(\frac{A_B}{E_{T\top}} = \frac{A_B{}'}{E_{T\top}}\right) \leftarrow \mathscr{D}_{\mathrm{UM}}\left(\frac{A_B}{E_{T\top}} = \frac{A_B}{E_{T\top}}\right) \leftarrow \mathscr{X}(E_{T\top}, A_B, A_B{}')$

This (higher order) type gives the type of predicate defining the subset of encodings that are valid for a particular input and output signature and raw expression language. This is further parametrized on the input and output *values*, thus giving a relation on first order quantities, making the relation directly intelligible to the coordination semantics of HBCL. The equalities in parentheses, giving explicit dependent type arguments to the types of untimed data, are needed because there are two different box bindings present (one not primed, one primed) for input and output bindings.

4.4.3.5 Untimed box expression language result

Meta-variable or constant	Type name	Type definition
D_{UMO}	$\mathscr{D}_{\mathrm{UMO}} \leftarrow \kappa_{\mathsf{exprSem}} \leftarrow \tau_{\mathscr{X}} \leftarrow \mathscr{A}_B{}' \leftarrow \mathscr{A}_B \leftarrow \mathscr{E}_{T\top}$	$\sigma\left(\mathsf{exprSem}_{\mathsf{Prop}}\left(\frac{X_\top}{A_B{}' A_B E_{T\top}}, \frac{D_{\mathrm{UM}}}{A_B E_{T\top}}\right)\right) \leftarrow \mathscr{D}_{\mathrm{UM}}\left(\frac{A_B}{E_{T\top}} = \frac{A_B}{E_{T\top}}\right) \leftarrow \mathscr{X}(E_{T\top}, A_B, A_B{}')$

This object gives the type of output data as parametrized by the input data and the semantics of the language. The ‘O’ appended to the subscript in the name of the data object signifies that it is an input-qualified *output* type.

4.4.3.6 Untimed box expression language

Meta-variable or constant	Type name	Type definition
A	$\mathscr{A} \leftarrow \mathscr{A}_B{}' \leftarrow \mathscr{A}_B \leftarrow \mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} \tau_{\mathscr{X}} \\ \kappa_{\mathsf{exprSem}}(E_{T\top}, A_B, A_B{}', \mathscr{X}) \\ \\ \mathsf{implExists}_{\mathsf{Prop}} : \exists \mathrm{impl}, \\ \mathrm{impl} \in \begin{array}{l} \mathscr{X}(E_{T\top}, A_B, A_B{}') \to \\ \mathscr{D}_{\mathrm{UM}}\left(\frac{A_B}{E_{T\top}} = \frac{A_B}{E_{T\top}}\right) \to \\ \frac{\mathscr{D}_{\mathrm{UMO}}}{E_{T\top}, \frac{A_B}{E_{T\top}}, \frac{A_B{}'}{E_{T\top}}, \mathscr{X}(E_{T\top}, A_B, A_B{}')\mathsf{exprSem}_{\mathsf{Prop}}}\left(\frac{X_\top}{A_B{}' A_B E_{T\top}}, \frac{D_{\mathrm{UM}}}{A_B E_{T\top}} \right) \end{array} \end{array} \right.$

The untimed box expression language is defined by the Cartesian product of three things. First, it contains a static semantic object type. Second, it has a predicate on that static semantic object, input and output types, and data that expresses the big-step semantics as a relation between inputs and outputs. Third, it provides proof of existence of an implementation, which can be discharged by providing a total function that produces inhabitants of the result type (a σ-type over outputs, parametrized by inputs).

4.4.3.7 Memory specification object

Meta-variable or constant	Type name	Type definition
M	$\mathscr{M} \leftarrow \mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} freq \\ \mathscr{T}_{H\perp}(E_{T\top}.U_\top, E_{T\top}.U_{U\top}, E_{T\top}.U_{H\top}) \\ \mathbb{Z} \\ \mathbb{N} \\ \mathbb{N} \end{array} \right.$

The memory specification object combines a frequency, a harmonic type, a time to and from live as a signed integer, and (for the purposes of the present implementation for reasons discussed in appendix A.5.8.2) a minimum and maximum memory length. It is dependent in the type of a full type environment, to ensure that the types referred to are fully defined.

4.4.3.8 Memory specification object map by variable identifier

Meta-variable or constant	Type name	Type definition
M_{MV}	$\mathscr{M}_{\text{MV}} \leftarrow \mathscr{E}_{T\text{T}}$	$\prod \left\lvert \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{M}(E_{T\text{T}}) \end{array} \right.$

This map gives a set of variable identifiers and, for each, specifies a memory to which it refers.

4.4.3.9 Raw memory data object

Meta-variable or constant	Type name	Type definition
N	$\mathscr{N}$	$\prod_{i=0}^{n} \mathscr{D}_{\text{-}i} \times \mathbb{N}$

The raw memory data object is a tuple of data values, each being associated with a natural number, which specifies a unique time when interpreted according to a frequency (this interpretation is added in the complete memory data object below).

4.4.3.10 Memory data by variable identifier

Meta-variable or constant	Type name	Type definition
N_{MV}	$\mathscr{N}_{\text{MV}} \leftarrow \mathscr{M}_{\text{MV}} \leftarrow \mathscr{E}_{T\text{T}}$	$w \in \prod \left\lvert \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{N} \end{array} \right. :$ $\mathsf{memMapWF}_{\mathsf{Prop}}\left(E_{T\text{T}}, \frac{M_{\text{MV}}}{E_{T\text{T}}}, w\right)$

This object specifies a map of memory data objects by variable identifier, and ensures that each such memory matches the corresponding memory type in the map of memory types (supplied in one of the type parameters to the map of memories).

4.4.3.11 Harmonic box binding specification

Meta-variable or constant	Type name	Type definition
B	$\mathscr{B} \leftarrow$ $\mathscr{M}_{\mathrm{MV}}' \leftarrow$ $\mathscr{M}_{\mathrm{MV}} \leftarrow$ $\left(\forall f : \mathit{freq}\right) \leftarrow$ $\mathscr{A}_B' \leftarrow$ $\mathscr{A}_B \leftarrow$ $\mathscr{E}_{T\top}$	$\prod$ $\kappa_{\mathsf{boxMemFreqCompatIn}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}}{E_{T\top}}, f\right)$ $\kappa_{\mathsf{boxMemFreqCompatOut}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}'}{E_{T\top}}, f\right)$ $\kappa_{\mathsf{boxMemCompatIn}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}}{E_{T\top}}, \frac{A_B}{E_{T\top}}\right)$ $\kappa_{\mathsf{boxMemCompatOut}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}'}{E_{T\top}}, \frac{A_B'}{E_{T\top}}\right)$ $\kappa_{\mathsf{boxDataConvIn}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}}{E_{T\top}}, f, \frac{A_B}{E_{T\top}}\right) \leftarrow \mathscr{D}_{\mathrm{UM}}(E_{T\top}, A_B) \leftarrow \mathscr{N}_{\mathrm{MV}}\left(E_{T\top}, \frac{M_{MV}}{E_{T\top}}\right)$ $\kappa_{\mathsf{boxDataConvOut}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}'}{E_{T\top}}, f, \frac{A_B'}{E_{T\top}}\right) \leftarrow \mathscr{N}_{\mathrm{MV}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}'}{E_{T\top}}\right) \leftarrow \mathscr{D}_{\mathrm{UM}}(E_{T\top}, A_B')$ $\mathsf{convInExists}_{\mathsf{Prop}}$: $\exists \mathrm{impl},\ \mathrm{impl} \in \begin{array}{l} \mathscr{N}_{\mathrm{MV}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}}{E_{T\top}}\right) \rightarrow \\ \sigma\left(\mathsf{boxDataConvIn}_{\mathsf{Prop}}\left(\frac{N_{\mathrm{MV}}}{M_{\mathrm{MV}} E_{T\top}}\right)\right) \end{array}$ $\mathsf{convOutExists}_{\mathsf{Prop}}$: $\exists \mathrm{impl},\ \mathrm{impl} \in \begin{array}{l} \mathscr{D}_{\mathrm{UM}}\left(E_{T\top}, \frac{A_{B\top}'}{E_{T\top}}\right) \rightarrow \\ \sigma\left(\mathsf{boxDataConvOut}_{\mathsf{Prop}}\left(\frac{D_{UM}}{A_{B\top}', E_{T\top}}\right)\right) \end{array}$

The harmonic box language wraps an untimed box language within a temporal wrapper that specifies mappings between the timed (oid) types of the coordination language and the untimed oid types of an untimed box language. It consists of the definitions of the types of six predicates, which are used to constrain the untimed boxes that can be instantiated within a harmonic box in a semantically consistent way. The first two predicates restrict the input and output maps of memory specifications with respect to a box frequency. This is needed because, although the frequency of the memory map as a whole is determined by the lowest common multiple of frequencies, in a fully general case, a harmonic box may have some other frequency. For present purposes, however, these predicates are used to ensure that a box frequency is an integral multiple of the frequency of these memory maps. This precludes complications to the semantics that would arise from variable minimum numbers of time slices being available in each memory (the variability would be caused as the boxes fell in and out of phase with the memory maps). The next two predicates ensure that input and output memory maps, as qualified by the phase restricting predicates, are compatible with the untimed box input and output interfaces. The final two predicate types define the acceptable data value produced from a particular input memory and the acceptable output memory produced from an output data type. The object of the two conversion functions is to find the single inhabitant of the σ-types derived from these predicates. The last two elements of the hamonic box binding are the type of existential proof of a function generating such σ-types satisfying these last two predicates from suitably constrained inputs.

4.4.3.12 Harmonic box (untimed box content version)

Meta-variable or constant	**Type name**	**Type definition**
$H_{U\top}$	$\mathscr{H}_{U\top} \leftarrow$ $\mathscr{M}_{\mathrm{MV}}' \leftarrow$ $\mathscr{M}_{\mathrm{MV}} \leftarrow$ $\left(\forall f : \mathit{freq}\right) \leftarrow$ $\mathscr{E}_{T\top}$	$\Pi \left\| \begin{array}{l} \mathscr{A}_B \\ \mathscr{A}_B{}' \\ \mathscr{A}\left(E_{T\top}, \frac{A_B}{E_{T\top}}, \frac{A_B{}'}{E_{T\top}}\right) \\ \mathscr{B}\left(E_{T\top}, \frac{A_B}{E_{T\top}}, \frac{A_B{}'}{E_{T\top}}, f, \frac{M_{\mathrm{MV}}}{E_{T\top}}, \frac{M_{\mathrm{MV}}{}'}{E_{T\top}}\right) \\ A.\mathscr{X}\left(E_{T\top}, \frac{A_B}{E_{T\top}}, \frac{A_B{}'}{E_{T\top}}\right) \\ \\ \quad A.\mathscr{X}\,(E_{T\top}, A_B, A_B{}') \to \mathscr{D}_{\mathrm{UM}}\left(\frac{A_B}{E_{T\top}} = \frac{A_B}{E_{T\top}}\right) \to \\ \quad \mathscr{D}_{\mathrm{UMO}}\left(\begin{array}{l} A.\mathscr{X}\,, \mathsf{exprSem}_{\mathsf{Prop}} \\ E_{T\top}, \frac{A_B}{E_{T\top}}, \frac{A_B{}'}{E_{T\top}} \\ \frac{X_\top}{A_B{}' A_B E_{T\top}}, \frac{D_{\mathrm{UM}}}{A_B E_{T\top}} \end{array}\right) \\ \\ \mathscr{N}_{\mathrm{MV}}\left(E_{T\top}, \frac{M_{\mathrm{MV}}}{E_{T\top}}\right) \to \sigma\left(B.\mathsf{boxDataConvIn}_{\mathsf{Prop}}\left(\frac{N_{\mathrm{MV}}}{M_{\mathrm{MV}} E_{T\top}}\right)\right) \\ \\ \mathscr{D}_{\mathrm{UM}}\left(E_{T\top}, \frac{A_{B\top}{}'}{E_{T\top}}\right) \to \sigma\left(B.\mathsf{boxDataConvOut}_{\mathsf{Prop}}\left(\frac{D_{UM}}{A_{B\top}', E_{T\top}}\right)\right) \end{array}\right.$

A harmonic box contains the program that is to be run each time the box is executed. It is dependent in the type of the type environment, its frequency of execution, and the specification of its input and output memories. The product type of its specification binds the input and data maps to plain data types for the use of the expression language ($\mathscr{A}_B$ and $\mathscr{A}_B{}'$). It also fixes the untimed box expression language and its temporal binding, the untimed box language specification $\mathscr{A}$, and the specification of how the input and output memories and plain data types are mapped to each other ($\mathscr{B}$). $\mathscr{A}.\mathscr{X}$ is a static semantic object for the expression language $\mathscr{A}$. The final three components are implementations of the expression language with the input and output conversion functions.

4.4.3.13 Input memory specification object map by memory identifier

Meta-variable or constant	Type name	Type definition
M_{MOI}	$\mathscr{M}_{\mathrm{MOI}} \leftarrow \mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{memFB}) : \\ o \rightarrow \mathscr{M}(E_{T\top}) \end{array} \right.$

This object maps input memory object identifiers to memory specification objects. The OIDs are indicated as belonging to FIFO-box memory OIDs.

4.4.3.14 Output memory specification object map by memory identifier

Meta-variable or constant	Type name	Type definition
M_{MOO}	$\mathscr{M}_{\mathrm{MOO}} \leftarrow \mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{memBF}) : \\ o \rightarrow \mathscr{M}(E_{T\top}) \end{array} \right.$

This object maps output memory object identifiers to memory specification objects. In contrast to the similar input memory map, the OIDs are indicated as belonging to box-FIFO memory OIDs.

4.4.3.15 Harmonic box with memory identifier binding

Meta-variable or constant	Type name	Type definition
$H_{O\top}$	$\mathscr{H}_{O\top} \leftarrow$ $\mathscr{M}_{\mathrm{MOO}} \leftarrow$ $\mathscr{M}_{\mathrm{MOI}} \leftarrow$ $\left(\forall f : freq\right) \leftarrow$ $\mathscr{E}_{T\top}$	$\prod \left\lvert \begin{array}{l} \mathscr{M}_{\mathrm{MV}}(E_{T\top}) \\ \mathscr{M}_{\mathrm{MV}}'(E_{T\top}) \\ \mathscr{H}_{\top}\left(E_{T\top}, f, \frac{M_{MV}}{E_{T\top}}, \frac{M_{MV}'}{E_{T\top}}\right) \\ i : \frac{M_{MV}}{E_{T\top}} \leftrightarrow \frac{M_{MOI}}{E_{T\top}} \\ o : \frac{M_{MV}'}{E_{T\top}} \leftrightarrow \frac{M_{MOO}}{E_{T\top}} \end{array} \right.$

The harmonic box object with its memory identifier binding provides the mapping from the map of memories as plain variables (this is what the memories look like from the inside of the box) with the corresponding OID-indexed memory specifications, as they are viewed from the outside of the box. The OID memories are dependent arguments, along with the box frequency and type environment, while the variable maps are hidden inside the product, along with the harmonic box specification and a bijective map of variables to OIDs for input and output memories. The harmonic box specification object does not have a 'U' subscript because it need not in general be restricted to a harmonic box implementation that wraps an untimed box language. The final two arguments of the untimed harmonic box are the input and output conversion functions. These are not specified in the box binding *B*, which only requires proof of their existence.

4.4.3.16 Logical instance signature

Meta-variable or constant	Type name	Type definition
I_{ST}	$\mathscr{I}_{\mathrm{ST}}$	$\prod \left\| \begin{array}{l} \mathscr{E}_{TT} \\ \mathscr{M}_{\mathrm{MOO}}(E_{TT}) \\ \mathscr{M}_{\mathrm{MOI}}(E_{TT}) \end{array} \right.$

The logical instance signature is the type environment, together with a mapping of input and output memory OIDs to memory specifications dependent on that type environment.

4.4.3.17 Logical instance closure signature

Meta-variable or constant	Type name	Type definition
I_{CT}	$\mathscr{I}_{\mathrm{CT}}$	$\prod \left\| \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \rightarrow \mathscr{I}_{\mathrm{ST}} \times \prod \left\| \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \rightarrow \mathscr{I}_{\mathrm{ST}} \end{array} \right. \end{array} \right.$

The signature of a logical instance closure is a map of logical instance identifiers to a map of logical instance signatures. It declares which instances must be supplied by the environment in order to form a complete logical instance closure, in which every instance is fully specified without any free parameters.

4.4.3.18 Logical instance and logical library raw static semantic object

Meta-variable or constant	Type name	Type definition
I_{T}	$\mathscr{I}_{\mathsf{T}} \leftarrow$ $\mathscr{I}_{\mathsf{ST}} \leftarrow \mathscr{I}_{\mathsf{CT}}$	$\prod \left\lvert \begin{array}{l} \mathscr{E}_{T\mathsf{T}} \\ \mathscr{M}_{\mathrm{MOI}}(I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}) \\ \mathscr{M}_{\mathrm{MOO}}(I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}) \\ hboxes : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{hbox}) : \\ o \to \prod \left\lvert \begin{array}{l} \forall f : freq \\ \mathscr{M}_{\mathrm{MOI}}{}'(I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}) \\ \mathscr{M}_{\mathrm{MOO}}{}'(I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}) \\ \mathscr{H}_{\mathrm{O}\mathsf{T}}\left(\begin{array}{c} (I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}), f \\ \frac{M_{\mathrm{MOI}}{}'}{I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}}, \frac{M_{\mathrm{MOO}}{}'}{I_{S\mathsf{T}}.E_{T\mathsf{T}} \oplus E_{T\mathsf{T}}} \end{array} \right) \end{array} \right. \end{array} \right. \\ obs : \prod \left\lvert \begin{array}{l} o, o' \in \mathscr{P}(\mathscr{O}_{memFB}) : \\ o \leftrightarrow o' \end{array} \right. \\ manif : \prod \left\lvert \begin{array}{l} o, o' \in \mathscr{P}(\mathscr{O}_{memBF}) : \\ o \leftrightarrow o' \end{array} \right. \\ fifos : \prod \left\lvert \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{O}_{memBF} \times \mathscr{O}_{memFB} \end{array} \right. \\ linstnest : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \to \prod \left\lvert \begin{array}{l} \mathscr{I}_{\mathrm{C}\mathsf{T}} \\ \mathscr{I}_{\mathrm{S}\mathsf{T}}(\mathscr{I}_{\mathrm{C}\mathsf{T}}) \\ \mathscr{I}_{\mathsf{T}}(I_{\mathrm{C}\mathsf{T}}, I_{S\mathsf{T}}) \end{array} \right. \end{array} \right. \\ liblinsts : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \to \mathscr{O}_{linst} \end{array} \right. \\ linstttfl : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \to \mathbb{Z} \end{array} \right. \\ nestlibs : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{llib}) : \\ o \to \prod \left\lvert \begin{array}{l} \mathscr{I}_{\mathrm{C}\mathsf{T}} \\ \mathscr{E}_{T\mathsf{T}} \\ \mathscr{L}(I_{\mathrm{C}\mathsf{T}}, E_{T\mathsf{T}}) \end{array} \right. \end{array} \right. \end{array} \right.$
L_{T}	$\mathscr{L}_{\mathsf{T}} \leftarrow \mathscr{I}_{\mathrm{C}\mathsf{T}}$	$\prod \left\lvert \begin{array}{l} \mathscr{E}_{T\mathsf{T}} \\ linsts : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \to \prod \left\lvert \begin{array}{l} \mathscr{I}_{\mathrm{C}\mathsf{T}} \\ \mathscr{I}_{\mathrm{S}\mathsf{T}} \\ \mathscr{I}_{\mathsf{T}}(I_{\mathrm{C}\mathsf{T}}, I_{S\mathsf{T}}) \end{array} \right. \end{array} \right. \\ nestlibs : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{llib}) : \\ o \to \prod \left\lvert \begin{array}{l} \mathscr{I}_{\mathrm{C}\mathsf{T}} \\ \mathscr{E}_{T\mathsf{T}} \\ \mathscr{L}(I_{\mathrm{C}\mathsf{T}}, E_{T\mathsf{T}}) \end{array} \right. \end{array} \right. \end{array} \right.$

Logical instances and libraries form a mutually dependent hierarchy, as described in chapter 3. The raw objects we specify here invite qualification by consistency predicates, but, as with many predicates, we leave filling these out to further work, since we have prioritized progressing to operational semantics in order to achieve a working interpreter, according to the stated methodology.

We now briefly describe the members of these Cartesian products in the order they appear. Logical instances are dependent in the type of their signature and closure signature; logical libraries are dependent in the type of an instance closure signature sufficient to completely define all recursively nested instances and libraries.

The logical instance contains the usual type environment object, along with OID maps of local input and output memory specifications that are dependent on the exclusive sum of the local signature type environment and the locally declared type environment: in other words, those type environments must not contain overlapping key sets. The *hboxes* field is a mapping of box identifier OIDs to harmonic boxes, which are specified with another product in order to contain the necessary dependent type arguments for each box. The *obs* field contains a set of OIDs exposed to the outside world and a set of OIDs inside the scope of the local instance, which might be locally defined memories, or the memories exposed by nested instances. A bijective mapping is given between these two sets. The *manif* field does the same for 'manifestations', or output memories. The *fifo* field is a mapping from a name for the FIFO to the Cartesian product of an output memory and an input memory. The *linstnest* is a map of nested instances; *liblinsts* is a map of local instance identifiers to the library-qualified instance which is being instantiated to define that instance. *linstttfl* is a map which moves the timescale within an instance backwards or forwards relative to the enclosing instance. This becomes essential when multiple instances of the same specification are pipelined, where the temporal semantic value of their inputs and outputs clearly shifts, but from the inside of the instance, we of course wish the semantics to remain self-consistent and invariant. The nested instance does not 'know' that it is being deceived as to the real wall-clock time. *nestlibs* are just the nested libraries that are only available within that instance or other instances or libraries it encloses.

Logical libraries contain a type environment, a map of nested logical instances and a map of nested logical libraries.

4.4.3.19 Time object

Meta-variable or constant	Type name	Type definition
t	$\mathcal{t} \leftarrow \left(\forall f : freq\right)$	$\mathbb{N}$

The time object is a natural number counting ticks since time zero. The rate of the ticks is determined by the frequency in the dependent type of the time.

4.4.3.20 Memory data by input memory object identifier

Meta-variable or constant	Type name	Type definition
N_{MOI}	$\mathscr{N}_{\text{MOI}} \leftarrow \mathscr{M}_{\text{MOI}} \leftarrow \mathscr{E}_{T\mathsf{T}}$	$p \in \prod \left\| \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{memFB}) : \\ o \to \mathscr{N} \end{array} \right. : \mathsf{memMapOidWF}_{\mathsf{Prop}}\left(E_{T\mathsf{T}}, \frac{M_{\text{MOI}}}{E_{T\mathsf{T}}}, p\right)$

This is the version of the input map for memories indexed by OIDs.

4.4.3.21 Input stream

Meta-variable or constant	Type name	Type definition
S	$\mathscr{S} \leftarrow t \leftarrow \mathscr{M}_{\text{MOI}} \leftarrow \left(\forall f : freq\right) \leftarrow \mathscr{E}_{T\mathsf{T}}$	$\coprod \left\| \begin{array}{l} \varnothing \\ \mathscr{N}_{MOI\mathsf{T}}\left(E_{T\mathsf{T}}, f, M_{MOI\mathsf{T}}, t\right) \end{array} \right.$

This object describes a coinductive input stream.

4.4.3.22 Memory data by output memory object identifier

Meta-variable or constant	Type name	Type definition
N_{MOO}	$\mathscr{N} \leftarrow \mathscr{M}_{\text{MOO}} \leftarrow \mathscr{E}_{T\mathsf{T}}$	$p \in \prod \left\| \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{memBF}) : \\ o \to \mathscr{N}_{MOO} \end{array} \right. : \mathsf{memMapOidWF}_{\mathsf{Prop}}\left(E_{T\mathsf{T}}, \frac{M_{\text{MOO}}}{E_{T\mathsf{T}}}, p\right)$

This is the version of the output map for memories indexed by OIDs.

4.4.3.23 Time-agnostic coordination dynamic semantic object

Meta-variable or constant	**Type name**	**Type definition**
$C_{\top}$	$\mathscr{C}_{\top} \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\prod \left\lvert \begin{array}{l} \mathscr{N}_{\mathrm{MOI}}((I_{S\top}.E_{T\top}) \oplus E_{T\top}, M_{\mathrm{MOI}}) \\ \mathscr{N}_{\mathrm{MOO}}((I_{S\top}.E_{T\top}) \oplus E_{T\top}, M_{\mathrm{MOO}}) \\ \mathit{coordnest} : \prod \left\lvert \begin{array}{l} o \in \mathscr{P}(\mathscr{O}_{linst}) : \\ o \rightarrow \prod \left\lvert \begin{array}{l} \mathscr{I}_{C\top} \\ \mathscr{I}_{S\top} \\ \mathscr{I}_{\top} \\ \mathscr{C}_{\top}(I_{C\top}, I_{S\top}, I_{\top}) \end{array} \right. \end{array} \right. \end{array} \right.$

The time-agnostic coordination dynamic semantic object consists of a map of input memories, a map of output memories, and a map of nested dynamic coordination objects.

4.4.3.24 Time-qualified coordination dynamic semantic objects

Meta-variable or constant	**Type name**	**Type definition**
$C_{\mathrm{Boxes}\top}$	$\mathscr{C}_{\mathrm{Boxes}\top} \leftarrow \left(\forall f : \mathit{freq}\right) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\sigma\left(\mathsf{Boxes}_{\mathsf{Prop}}\left(I_{C\top}, I_{S\top}, I_{\top}, f_{\top}, t_{\top}\right)\right)$
$C_{\mathrm{MemBF}\top}$	$\mathscr{C}_{\mathrm{MemBF}\top} \leftarrow \left(\forall f : \mathit{freq}\right) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\sigma\left(\mathsf{MemBF}_{\mathsf{Prop}}\left(I_{C\top}, I_{S\top}, I_{\top}, f_{\top}, t_{\top}\right)\right)$
$C_{\mathrm{FIFOs}\top}$	$\mathscr{C}_{\mathrm{FIFOs}\top} \leftarrow \left(\forall f : \mathit{freq}\right) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\sigma\left(\mathsf{FIFOs}_{\mathsf{Prop}}\left(I_{C\top}, I_{S\top}, I_{\top}, f_{\top}, t_{\top}\right)\right)$
$C_{\mathrm{InnerFIFOs}\top}$	$\mathscr{C}_{\mathrm{InnerFIFOs}\top} \leftarrow \left(\forall f : \mathit{freq}\right) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\sigma\left(\mathsf{InnerFIFOs}_{\mathsf{Prop}}\left(I_{C\top}, I_{S\top}, I_{\top}, f_{\top}, t_{\top}\right)\right)$
$C_{\mathrm{MemFB}\top}$	$\mathscr{C}_{\mathrm{MemFB}\top} \leftarrow \left(\forall f : \mathit{freq}\right) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\sigma\left(\mathsf{MemFB}_{\mathsf{Prop}}\left(I_{C\top}, I_{S\top}, I_{\top}, f_{\top}, t_{\top}\right)\right)$

These objects are σ-types that use a predicate on the time-agnostic coordination object to signify that particular stages of execution are enabled. $C_{\mathrm{Boxes}\top}$ is the coordination state between the execution of FIFO-box memories and boxes. The predicate $\mathsf{Boxes}_{\mathsf{Prop}}$ requires that the state of box input memories for runnable boxes is ready to be read and that the state of output memories is ready to be written. $C_{\mathrm{MemBF}\top}$ is the coordination state between the execution of boxes and box-FIFO memories. The predicate $\mathsf{MemBF}_{\mathsf{Prop}}$ requires that all boxes connected to the memory that are scheduled to have appended values have written them, and that all FIFOs that could have read values that are about to be deleted have read them. $C_{\mathrm{FIFOs}\top}$ is the coordination state between the execution of box-FIFO memories and FIFOs. The predicate $\mathsf{FIFOs}_{\mathsf{Prop}}$ requires that the box-FIFO memories are ready to be read and the FIFO-box memories are ready to be written, so that all FIFOs within scope are ready to be executed. $C_{\mathrm{InnerFIFOs}\top}$ is the coordination state of nested logical instances that are ready for inner FIFO execution. The predicate $\mathsf{InnerFIFOs}_{\mathsf{Prop}}$ requires that those box-FIFO memories that are not exported to an enclosing instance are ready for FIFO execution. Those memories that have been exported and thus already processed are already in the box-FIFO memory-enabled state. $C_{\mathrm{MemFB}\top}$ is the coordination state betweeen the execution of FIFOs and FIFO-box memories. The predicate $\mathsf{MemFB}_{\mathsf{Prop}}$ requires that all FIFOs connected to memories that are scheduled to have appended values have done so, and that all boxes that could have read values that are about to be deleted have read them. These execution steps are described further in appendix C.3.

Versions of the coordination state variable prefixed by a 'μ', such as μC_{FIFOs}, indicate a map of coordination objects indexed by a local logical index identifier. This is a shorthand for the type of *coordnest* in the concrete structure of each (time agnostic) coordination type.

4.4.3.25 Trace dynamic semantic objects

Meta-variable or constant	**Type name**	**Type definition**
$\mathrm{Tr}_{\mathrm{Boxes}\top}$	$\mathscr{Tr}_{\mathrm{Boxes}\top} \leftarrow \mathscr{C}_{\mathrm{Boxes}\top} \leftarrow (\forall f : \mathit{freq}) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_\top \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\prod \left\| \begin{array}{l} \frac{C_{\mathrm{MemBF}\top}}{I_{C\top} I_{S\top} I_\top f_\top t_\top} \\ \frac{\mathrm{Tr}_{\mathrm{MemBF}\top}}{I_{C\top} I_{S\top} I_\top f_\top t_\top C_{\mathrm{MemBF}\top}} \end{array} \right.$
$\mathrm{Tr}_{\mathrm{MemBF}\top}$	$\mathscr{Tr}_{\mathrm{MemBF}\top} \leftarrow \mathscr{C}_{\mathrm{MemBF}\top} \leftarrow (\forall f : \mathit{freq}) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_\top \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\prod \left\| \begin{array}{l} \frac{C_{\mathrm{FIFOs}\top}}{I_{C\top} I_{S\top} I_\top f_\top t_\top} \\ \frac{\mathrm{Tr}_{\mathrm{FIFOs}\top}}{I_{C\top} I_{S\top} I_\top f_\top t_\top C_{\mathrm{FIFOs}\top}} \end{array} \right.$
$\mathrm{Tr}_{\mathrm{FIFOs}\top}$	$\mathscr{Tr}_{\mathrm{FIFOs}\top} \leftarrow \mathscr{C}_{\mathrm{FIFOs}\top} \leftarrow (\forall f : \mathit{freq}) \leftarrow \mathscr{t} \leftarrow \mathscr{I}_\top \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\coprod \left\| \begin{array}{l} \varnothing \\ \prod \left\| \begin{array}{l} \frac{C_{\mathrm{MemFB}\top}}{I_{C\top} I_{S\top} I_\top f_\top t_\top} \\ \frac{\mathrm{Tr}_{\mathrm{MemFB}\top}}{I_{C\top} I_{S\top} I_\top f_\top t_\top C_{\mathrm{MemFB}\top}} \end{array} \right. \end{array} \right.$

Meta-variable or constant	Type name	Type definition
$\mathrm{Tr}_{\mathrm{MemFB}\top}$	$\mathscr{Tr}_{\mathrm{MemFB}\top} \leftarrow \mathscr{C}_{\mathrm{MemFB}\top} \leftarrow \left(\forall f : freq\right) \leftarrow t \leftarrow \mathscr{I}_{\top} \leftarrow \mathscr{I}_{S\top} \leftarrow \mathscr{I}_{C\top}$	$\prod \left\vert \begin{array}{l} \frac{t'_{\top}}{f_{\top}} \\ \frac{C_{\mathrm{Boxes}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t'_{\top}} \\ \frac{\mathrm{Tr}_{\mathrm{Boxes}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t'_{\top} C_{\mathrm{Boxes}\top}} \\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\frac{t_{\top}}{f_{\top}}, \frac{t'_{\top}}{f_{\top}}\right) \end{array} \right.$

These objects contain sequences of time-qualified coordination objects to form execution traces. In each case, the type definition is the Cartesian product of the current coordination state and the next part of the trace (until infinity or the end of the execution). Trace structures are therefore *co*-inductive. This trace is implicitly a σ-type: in each step, there is also given in each constructor a predicate that captures the dynamic semantics; we describe them now.

4.4.3.25.1 Boxes step predicate description

The predicate required in building a trace with a boxes state at its head is a relation over $\frac{C_{\mathrm{Boxes}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$ and $\frac{C_{\mathrm{MemBF}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$. In each case, the instance $I_{\top}$ with frequency f defines the static scope of the object, and is present among the dependent arguments in the case of both coordination objects. All memory states remain the same, except for the box-FIFO memories of those to which have been added the products of a box execution. The only boxes (or inner coordination states) that are executed are for which $\frac{1}{f_{\mathrm{box}}} \mid t_{\mathrm{current}}$.

4.4.3.25.2 Box-FIFO step predicate description

Enabled memories at a particular instant have a simple relation between $\frac{C_{\mathrm{MemBF}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$ and $\frac{C_{\mathrm{FIFOs}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$. Each activated memory discards values that have expired and can no longer be read. In an implementation, the operation also involves copying data received in the last memory cycle from a write buffer to a read buffer. The activated memories are those for which $\frac{1}{f_{\mathrm{MemBF}}} \mid t_{\mathrm{current}}$.

4.4.3.25.3 FIFOs step predicate description

At present, FIFOs in HBCL are restricted to execute at the same frequency of their connected memories, which must be the same. The predicate over $\frac{C_{\mathrm{FIFOs}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$ and $\frac{C_{\mathrm{MemFB}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$ works similarly to the boxes predicate, except there is no computational content, and values that are new on this time slice in the box-FIFO memory at the 'in' end of each activated FIFO are instead just appended to the FIFO-box memory at the 'out' end of each FIFO. In further work, this would be refined so that FIFOs themselves had associated state variables, as the current arrangement requires buffers of indeterminate length in FIFO-box memories to accommodate FIFOs of different lengths. Again, as with boxes, the FIFOs activated at any instant are those for which $\frac{1}{f_{\text{FIFO}}} \mid t_{\text{current}}$.

4.4.3.25.4 FIFO-box step predicate description

The relation between $\frac{C_{\text{MemFB}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t_{\top}}$ and $\frac{C_{\text{Boxes}\top}}{I_{C\top} I_{S\top} I_{\top} f_{\top} t'_{\top}}$ is exactly the same as for the box-FIFO memory execution, except that another set of memories are involved, and that this time the next enabled step (boxes enabled) is at the next time slice $t'_{\top}$ occurs $\frac{1}{f}$ after $t_{\top}$.

4.4.4 Expression language

In defining an expression language, the aim is to produce a member of the type of expression languages. The object of the present section is to provide the type of the static semantic object, which is a parameter to the type of the big-step expression predicate, and which is given as a Curried argument to a correct interpreter of the expression language. We do not define the big-step predicate here, since, according to our stated methodology, we have prioritized the operational semantics, which give rise directly to an interpreter. The idea is that the predicate can be developed by progressively adding to its structure, and the interpreter modulo code extraction (and concomitant discarding of predicates) is unchanged in each increment towards this goal.

In the expression language semantic domain, we also provide additional structures to cope with inconsistent 'null' objects and the union of these with their concrete cousins. These are only necessary for the static semantics, which we provide in appendix C.

4.4.4.1 Simple type and function type static semantic object

Meta-variable or constant	Type name	Type definition
$\mathsf{Y}_{\perp}$	$\mathscr{Y}_{\perp}$	$\varnothing$
$Y_{\top}$	$\mathscr{Y}_{\top} \leftarrow \mathscr{U}_{\top}$	$\coprod \left\| \begin{array}{l} \textit{type} \\ \textit{type} \times \textit{type} \end{array} \right.$
$\mathbf{Y}_{\top\perp}$	$\mathfrak{Y}_{\top\perp}$	$\big(\mathscr{Y}_{\top}(U_{\top})\big) \sqcup \mathscr{Y}_{\perp}$

The first inhabitant of the semantic domain is the expression language's elaborated type system. This aggregates the syntactic definition of simple types and function types within a co-product. The expression language type system is first order, in which functional 'types' are an external construction over the raw type system. There exists for each such type a minimal set of type identifiers which define the environments within which the type is well-formed.

4.4.4.2 Variable and function type environment static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{R}_\bot$	$\mathscr{R}_\bot$	$\varnothing$
$R_\top$	$\mathscr{R}_\top \leftarrow \mathscr{U}_\top$	$\prod \left\lvert \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{Y}_\top(U_\top) \end{array} \right.$
$\mathbf{R}_{\top\bot}$	$\mathfrak{R}_{\top\bot}$	$\big(\mathscr{R}_\top(U_\top)\big) \sqcup \mathscr{R}_\bot$

The variable and function type environment specifies which identifiers are bound to the declarations of types and functions in the environment, and the types that are associated with each such identifier.

4.4.4.3 Expression static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{K}_\bot$	$\mathscr{K}_\bot$	$\varnothing$
$K_\top$	$\mathscr{K}_\top \leftarrow \mathscr{R}_\top \leftarrow \mathscr{Y}_\top \leftarrow \mathscr{U}_\top$	$\coprod \left\lvert \begin{array}{l} \mathscr{Q}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \\ \mathscr{C}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \\ \prod \left\lvert \begin{array}{l} v_{varid} \\ U'_\top \\ \mathscr{T}_\top(U'_\top) \\ \mathscr{K}_\top\left(U'_\top, \frac{R_\top}{U'_\top}, \frac{\mathscr{Y}'_\top}{U'_\top}\left(\frac{T_\top}{U'_\top}\right)\right) \end{array} \right. \end{array} \right.$
$\mathbf{K}_{\top\bot}$	$\mathfrak{K}_{\top\bot}$	$\left(\mathscr{K}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right)\right) \sqcup \mathscr{K}_\bot$

The expression static semantic object defines expressions that can be reduced to data, given a definition closure that is consistent with its environment. An expression decomposes into three different kinds of object: patterns, constructors and function applications.

4.4.4.4 Pattern static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{Q}_\bot$	$\mathscr{Q}_\bot$	$\varnothing$
$Q_\top$	$\mathscr{Q}_\top \leftarrow \mathscr{R}_\top \leftarrow \mathscr{Y}_\top \leftarrow \mathscr{U}_\top$	$\mathit{varid} \times \mathscr{J}_\top$
$\mathbf{Q}_{\top\bot}$	$\mathfrak{Q}_{\top\bot}$	$\mathscr{Q}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \sqcup \mathscr{Q}_\bot$

The pattern object consists of a variable identifier which refers to a variable in the current execution environment closure, and a list of pattern elements that recurses through the structure of the data object in order to reference the wanted component.

4.4.4.5 Pattern list static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{J}_\bot$	$\mathscr{J}_\bot$	$\varnothing$
$J_\top$	$\mathscr{J}_\top \leftarrow \mathscr{Y}_\top \leftarrow \mathscr{U}_\top$	$\prod_{i=0}^{n}\left(\coprod \left\vert \begin{array}{l} \mathit{varid} \\ \mathbb{N} \end{array}\right.\right)_i$
$\mathbf{J}_{\top\bot}$	$\mathfrak{J}_{\top\bot}$	$\mathscr{J}_\top \sqcup \mathscr{J}_\bot$

The pattern element is a co-product of either a variable name (for dereferencing a record parameter) or a natural number (for dereferencing a tuple element by positional parameter).

4.4.4.6 Constructor static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{C}_\bot$	$\mathscr{C}_\bot$	$\varnothing$
$C_\top$	$\mathscr{C}_\top \leftarrow \mathscr{R}_\top \leftarrow \mathscr{Y}_\top \leftarrow \mathscr{U}_\top$	$\coprod \left\vert \begin{array}{l} \mathit{boolconst} \\ \prod_{i=0}^{n}\left(\mathscr{K}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right)\right)_i \\ \prod \left\vert \begin{array}{l} v \in \mathscr{P}(\mathit{varid}) : \\ v \rightarrow \mathscr{K}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \end{array}\right. \end{array}\right.$
$\mathbf{C}_{\top\bot}$	$\mathfrak{C}_{\top\bot}$	$\mathscr{C}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \sqcup \mathscr{C}_\bot$

The constructor object has three branches: it can construct a new object out of a base type, a tuple of expressions or an identifier-indexed map of expressions.

4.4.4.7 Variable and function definition static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{Z}_\perp$	$\mathscr{Z}_\perp$	$\varnothing$
$Z_\top$	$\mathscr{Z}_\top \leftarrow \mathscr{R}_\top \leftarrow \mathscr{Y}_\top \leftarrow \mathscr{U}_\top$	$\coprod \left\| \begin{array}{l} \prod \left\| \begin{array}{l} \mathscr{T}_\top(U_\top) \\ \mathscr{D}_\top\left(U_\top, \frac{T_\top}{U_\top}\right) \end{array}\right. \\ \prod \left\| \mathscr{K}_\top\left(U_\top, \frac{R_\top}{U_\top}, \frac{Y_\top}{U_\top}\right)\right. \\ \prod \left\| \begin{array}{l} v_{varid} \\ U'_\top \\ \mathscr{T}_\top(U'_\top) \end{array}\right. \\ \prod \left\| \begin{array}{l} v_{varid} \\ U'_\top \\ \mathscr{T}_\top(U'_\top) \\ \mathscr{K}_\top\left(U'_\top, \frac{R_\top}{U'_\top}, \frac{\mathscr{Y}'_\top}{U'_\top}\left(\frac{T_\top}{U'_\top}\right)\right) \end{array}\right. \end{array}\right.$
$\mathbf{Z}_{\top\perp}$	$\mathfrak{Z}_{\top\perp}$	$\mathscr{Z}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \sqcup \mathscr{Z}_\perp$

This object defines variables (whether reduced already or not) and functions (built-in and user-defined). The dependent arguments express the requirement for the definitions to be well-formed in their environment. These arguments also affect the meaning of the definition, where it references values and functions defined outside its own scope.

4.4.4.8 Variable and function environment static semantic object

Meta-variable or constant	Type name	Type definition
$\mathtt{W}_\perp$	$\mathscr{W}_\perp$	$\varnothing$
$W_\top$	$\mathscr{W}_\top \leftarrow \mathscr{R}_\top \leftarrow \mathscr{U}_\top$	$\prod \left\| \begin{array}{l} v \in \mathscr{P}(varid) : \\ v \to \mathscr{Z}_\top\left(U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top}\right) \end{array}\right.$
$\mathbf{W}_{\top\perp}$	$\mathfrak{W}_{\top\perp}$	$\mathscr{W}_\top\left(U_\top, \frac{R_\top}{U_\top}\right) \sqcup \mathscr{W}_\perp$

This environment is an associative array of definitions, indexed by variable identifiers. To be well-formed, each member of this structure must match a structure in the declaration map of the dependent arguments. If this object is subscripted ‘Cmplt’, every such declaration is defined; if subscripted ‘Clos’, it is a closure, composed of a stack of variable and function environments.

4.5 Further detail

Please see appendix C.1 for the concrete syntax of HBCL and appendix C.2 for the expression language static semantics: this adequately demonstrates our points about how we choose to approach static semantics. Since we do not have a compiler, to design static semantics for the coordination language at this stage would be an otiose exercise, and they are therefore omitted.

4.6 Reference interpreter

The structural operational semantics given in appendix C.3 give a direct specification of a reference interpreter, which can be constructed by rendering each rule as a function in a proof assistant or functional programming language. The entry point to the interpreter is the FIFO execution step of the super-step execution cycle. The conclusion of rules are pattern-matched by these functions, and call the functions corresponding to the rules invoked in the premises. In the rest of this thesis, we first translate these semantic rules into functions of a proof assistant, and then use the code extraction facilities of the proof assistant to export executable functional code, which we compile. Finally, we use the resulting reference interpreter to run our example programs, so that we can evaluate the results and draw conclusions, before demonstrating the scalability of the system with a much more involved scenario.

4.7 Soundness, completeness and bisimilarity properties

Soundness and completeness generally refer to establishing static truths. For logics, this involves quantification over valid arguments of the logic; for programming languages, these properties quantify over valid input programs, or over time. In the case of a logic, we want to know whether any argument that follows the rules of that logic is true (sound) and whether all truths can be proved (complete). In the case of a programming language, we are interested in whether its static semantics admit only programs that evolve in ways permitted by the dynamic semantic rules (sound), and whether all such valid programs are admitted by the static semantics (complete). The two questions are related, where 'truth' in the case of logic is exchanged for producing absurd states that are not addressed by the dynamic semantics. In the case of languages in which the application of dynamic semantic rules involves a non-deterministic choice, such as where there is a choice of reduction orders, one can also choose to interpret soundness as meaning that any set of non-deterministic choices results in the same end state (assuming that the process halts) and completeness as referring to whether the language is Turing-complete or not.

This raises two questions: the first is concerned with obtaining a more precise definition of these prohibited absurd states of the dynamic semantics; the second asks what the relationship is between this static idea of soundness and completeness on the one hand, and the dynamic one, on the other. We will address these in turn, as in each case a discussion of the issues leads immediately to seeing how we axiomatize the relevant properties with our formalization of HBCL and set about demonstrating them. In each case, the problems can be — and often are — phrased in terms of type-checking, although this is not the only way of setting up the problem. The issues can also be thought of more generally in terms of accepting automata, as described in section 2.1.

4.7.1 Soundness

What are these absurd states that we are trying to avoid? They include such things as the dynamic semantics looking up missing or mis-typed definitions in an environment, or relying on a property of a program that does not turn out to hold true. In general, this is a difficult but uninteresting task, and in our approach to formalizing HBCL, we use a technique of deeply embedding our language in a logic that makes it easier: we derive soundness directly from the soundness of the formalizing logic, which in our case is the Calculus of (Co-)inductive Constructions implemented by Coq. We define a run-time semantic domain that is built using σ-types, and which therefore contains propositions. We employ a method using dependent typing in which an encoding of an HBCL type parametrizes a Coq type to form another Coq type. This final type has an underlying data structure that is restricted by the parametrized predicate of a parametrized σ-type to take values that are within the HBCL type. It is impossible *by construction* to build an absurd state of the kind that we are trying to avoid by demonstrating soundness. The compiled static semantic object is a Curried argument to an interpreter function, which produces a native function in the logic that can only build objects of the dynamic semantic domain that are correct by construction. If we discharge all admitted lemmas necessary to make this work, then we can prove soundness of the language, up to the strength of our belief that Coq is a sound logic (that is, it only allows us to prove truths). This approach works because we insist that the static semantic object can only be built if it is annotated with cost information that proves termination in a particular instance, so the fact that the logic must terminate, but the language, in the *general* case does not, does not generate an impossibility.

Static soundess is therefore shown by writing the semantic rules over a predicate-carrying ‘static semantic object’. We proceed by writing down each rule using a predicate relation over environment, implicant and implicand, and can show consistency by finding an executable function that witnesses its computability. The computablility of the top-level relation, which relates the abstract syntax tree to the static semantic object, follows inductively from all of the subsidiary rules, which all have function analogues. The question of what this function does if an invalid program is detected is dealt with below when we discuss completeness. In appendix B, we develop a style of giving semantic rules that allows predicate terms, but which is more generic than a specific logic such as Coq. However, precisely because it is generic and not actually a logic in itself, it does not conclusively prove anything on its own, which is why we go on to formalize the same structures in Coq.

Dynamic soundness follows directly from the fact that our interpreter function is written in the primitive-recursive language of Coq, and we take on trust that Coq is sound. Coq and constructive logics of its kind are special cases among proof tools, because its embodiment of the Curry-Howard isomorphism and type system that encompasses both concrete and propositional types unifies logical deductions and concrete execution into the same semantics. If we regard the predicate characterization of the semantics as canonical, then we must also prove that the predicate is injective if we want to be sure that any function inhabiting the type is sound, in that it produces the same answer for the same input arguments.

In summary, then, we can only show soundness *up to* the soundness of the formalizing logic. This might at first seem a sleight of hand: we do not anywhere explicitly axiomatize soundness as some proposition that we seek to prove. It is not a sleight of hand; since we are relying on Coq's own idea of soundness to demonstrate soundness of HBCL, we would have to axiomatize Coq within Coq to obtain a semantic handle on soundness in this context, and Tarski and Gödel prevent us from doing so without introducing extra logical axioms. It is reasonable for present purposes to take Coq as being a sound system, since the Coq system is generally accepted to be sound, and to enquire whether it acutally *is* sound beyond this would take us into philosophical questions that we outlined in chapter 2, the further pursuit of which is outside the purview of this thesis.

4.7.2 Completeness

Hbcl is at first sight statically complete because we insist that any box language in HBCL must come with enough annotations to construct a proof of termination. How, then, can HBCL be Turing-complete? The answer is that the proof of termination we have discussed is the proof of termination of a *slice* of an execution. A box that can suspend its state (trivially possible through the use of a feedback FIFO), can continue a calculation indefinitely, and need never halt, as HBCL state traces are coinductive structures. Hbcl can thus be made Turing-complete by using a box language whose semantics are unaware of the suspension of execution into time slices, and where that particular box language is

Turing-complete. If such Turing-complete *box languages* are given semantic access to an infinite temporal modality, they cannot themselves be *statically* complete, as that would imply a solution to the halting problem. Hbcl is therefore statically complete *but for* any box languages that axiomatize infinite time and are Turing-complete. Dynamically, hbcl is Turing-complete in that it can axiomatize nand gates in a simple expression language, and thus function as a complete hardware description language: but it can only *simulate* the executions of Turing complete languages when exported from a tool like Coq, because the cofixpoints needed to do so are not *executable* inside Coq: if they were, Coq would have to be logically unsound.

If, then, hbcl is reduced to a degenerate case in which the only allowed box language is one dealing with nand expressions — a hardware description language — this version of hbcl does not fall foul of the 'but for' condition above, so how can it be both statically sound and complete? The issue here is the semantic distinction between hardware description and ordinary programming languages. The key difference between a hardware description language and an ordinary programming language is that a hardware description language does not contain arbitrary functions or return results asynchronously; rather, it describes (possibily infinite) system histories. The question of whether it terminates or not is therefore not a question that makes sense: it is *axiomatized* through a coinductive constructor to have the possibility never to terminate, and as such is capable of simulating Turing-complete programs that may never terminate either. As we hinted in chapter 2, this makes us prefer a definition of dynamic completeness that is axiomatized *physically*, in the spirit of Gandy and Sieg. If we follow this line of thought, we might say that a physically complete language can be viewed as one that can emulate any (deterministic) part of physics that we choose up to some limit of discretization; we gain access to such arguments for the present work through the observation that hbcl can model arbitrary systems of clocked nand gates, what we know about computer architecture, the Church-Turing conjecture, and the kinds of physical simulations that can be constructed using computers.

4.7.3 Bisimilarity

Most interesting dynamic properties can be shown by axiomatizing them, defining a bisimilarity predicate, and then showing that the language produces only executions that match the axiomatization, according to the bisimilarity predicate. This includes application-specific properties whose predicates reference an embedding of some other problem domain in a common logic. To take an example that is reflexive in hbcl, we often want to know that two hbcl programs are bisimilar. In the case of the parallel composition example, we want to know that the traces of two example programs on their own map exactly to the combined trace of the composition. The same goes for the replication example, except that in this case, we may be able to show under an interpretation function that the two are equivalent, notwithstanding that one or more replicas may diverge from the trace that is 'correct'. Timing properties may also be displaced in a systematic way, in a discretized version (up to some rational-numbered frequency) of our rubber sheet analogy. The replication example again shows this in action, and we revisit what bisimilarity proofs in these cases look like in the context of the traces presented in chapter 6.

4.8 Summary

We have now seen how HBCL can be described by a predicate-loaded semantic domain. We have also formalized the semantics in a structural operational semantic style in appendix C.3, providing an inhabitant of the type of interpreters, existential proof of which is needed by the semantic domain in order to establish soundness. In chapter 5, we will remove the remaining semantic ambiguities by embedding HBCL in a fully formalized theory. In the constructive logic of Coq, the result is particularly convenient, since the type hierarchy of Coq enables us automatically to extract an interpreter from the formalization of the operational semantics, discarding proof terms. In Isabelle, we would have to work directly with predicates, which makes extraction a little more difficult, although plenty of tools exist to assist with this process. We have also discussed the formal properties that HBCL should exhibit and how they are demonstrated, and explained how bisimilarity predicates over different HBCL programs can be constructed. We extend this in the next chapter, in which, after presenting our formalization in Coq, we address how arbitrary properties may be specified over an HBCL signature in a suitably general logic.

Chapter 5

Semantics and interpreter in Coq

Throughout this thesis, we have been developing our exposition of HBCL with increasing degrees of formality. In chapter 3, we described the language intuitively and informally. In chapter 4, we presented the language in terms of a predicate-loaded semantic domain; we detailed the operational semantic rules in appendix C.3. The use of predicates and dependent types was, however, informal, in that we did not give enough information to fully axiomatize the logic, or to check the soundness of the semantics mechanistically. We now substantially close this remaining informality by rendering HBCL in the logic of a *proof assistant*. We thus attempt to remove informality and its concomitant uncertainty to the maximum practical extent with the deductive reasoning tools that are currently available. A simple example showing how the notations used in chapter 4 and appendix C.3 relate to the kind of rendering in Coq we see in this chapter can be found in appendix B.

There is, with some variations, a correspondence between Coq modules and language modules. In section 3.3 and Figure A.3 we gave a description of the structure of HBCL, which is mirrored in the Coq modules. In this chapter, in section 5.4, we give an outline of each of these statically parametrized parts of the language: they collectively set up the basic data structures needed by the coordination language, including the objects that define the *types* of harmonic and untimed box languages. We then go on in section 5.5 to describe the coordination language module functor (which is designed to be instantiated by the static module parameters), and explain how the semantic rules given in appendix C.3 are rendered in Coq. Finally, in section 5.6, we discuss our example expression language.

5.1 Choice of embedding for HBCL

The raw structures over which logical formulæ and sentences range have no meaning other than our intuitive grasp of them as rules for building objects, from which may arise the notion of *types*. It is this meaninglessness that enables such structures to be checked by inanimate machines, slavishly testing that structures are well-formed, that formulæ are grammatically correct and that rules of inference are legitimately applied. In formalizing a language like HBCL, we therefore make a number of choices, picking a basic logic, a spatio-temporal ontology, a style of semantic representation, and a way of constructing an interpreter or simulator. None of these choices is canonical, but we think there are good reasons for all of them, which we now describe.

If we espouse a structuralist philosophy, our ideal response to this lack of canonicity should be to use many axiomatizations, not only of our ontology and language, but of the logics within which we reason about them. If we accept the grand challenge of J. Strother Moore, it is surely an inevitable corollary that multiple axiomatizations must be the lynchpin of verified and verifying systems in heterogeneous logics. An abundance of structure morphisms would allow us to know that every one is talking about the same system, amounting to a kind of super-equivalence class by human labelling.

5.1.1 Choice of logic and proof assistant

We choose the Calculus of (Co-)inductive Constructions [56, 87, 152] and version 8.3 of the Coq proof assistant [68] as our logic and proof assistant respectively.

The choice is first narrowed by our previous decision to adopt an intuitionistic and dependently typed logic in the way we specified the semantics. Why use an intuitionistic logic? Intuitionistic logics work well with the verification of functional programs: the Curry-Howard isomorphism means that concrete functions can be shadowed by propositional counterparts that construct proofs for everything that the program can construct. Executable programs do not have a computational equivalent of an excluded middle, and so intuitionistic logic's apparent lack of power in this regard is not a problem. The ability of a unified type system that includes propositional types also makes it possible to use σ-types, which carry proof terms with the concrete types: this is very useful in giving a type-theoretic characterization to compiled objects and executions where semantically inadmissible structures are not members of the relevant type. The extraction mechanism from this kind of logic is straightforward: the propositional components of these types are stripped away leaving correct code.

Having decided to use an intuitionistic, or constructive, logic, Coq is chosen because it is the most well-developed and widely understood tool available in this paradigm.

5.1.2 Choice of ontology and temporal logic

A coinductive type dependently typed on an axiomatization of time gives us temporal modality, and axiomatization of object identifiers gives us unique designation. These are the only concepts in reality to which HBCL needs to make reference, and so it is attractive not to use unnecessary layers of embedded modal logics. There is only *one* physical reality,[1] and labelling it consistently is vital if the strong specifications we wish to produce of physical systems are going to be useful in the physical world.

5.1.3 Choice of semantic representation

The deep embedding and type-theoretical semantics we have discussed in this chapter accord with the separation of Pre-HBCL from full HBCL. By specifying nothing about computation in Pre-HBCL, we allow the formalizing logic to access directly, and quantify over, the ontology of HBCL programs. This leads to the possibility of reasoning about an arbitrary application domain in a proof assistant, treating Pre-HBCL as if it were a header file in a programming language. An application domain predicate derived from such a process may then be used to constrain the behaviour of an HBCL program, where the logic in which the predicate is stated is the common denominator. This behaviour in such a predicate characterization can be specified completely independently of the full HBCL function that implements it. Indeed, any other language that satisfied the ontological signature with hard bounds on execution resources would do equally well, as long as it, as interpreted by its semantics, satisfied the predicate of the application.

In summary, we have an ontologically sound means of referring to the subject of a specification: it is this that facilitates the comprehensive decoupling of specification and implementation. Either can be freely mixed using suitable morphisms. As we began to advance in chapter 2, this, with its potential to use multiple axiomatizations in multiple logics, is a much more flexible method than using monolithic languages for programming, specification and implementation.

5.2 Specification in Coq: methods and constraints

Our workflow with Coq is first to formalize an abstract sytnax tree, followed by a semantic domain and then the operational semantics. The operational semantics are rendered as functions, which implies that the top-level function should be an executable interpreter for the the language being formalized. We refer to this interpreter in what follows as the *reference* interpreter. It is not an interpreter that has an efficient execution time for a given program, but it is ideal for prototyping a language, because its operation mirrors the operational semantics extremely closely. Optimized interpreters with bisimilar behaviour could be added later. Given that HBCL boxes are designed to be realized on parallel hardware, and to stress that the monolithic interpreter is not the preferred run-time environment, we sometimes prefer to call the interpreter a *simulator*.

We do not execute our interpreter within Coq, but use the extraction facilities to export an OCaml program, which allows us to use the OCaml debugger. In the presence

[1]This is itself an axiom, but not a controversial one, unless we espouse the most exotic of cosmological theories.

of bugs and admitted lemmas, a function in Coq cannot be guaranteed to do what we expect, or even to terminate, but there is no mechanism to extract debugging information. OCaml [162] is a multi-paradigm language supporting functional programming. It has a long history, but the only relevant part of that history here is that recent versions of Coq are written in OCaml, and many of its standard libraries mirror corresponding OCaml structures. Both Coq and OCaml are flagship projects of INRIA.[2] OCaml is the best-supported language for code export in Coq, although there is still some mismatch between the type systems and module semantics of Coq and OCaml, which cause us some problems in chapter 6. When code is exported from Coq, proof terms are stripped away. Coq's type system is subdivided into 'Sorts'. The `Set` Sort is that of concrete objects, of the kind that can be exported into an ordinary programming language such as OCaml. The `Prop` Sort is the type of logical propositions, which are discarded when code is exported. There is also an automatically managed tower of '`Type` universes', in which are defined the types of more complex objects that may contain predicates over objects in `Set` or other `Type` Sorts. This unification of concrete and propositional types within a single system is powerful. Proofs are functions like any others, except that they belong in the impredicative `Prop` Sort. This is the Curry-Howard isomorphism in action [172].

OCaml is ordinarily a strictly evaluated language, meaning that sub-expressions are evaluated as soon as they appear in the execution path. However, OCaml also provides a construction for 'lazy' evaluation, where sub-expressions are only evaluated when their parent expressions are dereferenced in the execution path. This structure is needed when Coq exports cofixpoints, which are functions that may execute indefinitely. If the 'lazy' evaluation mechanism did not exist, the use of cofixpoints would cause non-terminating computations.

5.2.1 Termination, well-foundedness and consistency

Turing-complete languages enable one to write non-terminating functions. This can be a problem in logical systems: if it were possible to write down recursive functions that never terminated without providing some proof that this could not happen, the logic would be inconsistent. Different logics approach this problem in different ways, but Coq avoids the problem by making its ability to express computation Turing-*in*complete. In particular, recursive functions are required to be primitive recursive. Fixpoints are required to have a parameter which is structurally smaller on each recursive call. This is achieved by forming the recursive argument from within a match on a constructor. Often, Coq can determine this parameter itself, but sometimes it cannot, and so we occasionally give it explicitly, using the `{struct a}` notation. This tells Coq that on each

[2]Institut National de Recherche en Informatique et en Automatique, France.

call of the fixpoint, each '`a`' is structurally smaller than it was on the last invocation. This is known as a syntactic guard condition.

The primitive recursive restriction can be a problem in expressing certain functions, where there is no obvious structural recursion, but yet we know the function must terminate. The answer to these problems lies in an *accessibility predicate*. Rather than a fixpoint reducing structurally on a concrete argument, it can reduce structurally in the size of a proof object. This is done by establishing a total order on the object involved in each reduction step, instantiating an accessibility predicate `Acc` on this total order, and flagging this up as the reducing argument to the fixpoint in question using the `struct` keyword.[3] Using this technique, we can simulate Turing-complete languages within Coq, but only for those inhabitants of such languages that come with a termination proof.

5.2.2 Key Coq programming paradigms

5.2.2.1 Maps, records or associative arrays

The terms 'map', 'record' and 'associative array' are synonyms, and we use them interchangeably. In our propositional calculus example, our environment of previously inferred formulæ was implemented as a list. The formulæ were anonymous, which meant that the prover algorithm had to perform a monotonous search to find appropriate inferences. In a programming language, we need to bind definitions to identifiers, which requires that we use associative arrays.

The associative arrays needed to implement records and environment types are not a native part of Coq, but are axiomatized and implemented in libraries. As the inductive structure of an associative array implementation is not canonical, most relations we use in proofs have to be proved as morphisms over equivalence classes of associative array structures. Since the definition of an associative array admits an arbitrary number of implementations, the standard Coq `Module Type` of a map hides the structure of implementations. This causes us problems with strict positivity and module structure, which we review below.

5.2.2.2 Strict positivity

Coq has a strict positivity requirement when defining inductive types, which constrains our design choices. In the definition of inductive types, or a set of mutually inductive types, recursive references to the types being defined must be 'strictly positive'. In practice, this allows us to place such references anywhere in a definition except as dependent parameters to one of the types, or as a parameter to an object which itself forms a depen-

[3]Coq provides some extra syntactic sugar to help with this, though we prefer not to use it, as it can obscure what is going on, and can involve constructing unnecessary record arguments, since some of these features require the fixpoint to have only one argument.

dent parameter to the type we are defining: a type cannot parametrize itself. If it could, Coq's logic would be inconsistent. We *can* pass a reference to the type being inductively defined to parametrize a reference to another *inductive* type, but not to a function type, because Coq tacitly unwraps the definition of this referenced type into the current definition. This unwrapping is only possible with inductive types. Maps are important structures that are parametrized in the type of their elements. An important effect of this is that, where we want to include a map of the type being defined within its own definition, we have to use the concrete implementation of a map, rather than its functional characterization from a module parameter, so that Coq's type-checker will tacitly inline it. Another more general implication of the strict positivity requirement is that we cannot reference a σ-type of a particular proposition from within the definition of that proposition. This necessitates that we adopt a design pattern in which we keep the propositional and concrete components of inductive types separate.

5.2.2.3 Module design in Coq

There are two basic classes of object in Coq's module system: `Modules` and `Module Types`. Any object can be defined in both a `Module` and a `Module Type`, but any object that is defined in a `Module` that is declared to be of a particular `Module Type` must define *every* object in that `Module Type`. If the object was *defined* in the `Module Type`, then the definition in the `Module` must be exactly the same as that in the `Module Type`. More usefully, `Module Types` can declare, but *not* define `Parameters` and `Axioms`, which are conventionally in the Coq 'Sorts' of concrete types and propositions respectively, although Coq treats the keywords synonymously. In addition, `Modules` and `Module Types` can extend other `Modules` and `Module Types`. This amounts to a kind of subtyping, in which subtypes may add more declarations or definitions to their parent types. This subtyping is not true subtyping, being a syntactic rather than a semantic feature. Module functors allow modules to be parametric in the types of other modules.

The abstraction involved in a `Module Type` is very similar to the interface abstractions of many ordinary programming languages, where a `Module Type` may look similar to a Java Interface or Abstract Class, or a C header file. However, there are also some crucial differences, given that we are operating in a proof assistant's type system that also includes truth-valued semantics. As even types have types in Coq's rich system of dependent types, we can do things that are not possible in ordinary languages, even those with type variables or 'generic types'.[4]

The paradigm used by Coq's Standard Library utilities usually entails defining a `Module Type` with a parameter `t` of type `Type`, which requires an implementing module

[4]Although Brady's new language, Idris [36], is a programming language that does have dependent types.

to provide a *concrete* type.[5] The user of the library then instantiates objects of *this* type. An interface of the kind familiar from ordinary programming languages is then given in terms of function types, and axioms are stated about how the structure that is the main subject of the module is affected by each interface function, and the behaviour of those functions. This gives rise to the informal use of the term 'module signature' to refer to a `Module Type`. These may make use of inductive types in the `Prop` sort. Any complete implementation is then expected to provide proofs in terms of the structure and function implementations that satisfy the axioms over the concrete types. Usually there is only one main concrete type per `Module`, but this is not always the case.

This paradigm is elegant, but there are two details which cause us problems, owing to our use of nested associative arrays. The main type of a module is built from a raw concrete inductive type and a predicate. However, this substructure is hidden and only the resulting composite type is exposed by the Standard Library `Module Types`. This has the consequence that we cannot separate out the predicate and incorporate it into the further predicate of an object that instantiates one of these Standard Library types, without destroying the `Module Type` abstraction. If we use a Standard Library map (associative array) in a mutually nested way, we also fall foul of Coq's strict positivity requirement, since the abstract type of this unified object is a function. We would like to recast the `Module Type` interface of the Standard Library to separate out the concrete inductive type from its predicate, but this is not possible either, because Coq lacks the syntax and semantics necessary to declare a parameter as being restricted to an inductive type.

Listing 5.1: The raw coordination object

```
Inductive CoordStateRaw : Type :=
| CoordStateRaw_make :
    InMemModBox.MDatTimeMapRaw → OutMemModBox.MDatTimeMapRaw
    →
    LInstMapModRaw.t ( CoordStateRaw) → CoordStateRaw.
```

Extracting a definition from the next chapter, we can see how the nested map problem arises in the type of the raw coordination state in Listing 5.1 (the raw map is only visible because we have instantiated the Standard Library module functor with a specific implementation and have used the <: operator to make the implementation visible beneath

[5]Since `Types` can have arguments in `Type`, Coq differentiates these types into ordered type universes: this information is not usually available to the Coq user. It can cause problems when Coq cannot work out how to assign a non-circular ordering to types, which typically occurs in developments with complex dependencies and multiple `Type` arguments. As well as the `Type` 'Sort', Coq also has predicative `Set` and impredicative `Prop` Sorts. The `Set` Sort is suitable for specifying inductive types without dependent arguments. Our extensive use of σ-types means that Coq must maintain a very large graph of type universes. The lack of explicit type universe polymorphism [174] in Coq has led to a situation (which we discuss in section 6.1 and section 6.9) in which Coq cannot unify the type parameters of our assembled module functors, so that full module assembly has to wait until after code is exported into OCaml.

its `Module Type` interface). The map `LInstMapModRaw.t`, which has been instantiated in the type of its keys by a module functor, requires the argument `CoordStateRaw` to give the type of its elements. However, `CoordStateRaw` is the very type we are defining, so we are close to falling foul of the strict positivity requirement if Coq cannot unfold the definition of the inductive map. We now explain in more detail why this is a problem.

The issue of predicate separation arises with almost any data structure in Coq more complex than a list, where some sort of predicate is necessary to check that the underlying data structure is well formed. For instance, the Standard Library implementation of the unordered map library (that we use repeatedly) is implemented as a list of pairs, and the predicate is necessary to ensure that there are no duplicate entries. With more sophisticated implementations, the predicates can become quite involved. When we embed the Standard Library in our nested structure, even putting aside the module problem, we cannot embed the full type with its predicate, because it becomes a parameter of itself, and violates Coq's strict positivity requirement.

The problem of defining a nested structure using the type of associative array implementations arises because this dependent type looks the same as a function, which can of course include arbitrary λ terms. However, if Coq knows that this self application is of a dependent *inductive* type — as opposed to a plain function — then Coq can see that there is no inconsistency, and the nesting is permitted. To maintain a proper separation of map interface from implementation, the interface of the Standard Library map would not only need to include separate treatment of the underlying type and predicate, but also allow type parameters in `Module Types` to be declared as inductive. Unfortunately, Coq and its Standard Library do not facilitate either of these things, so at present we are forced to make the underlying list implementation visible to our specification of a nested logical instance, which is not ideal.

The richness of the dependent type system and the semantics of module parameters, which allow early parameters to be given to later parameters, cause problems in code export, when the type system or module system in the target executable language have more restrictive semantics. We return to this subject in chapter 6, when we see that Coq's export facilities to OCaml result in verbose module signatures and some odd type casts.

5.2.2.4 Cofixpoints

The use of coinductive types and cofixpoints occurs only once in our formalization, where our use of them axiomatizes infinite time. Fixpoints, by contrast, are pervasive. Our single use of cofixpoints is nevertheless extremely important, because it allows us to quantify HBCL over executions of arbitrary length.

Cofixpoints are duals of fixpoints. Instead of recursing on ever-smaller objects, cofixpoints recurse on ever-*larger* objects. In order to ensure that definitions of cofixpoints respect this property, they have their own syntactic guard condition. It restricts recursive calls, so that self-invocations may only occur under a constructor of a coinductive type. Instances of coinductive types may be infinitely large, but may be finite if a base-case constructor of the coinductive type is provided. The combination of these two features can cause problems when we know that a subset of a cofixpoint's possible invocations produces finite branches of the coinductive result type. When we translate the execution of nested HBCL instances into Coq, we would like to use the same super-step functions in the nested invocations as in the top-level, but we know from the operational semantics that nested invocations are only considered in short bursts: otherwise, they would starve for lack of input. We know that the structure of a nested instance will lead it always to produce a member of the finite subset of the coinductive type of execution traces, but we cannot use a cofixpoint for this, as there is no mechanism to convince Coq's type-checker that this particular case of deferred evaluation will actually terminate. The syntactic guard conditions of fixpoints and cofixpoints are mutually incompatible. Coq assumes that a cofixpoint is executed in a lazy evaluation style, and will not unfold the function from within a fixpoint environment, because, from the perspective of the calling fixpoint, it could produce a non-terminating execution which, if accepted by Coq, would undermine the strong normalization[6] and hence consistency of the logic.

The solution to this problem that we will use in the next chapter involves factoring out each branch of the mutally recursive fixpoint/cofixpoint, and instantiating it separately in fixpoint and cofixpoint forms. This allows us to avoid code duplication, but enables us to satisfy Coq's various syntactic guard conditions. Coq performs a kind of reduction in which it takes the separate definitions of the arms of the mutually recursive functions inline (Coq calls this δ-reduction) *before* type-checking the whole fixpoint or cofixpoint and the respective syntactic guard conditions.

5.2.2.5 Admitted lemmas in refinement proofs

These are lemmas that construct the necessary predicate to a σ-type, using information unpacked from predicates in the previous coordination object and inferences from the structure of the function. However, because they have been admitted, not all relevant truth terms have been passed to them, and so we know that in their present form they cannot be proved. The '`NOT_EOUGH_ARGS`' in the name of the lemma reminds us of this. That we have a lemma that we know is not true is not as frightening as it sounds, because every time we extract the code, Coq reminds us of axioms that have not been proven, including this one. This programming paradigm is necessary when our code carries proof, as without it, it would be impossible to extract a working prototype before the

[6]Term reductions always terminate, and terminate with the same result.

entire code base had been proven correct. We consider it is better to have placeholder lemmas of the ‘`NOT_EOUGH_ARGS`’ than to rely on side lemmas to show correctness — lemmas that would be easier to forget about when hardening the system for a production environment by removing all propositional axioms. Some functions would not even type-check if we relied on side lemmas, because there would not be enough information in some functions’ inner scopes to convince Coq of well-founded recursion. Coq reminds us that we have admitted lemmas in a development when we try to export code: a textual search for ‘`NOT_EOUGH_ARGS`’ lemmas and the command ‘`Admitted`’ tells us where to find them. It is worth remarking that the `admit` tactic (the action of which is to introduce a suitable axiom) often does not work because the tactic gives it every possible argument it can find: when the context is within a fixpoint, this includes the one whose size is decreasing, which results in Coq detecting that the fixpoint guard condition has been violated. This problem remains undetected until the command ‘`Defined.`’ is entered, since the guard condition is only checked at the point where the type-checking kernel is invoked, not by the tactic code.

5.3 Link between operational semantics and Coq functions

The operational semantics as given in appendix C.3 are closely related to the Coq functions given in this chapter, in the same way that the Coq functions given for the propositional calculus example followed their semantic rule counterparts. We cross-reference the main rules in the text, and provide them in tabulated form in appendix D.2 and appendix D.3. The low level functions in Coq diverge from the way that the operational semantics are given. This is because the operational semantics in Coq must provide enough detail to be executable, whereas the objective of the semantic style of appendix C.3 was to provide an intelligable summary, which elided some details. In addition, the semantic style of appendix C.3 provides a functional form of associative arrays and higher order mapping functions. Such forms can only be implemented indirectly in Coq, which causes further divergence. Finally, the Coq formalization uses a number of extra function arguments for convenience and efficiency at run-time, rather than deducing them from the static semantic object: this diverges from the rules of appendix C.3, which deal with this information implicitly.

5.4 Static formalization parameters

There are two possible ways we could understand ‘static’ in this context. Given that we are dealing with Coq, which has a rich module semantics, the static parameters can be seen as instances of a type of module that are parameters to module functors. However, module semantics have no special status in the underlying logic of Coq, since modules can be syntactically unfolded into module-less Coq (albeit with unwieldy and repetitious structures). There is nothing about modules that cannot be expressed in suitable dependent types, which removes repetition both syntactically and semantically. Coq module functors are largely a software engineering convenience, and we use them because many of the standard libraries of Coq that originated in the Compcert project expect this style of programming. The second way of understanding 'static' parameters in our language is more natural: the formalization or means of formalizing deeply embedded type systems, identifier spaces, box languages and restrictions on temporal semantics are fixed across a particular version of HBCL. To change any of these means to change the version of HBCL.

This can be contrasted with dynamic parameters of the language, such as box languages, which are first order objects in HBCL. It so happens that the static parameters of HBCL have been matched with static module parameters in Coq in the present formalization, but the choice is essentially arbitrary: in a logic such as Isabelle/HOL, there *are* no module semantics, as Isabelle theories are not modules, and cannot be, owing to decoupling of logic and meta-logic in Isabelle — theories are contained at the meta-logic level.

5.4.1 Pre-HBCL, full HBCL, and the type of box languages

As we discussed above, the most general outer shell of HBCL is contained in Pre-HBCL, which is a collection of inputs and outputs, the relationships between which may be characterized by predicates. It provides the ontology of the environment: that environment provides input streams that completely determine the evolution of a particular HBCL instance. Full HBCL provides a mechanism to fully specify designated output memories in terms of those inputs over time, with reference to a single clock, canonically international atomic time (TAI). Full HBCL is not the only language that could adequately specify these relationships. However, it maintains a high level of generality by further stratifying itself into two more abstraction layers, as defined by harmonic box languages and untimed box languages. The nested case of HBCL can be viewed as a special recursive case of a harmonic box, but in order to produce a tractable mutual induction scheme in Coq, the nested case is treated outside the harmonic box framework. There is no reason why a completely different coordination language could not also be embedded in a harmonic box, given an appropriate type system mapping. Similarly, the Church-Turing thesis implies that we could also embed HBCL within another language.[7]

The untimed box language record corresponds with the abstract type of untimed box languages given in the semantic domain in section 4.4.3.6.

[7]Albeit in a sandboxed ontology, unless we provide an ontology of the enclosing coordination language and an ontological 'bridge' to that of the encapsulated HBCL instance.

Listing 5.2: The type of untimed box languages

```
Record UExprLang := {
  CTDT : Type;
  CTDTP : ProtoT → CTDT → Prop;
  costB : CostBase TypeS CTDT CTDTP DataR DataP;
  AST : Set;
  parse : Encoding → AST;
  sso : InpOutpTypes CTDT CTDTP → InpOutpTypes CTDT CTDTP →
    Type;
  compile (itypes otypes : InpOutpTypes CTDT CTDTP) :
    option (sso itypes otypes);
  reduce (itypes otypes : InpOutpTypes CTDT CTDTP) :
    sso itypes otypes →
    sig (UDataPSTMatchesInpOutpTypes CTDT CTDTP itypes) →
    sig (UDataPSTMatchesInpOutpTypes CTDT CTDTP otypes)
}.
```

CTDT and CTDTP are the Coq types of the data structure that parametrize the cost function for the expression language. CTDT is the concrete type and CTDTP is the predicate that constrains its structure to a meaningful subset. These fields are present for future use, since the cost function we use is, at present, flat. costB is the function that operates on this data. Again, in the implementation we provide, this is flat over the HBCL type-space. AST is the abstract syntax tree, which formalizes the syntactic types shown in section 4.2 as Coq Inductive types.

The interesting fields of Listing 5.2 are sso and reduce. sso stands for 'static semantic object', and has a type of the Type Sort, dependent in the fingerprint of the untimed OID types of its inputs and outputs, and statements of the computational potential of each. reduce is a function that executes a program encoded in the sso. When sso is absorbed as a Curried argument into reduce, we have a Coq function that directly implements the program in an arbitrary language. It is guaranteed to terminate since it is a primitive recursive function in the logic of Coq, but it can nonetheless express a set of *terminating inhabitants* of a Turing-complete language. The sso can only be successfully compiled (*i.e.*, compile gives its option argument in Some as opposed to None) if the particular program carries adequate proof of termination by appropriate cost annotations, which a compile function can use to generate proofs of well-founded recursion. The coordination language is oblivious to the choice of untimed box or expression language (there may not be one at all if the harmonic box directly implements a temporally aware expression language). As far as the formalized coordination language is concerned, the reduction function is an ordinary Coq function.

The compile field of UExprLang has the Coq type of a compilation function, turning an instance of the abstract syntax into an option type wrapping a static semantic object. We do not implement this function in this development, but compile examples by hand. The compilation paradigm is that of the proof-checker whose formalization is given in appendix B. The option modifying the sso type enables the compiler to return something (None) when the abstract syntax tree it is parsing disobeys the static semantic rules.

The Coq representation of the binding of harmonic box languages to untimed box languages is a synthesis corresponding to the abstract type of harmonic box languages given in the semantic domain in section 4.4.3.11 and section 4.4.3.12. This composite object is less flexible than the two separate objects, because it assumes all harmonic box languages will consist of bindings to untimed languages. Given that we only have one untimed box language, our example expression language, the contraction makes the implementation simpler.

Each harmonic box has a signature that connects some of the instance inputs to outputs (these may be nested and/or not exposed in the instance's interface). The idea is that an overall predicate for an instance ensures that there are no unconnected memories, or memories that are connected more than once. Each harmonic box contains a complete description of the relevant box language and the particular program in that box language. This is inefficient in an implementation, but it mirrors the semantics cleanly and could be optimized later. If exported to a pure functional language, the compiler language is likely to remove the redundancy in any case.

Listing 5.3: The type of harmonic box languages

```
Record HBoxAbs := {
  uexprlang : ubox.UExprLang;
  boxfreqcorrectin : Freq →
    ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop;
  boxfreqcorrectout : Freq →
    ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop;
  IOtypePredIn : ∀(f : Freq)
    (freqm : sig (boxfreqcorrectin f ))
    (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop;
  IOtypePredOut : ∀(f : Freq)
    (freqm : sig (boxfreqcorrectout f ))
    (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop;
  convertInp : ∀
    (f : Freq)(tf : TTime f)
    (memvarmap : InMemModBox.otm.t ipm.Varid)
    (ttypes : sig
      (fun tmap ⇒
        ∃ vm,
          InMapVaridConvertPred boxfreqcorrectin f memvarmap vm tmap))
    (utypes : InpOutpTypes _ _),
    InTypePredConvert uexprlang boxfreqcorrectin f (proj1_sig ttypes) utypes
    IOtypePredIn memvarmap →
    InMemModBox.MDatBoxTime MDataInst.ReadEnabled
    (proj1_sig ttypes) f tf →
    option (sig (ubox.UDataPSTMatchesInpOutpTypes _ _ utypes));
  convertOutp : ∀
    (f : Freq)(tf : TTime f)
    (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
    (ttypes : sig
      (fun tmap ⇒ ∃ vm,
        OutMapVaridConvertPred boxfreqcorrectout f varmemmap tmap vm))
    (utypes : InpOutpTypes _ (ubox.CTDTP uexprlang)),
    OutTypePredConvert uexprlang boxfreqcorrectout f (proj1_sig ttypes)
    utypes IOtypePredOut varmemmap →
    sig (ubox.UDataPSTMatchesInpOutpTypes _ _ utypes) →
    OutMemModBox.MDatBoxTime MDataInst.WriteEnabled (proj1_sig ttypes) f tf
}.
```

Listing 5.3 shows the harmonic box binding. This binding is a parameter to the static semantic object of a harmonic box, which is the object that is directly held by the coordination language and called to compute harmonic boxes. `uexprlang` is the expression language bound by this `HBoxAbs`. `boxfreqcorrectin` and `boxfreqcorrectout` are predicates that restrict the input and output memory maps respectively. For the specific binding of type `HBoxAbs` that we will go on to define, these predicates restrict the box frequency to an integral multiple of each of the memory frequencies in the map. This ensures that boxes obtain the same number of pieces of data on each time slice. The restriction is necessary because our type system does not have types of dynamically variable size. `IOtypePredIn` and `IOtypePredOut` define a relation over the Cartesian product of, in the first case, input memories and input data type to the untimed box, and in the second case, output data type from the untimed box and output memories. This defines the semantics of the functions that perform these conversions in predicate form. `convertInp` converts its argument from a map of data with harmonic OID types into a piece of data defined over untimed OID types, while `convertOutp` performs the inverse operation. The other fields are predicates that ensure that the conversion functions are not asked to convert incompatible types.

5.4.2 HBCL module design in Coq

The modular design of HBCL in Coq follows the structure of the language shown in Figure A.3, repeated for convenience in Figure 5.1. Following the opposite direction of the dependency arrows, progressive modules accumulate more and more arguments in the module types on which they depend. This design removes arbitrary choices about type system embeddings and identifiers from the coordination language specification and implementation, treating them instead as parameters with the *type* of identifier and type system embeddings. The module types do, however, specify how elements of the type system are to be implemented: types are to be sized, and the type of data of these sized types is intrinsically typed through Coq's dependent typing.

The workflow has usually involved writing concrete modules and then abstracting them into module types. This is an incomplete process, especially in the case of the harmonic box and coordination modules, and there are at present many concrete definitions in module types that could be designed out in any further work. We view the production of a working prototype interpreter as a priority, reflecting the *operational* semantics. If further work permits, more detail can be added to the structure of predicates in parametrized σ-types; the function types whose co-domains form these strong types might then be factored out into *module* types or records that contain no

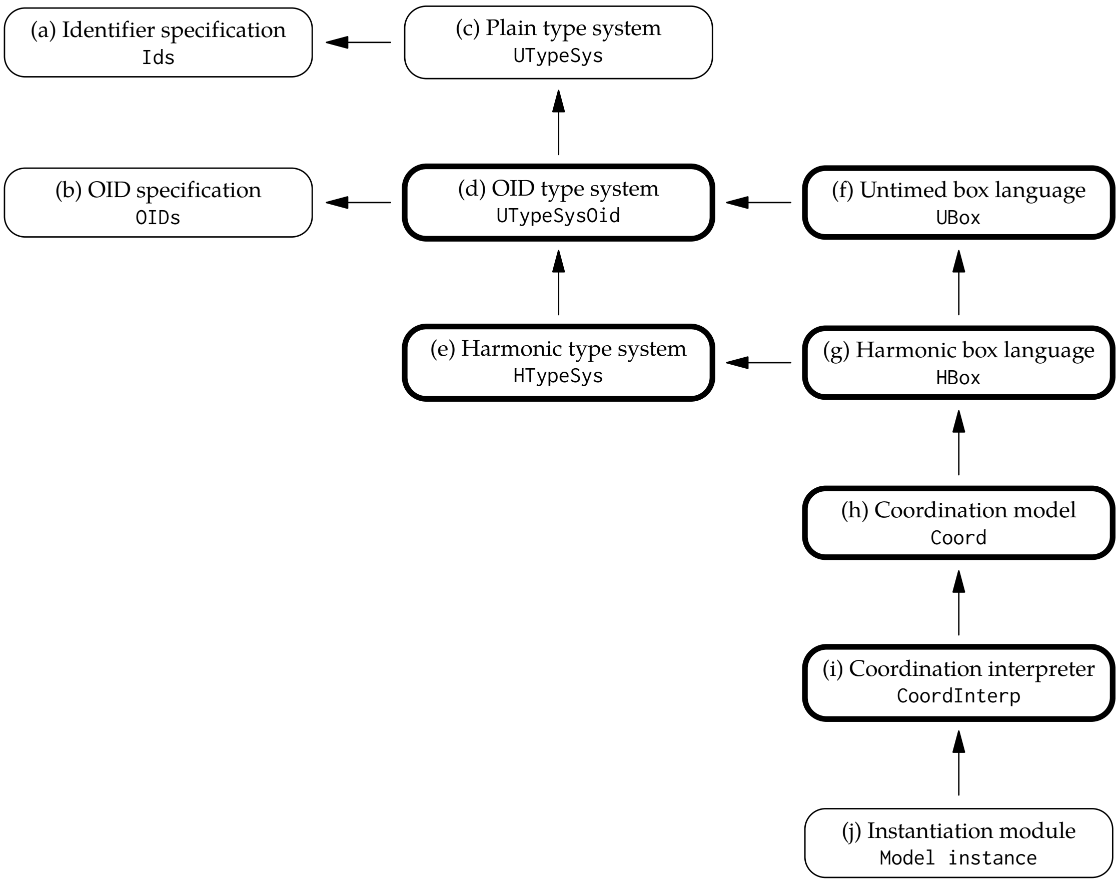


Figure 5.1: Structure of full HBCL

concrete functions. This would define the *type* of correct interpreters, dependent and thus parametrized in the arbitrary choices about identifiers and type systems in the foundational modules. In this way, all constructible interpreters would be constrained to be correct. These correspond to predicates of the semantic domain in section 4.4. It would also be much easier to reason about abstract properties of the language using such a predicate characterization than with reference to operational rules. However these rules, realized in functions, are nonetheless essential existential proof that the type of implementations is inhabited.

We give detailed listings of important definitions from the key concrete modules, along with selected module types in appendix D.

5.5 Formalization of the coordination language

This section gives the details of how the coordination language of full HBCL is implemented in Coq. First, we explain in detail how the super-step described in section 4.4.3.25 and given in more operational detail in appendix C.3 maps to the Coq code. We then present the code for the rest of the semantic rules involved in the coordination language, before presenting the semantics of the harmonic box binding.

Again, the full Coq sources take up a considerable amount of space, so we have given the main rules in appendix D.5, and illustrate our discussion here with much smaller gobbets.

The core of the coordination semantics of full HBCL is contained in the super-step. We now unpack the functions that implement each substep. Those mutually recursive functions referred to in each phase of this four-fold step are given as parameters to ordinary functions. This enables the same semantics to be instantiated in both fixpoint and cofixpoint forms for the nested and global levels of HBCL respectively.

5.5.1 Section variables

Each substep of the super-step takes some common parameters. These are set up as variables using Coq's sectioning mechanism. Within these sections these variables are taken as undefined parameters, which are elevated to parameters of every defined term that uses them on closure of the section. For this reason, these variables are declared once, but are accessible to all of the super-step substeps. They are given in Listing 5.4.

Listing 5.4: The `Section` variables

```
Variable f : Freq.
Variable linstsig : LInstSignature f.
Variable instTypeScopeMap : LInstMapMod.t LInstSignatureRaw.
Variable lissolib : LibClos instTypeScopeMap.
Variable lisso : Lisso f linstsig instTypeScopeMap lissolib.
```

The frequency with which the step is taken is given by ‘`f`’. ‘`linstsig`’ is an instance signature. It has an underlying raw inductive type beneath a σ-type definition, and presents the types of exposed input and output memories on the instance interface. The frequency qualification is a parameter to the inductive predicate of the underlying σ-type, which confirms that the frequency given is the lowest common multiple of the individual member frequencies. Aggregate frequencies are important in determining the resolution with which the instance must be stepped, since any activity occurring between each tick of the associated clock can be compressed into the state space transition of the instance signature frequency. `instTypeScopeMap` is a map of instance signatures indexed by instance identifiers. This is the set of instances to which the instance in question may refer, concrete instantiations of which must be provided in a closure object if the instance in question is itself to be fully specified without re-exporting the same dependencies. `lissolib` is that closure. Finally `lisso` is the instance itself, carrying as dependent arguments all of the objects we have just mentioned. This allows us to recruit Coq's type system to ensure that logical instances have the correct type and temporal properties for the uses to which they are to be assigned. The entry point to the interpreter is a Coq function to which this static semantic object is applied as a Curried argument, producing another function that takes an input stream and from it generates a trace of the system history.

5.5.2 FIFO step and entry point

Rule C.133, which implements FIFO execution, is realized in Coq in Listing D.18. The implementation in Coq departs slightly from the rule C.133, in that nested FIFOs are not processed until the next nested box step. We now examine this piece by piece. Listing 5.5 shows how `FIFOsStepCoordImpl` encapsulates a fixpoint and its accessibility predicate.

Listing 5.5: The shape of the FIFO step

```
Definition FIFOsStepCoordImpl(t : TTime f)
  (mti : InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig))))

  (cstate :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
    (‘ (InstSigOutputMems (‘linstsig))))
  :
  ((sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
    lisso)) × InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig))))%type.
refine (
  let fix FIFOsStepCoordImplInner(f′ : _)(t′ : TTime f′)(linstsig′ : _)
    (instTypeScopeMap′ : _)(lissolib′ : _)(lisso′ : _)
    (mti′ : InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig′))))
    (cstate′ :
      sig (CoordStateInnerFIFOsEnabled f′ t′ linstsig′ instTypeScopeMap′
        lissolib′ lisso′))(cstate′Acc : Acc CoordStateRecSizeLT (‘ (‘cstate′)))
    { struct cstate'Acc } :
    ((sig (CoordStateInnerMemFBEnabled f' t' linstsig' instTypeScopeMap'
       lissolib' lisso')) ×
    InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig)))
    )%type :=
[Inner fixpoint definition omitted]
in
    FIFOsStepCoordImplInner f t linstsig instTypeScopeMap lissolib
    lisso mti cstate _
).
Defined.
```

The arguments to `FIFOsStepCoordImpl` include the variables declared in Listing 5.4. ‘`t`’ is the present time slice. It is a natural number that counts ticks at the rate given by its dependent argument `f`. `mti` is a map of input memory states, satisfying the signature `linstsig`. The argument implied by the ‘`_`’ is the frequency of the input memory (`‘ (InstSigInputMems (‘linstsig)))`. `InstSigInputMems` is just a convenience to extract this from the signature.

`cstate` accommodates the object that holds the entire state space of the instance. It is a σ-type of a σ-type. The inner object is qualified by a predicate that ensures that it is statically well-formed. By this, we mean that every memory matches the static signature according to the memory identifiers, and the same for nested instances. The mapping between static and dynamic maps of this sort is strictly bijective. The temporal predicate in the outer σ-type, `CoordStateFIFOsEnabled`, makes the lists of time-stamped values in each memory compatible with each other and with the time `t`. This outer predicate is unique to each super-step substep. In the case of the FIFO step, the inner FIFO-box memories must be enabled for the current time slice. There is no mention of trace objects in rule C.133 or `FIFOsStepCoordImpl`. Trace objects are added in the super-step assembly phase, which is dealt with in rule C.128, rule C.129, rule C.130, rule C.131 and rule C.132, and in section 5.5.7. `mtoNest` is a convenience, containing no information that cannot be garnered from `cstate`. The output is a product of the next coordination state object and the input memories that need to be dealt with by nested instances. Here, the approach diverges from that given in rule C.133. Rule C.133 does not need this extra data for nested inputs, because it updates the coordination state objects in-place in a single invocation of the rule. The interpreter as presently realized instead defers these updates to the in-place objects until the nested instances are run, hence the need to keep track of this information in a product type. As further work, it is envisaged that this would be removed, which explains why the rule is structured as a fixpoint decreasing on the size of the instance definition. This is why arguments in the inner fixpoint are primed — they would change as the nested structure is traversed, with the instance being replaced by the relevant inner instance. We now examine Listing D.18 in digestible gobbets.

Listing 5.6: The coordination object deconstruction matches

```
match cstate′ as cstate′ return _ = cstate′ → _ with
  | exist csstatic cstp ⇒ fun J : cstate′ = exist _ csstatic cstp ⇒
    match csstatic as csstatic return _ = csstatic → _ with
      | exist csr cssp ⇒ fun J0 ⇒
        match csr as csr return _ = csr → _ with
          | CoordStateRaw_make mti″ mto′ csn
            ⇒
            fun J1 ⇒
```

Listing 5.6 shows how the raw underlying coordination state object is unwrapped from its σ-types and then itself deconstructed.

The matched variables that emerge from this deconstruction process (in a context where necessary facts about them can be proved) are the input memory map `mti''`, the output memory map `mto'`, and the nested coordination state map `csn`.

Listing 5.7: Deconstruction of the logical instance object

```
match (‘lisso′) with
  LInstSSORaw_make fmi′ fmo′ fmil fmol fmin fmon
  fl fn f‴
  mfi′ mfo′ mfin mfon boxmap obs manif fifos
  nestlinsts typeinsts instsigdat libinsts
  insttshiftmap libmapsigmap ⇒
```

The logical instance object contains several arguments, which we describe in turn. First, there are a number of frequencies. These are not strictly necessary, since they are aggregate (lowest common multiple) functions of the frequencies of memories specified within the instance object. They are there to improve the efficiency of the code, since to delve into the object for these frequencies every time they were needed would be unnecessarily computationally intensive. `fmi'` is the overall input frequency of the instance interface; `fmo'` is the overall output frequency. `fmil` is the overall input frequency of locally defined memories; `fmol` does the same for locally defined outputs. Similarly, `fmin` is the overall input frequency of nested memories whose interfaces are exposed to the enclosing instance in question; `fmon` does the same for nested outputs. These frequencies are needed in order to determine if local and/or nested operations need to be run on a particular input tick, whose frequency is the lowest common multiple of both input and output memories of local and nested memories. Similarly, the overall frequency of the instance (`f'''`) is the lowest common multiple of all input and output frequencies. This is the same as the lowest common multiple of `fl` and `fn`, which are the lowest common multiples of both input and output local and nested memories respectively.

`mfi'` and `mfo'` are the static specification maps for the locally defined input and output memory maps. `mfin` and `mfon` do the same for the nested case. `boxmap` is the map of local harmonic boxes. This specifies the computational element of the instance that is not due to nested instances.

`obs` and `manif` map the declared input and output memories of the instance respectively (which are unqualified identifiers) to identifiers that may be qualified by a nested instance. This ensures that how the instance uses nested interfaces is an implementation detail hidden from the user of the instance. The idea is that the user specifies the desired behaviour using a predicate, implementations of which are bisimilar *modulo* that predicate.

`fifos` associate output memories to input memories, thus specifying FIFOs. The temporal properties of those FIFOs are implicit in the memories at either end. The same connection logic that applies to `boxmap` also applies to `fifos`.

`nestlinsts` is a map of one-off nested instances. That is, they are directly defined and instantiated in the same place by their parent instance, and cannot be instantiated more than once or by any other instance. `typeinsts` is a map of instances that are specified elsewhere in an instance library. The library may be local or defined somewhere else. The libraries may be nested. The search semantics for libraries involve first looking in the local library, and then stepping back out of successive scopes and searching again, until either the instance definition is found, or the search fails at global scope. These resolution semantics ensure that no reference need be made to the absolute OID of the instance definition, and hence resolution semantics are invariant regardless of the prepended OID or instance environment.

`instsigdat` is a map that associates a signature and a set of dependencies with each required library instance. The signature ensures that a resolved library will fit the use to which it is put; the set of instance dependencies is required to establish the instances that may parametrize each such library instance's behaviour, and which need to be included in the closure of the current instance. This allows instances to be functors of other instances. `libinsts` is the local library of instances. `insttshiftmap` makes it possible to shift the time frame of the instantiated instances. This is necessary because, for example, if an instance occurs twice in a pipeline, the offsets from the current time for any particular time slice will be different in each case: older values will be considered by the second instance in the pipeline at the same time the newer values are being considered by the first instance in the pipeline. The time shift allows these delay semantics to be expressed from the point of view of the instance that itself instantiates the pipeline, but allows, by introducing a clock skew, the semantics inside the nested pipeline instances to be oblivious to this. This allows these inner instances to be instantiated in arbitrary temporal contexts. These facilities are used extensively in chapter 6.

`libmapsigmap` gives the signature of each map in the library. This could be derived from the library itself, but is present as a convenience to execution and proof.

Listing 5.8: Resolution of nest-qualified instance inputs

```
let mtiNestResolved :=

  exist _ (inMemBoxMapKeys (resolveObsInstQual obs)
    (‘ (mti′))) _
  in
```

This operation makes use of helper functions to recast the input memories so that those that are exposing some interface from a nested instance have the outer label dereferenced to the nested instance-qualified one.

Listing 5.9: Prepending of local inputs

```
let localUpdatedMti :
  InMemModInst.MDatMapTime fmil mfi′ :=
  let localMtiChanges :=
    FilterInputMapLocal _ _ mtiNestResolved
    in
    mtiPrepend _ _ (‘ (ttimeConv fmil f′ t′ _))
    _ _ (exist (InMemModInst.MDatMapFreqTimePred _
      mfi′)
    mti″ _) localMtiChanges
    in
```

In order to prepend local inputs, the memories are filtered to isolate those that are due for execution on the current cycle. The new data values can then be prepended to the list of values that are already in the memory. We need to do this because the core FIFO step does not process inputs that originate outside the scope of the instance.

Listing 5.10: Local FIFO execution

```
let (newLocMap, newNestMap) :=
  let nestedUpdatedMti :=
    let nestedMtiChanges :=
      FilterInputMapNested _ _ (‘linstsig′)
      mtiNestResolved
      in
      (exist (InMemModInst.MDatMapFreqTimePred _
        mfin) (‘nestedMtiChanges) _)
      in

      processFIFOs f′ t′ fmil fmin
      mfi′ mfin localUpdatedMti nestedUpdatedMti
      linstsig′ instTypeScopeMap′ lissolib′
      lisso′ cstate′
      (VaridMapMod.elements fifos)
      in
```

First the nested inputs are updated in a similar way to the local ones, then the function `processFIFOs` is called to step the local FIFOS. This moves values from box outputs to inputs as soon as they become available. A future implementation might keep explicit state for FIFOS, as is discussed in appendix C.3.2 and section 6.9.3. The FIFO-processing function returns local and nested input maps separately, since the nested map must be retained until it can be fed to the relevant nested box for deferred processing when it launches its own FIFO-step. This awkwardness is another reason why we would now prefer to update all memories in-place, to avoid passing round the supplementary arguments that do not appear in the semantic rules.

Listing 5.11: Nested FIFO execution

```
let newNestedCoordStates :
  LInstMapMod.t CoordStateRaw :=
  let csnmok :
    LInstMapModPred.NoDupType _ csn := _
    in
    (LInstMapMod.Build_t _ _ csnmok)
    in
```

This is a null operation, as we are currently not updating nested inputs. We should observe, however, that the scope of such an operation is the updating of nested inputs following memory dereferencing: movement of the FIFO only takes place during the nested invocation of a *box*, not a FIFO, and at that point the *local* FIFO code executes.

Listing 5.12: Construction of the new raw coordination state object

```
let newCSR := CoordStateRaw_make
  (InMemBoxWPties.update
    ('localUpdatedMti) ('newLocMap))
  mto'
  (LInstMapMod.this
    newNestedCoordStates)
  in
```

The construction of the new raw coordination state object involves re-combining the constituent maps, which have now been updated with new input values.

Listing 5.13: Construction of the new strong coordination object

```
let csStaticStrong := exist
  (CoordStateStaticPred f'
    linstsig' instTypeScopeMap'
    lissolib' lisso') newCSR _
  in
  let newUpdatedNestData :=
    exist _
    (InMemBoxWPties.update ('mti)
      ('newNestMap)) _
    in
    (exist
      (CoordStateInnerMemFBEnabled
        f' t' linstsig'
        instTypeScopeMap' lissolib'
        lisso') csStaticStrong _,
      newUpdatedNestData
    )
```

In Listing 5.13, we construct the new strong version of the coordination object, together with predicates for static and dynamic well-formedness. We admit the proofs for our stubbed-out predicates. The dynamic predicate has moved on to the FIFO-box-enabled state, the precondition for the next substep of the super-step to happen. This next step is not dependent on matching any further data from the input stream, so the four substeps can be thought of as being compressible into a single super-step. We retain the four separate phases for clarity and for reasons explained in section 5.5.3.

Listing 5.14: Match ends and supply of the equality proof

```
               end
            end eq_refl
          end eq_refl
        end eq_refl
```

Finally, as each match concludes, we supply the equality proofs for each match.

5.5.3 FIFO-box memory step

Rule C.134, which transforms the FIFO-box memory from a writable into a readable form and throws away stale values, is realized in Coq in Listing D.20. We now break this down and examine it.The basic structure of the function, extraction of the coordination state object and application of temporal filters are essentially the same as in section 5.5.2, so we start by looking at memory execution.

Listing 5.15: Memory execution

```
let mtiMemsExecuted :=
  InMemModBox.otm.map executeMem mtiMemTimeMapActiveSubset
  in
  let mtoMemsExecuted :=
    OutMemModBox.otm.map executeMem mtoMemTimeMapActiveSubset
    in
```

Memory execution involves reversing the list in a call to `executeMem` under a standard higher-order map function over the map of memories and throwing away stale values. The simplicity of this operation raises the question of why it is not incorporated in either the substantive box or FIFO execution steps. The reason is that we might later want to add a double-buffered memory in conjunction with a modified FIFO realization that stores its own state. Such a memory would need to be flushed at a moment when it is guaranteed not to conflict with a read or write by a box or FIFO. Even if not using a double buffer, some kind of temporal firewall is needed to produce a witness to the physical reconfiguration of data implicit in throwing away stale values. We also wish to stress that memories function independently of boxes or FIFOS, triggered only by the global clock.

While we could devise a bisimilar formalism that subsumed all steps (including double-buffered memories and the box step) into the driving FIFO step, this would obscure the time-triggered semantics. It would therefore not be a good canonical presentation.

Listing 5.16: Memory execution update

```
let mtiRawNew :=
  InMemBoxWPties.update (mti) mtiMemsExecuted
  in
  let mtoRawNew :=
    OutMemBoxWPties.update (mto) mtoMemsExecuted
    in
```

The memory update integrates the changed memories into the map that holds the memories that were not scheduled to change on this cycle, following the same pattern as the FIFO step of section 5.5.2.

Listing 5.17: Construction of the new coordination state object

```
let newCoordStateRaw :=
  CoordStateRaw_make mtiRawNew mtoRawNew csn
  in
  let newCoordStateRawStrongStatic :
    sig (CoordStateStaticPred f′ linstsig′
      instTypeScopeMap′ lissolib′ lisso′) :=
    exist _ newCoordStateRaw
    (depCoordStateStaticPredProcessNestedCSMapNOT_ENOUGH_ARGS _ _
_ _ _ _)

    in
    exist (CoordStateBoxesEnabled f′
      (tNext _ t′)
      linstsig′ instTypeScopeMap′ lissolib′
      lisso′) newCoordStateRawStrongStatic
    (depCoordStateOuterFIFOsEnabledProcessNestedCSMapNOT_ENOUGH_ARGS
_ _ _ _ _ _ _)
```

The construction of the strong (σ-type) dynamic coordination object is the same as in the FIFO case, but with two differences.

The first difference is that two admitted lemmas appear suffixed '`NOT_EOUGH_ARGS`', indicating that the lemmas will need to be supplied with more arguments from the current context before they can be proved.

The second difference is that the time of the next step is modified by `tNext`, because the next step being invoked is the box step, which by definition is the first thing to occur in a new time slice.

The equality proofs are the same as for the FIFO case.

5.5.4 Box step

The box step executes all of the harmonic boxes in the current logical instance and all its nested instances, reading values from the input (FIFO-box) memories and writing values to the output (box-FIFO) memories. It was described by rule C.135, and is realized in Coq in Listing D.21, while rule C.136 is covered by StepFunc in Listing D.22. We now look at each part in turn.

Listing 5.18: The box step package

```
Definition traceBoxesGenNFP(t : TTime f)
  (currState :
    sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
    (‘ (InstSigInputMems (‘linstsig))))
  (traceNestBoxesGenFunc : ∀ fn
    (csFIFOsEnabled :
      sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
        lisso CoordStateInnerFIFOsEnabled))
    (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
      (‘ (InstSigInputMems (‘linstsig)))),
    (sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso CoordStateOuterFIFOsEnabled) ×
    sig (mtoExternalNestPred (InstSigFreqMemOut (‘linstsig))
      ((InstSigOutputMems (‘linstsig))))
    )%type)
  (traceMemBFGenFunc : ∀
    csMemBFEnabled
    (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
      (‘ (InstSigOutputMems (‘linstsig)))),
    TraceMemBFEnab f t linstsig instTypeScopeMap lissolib
    lisso csMemBFEnabled
  )
  :
  (TraceBoxesEnab f t linstsig instTypeScopeMap lissolib lisso currState
  )%type.
    [Function body omitted]
Defined.
```

The function traceBoxesGenNFP in Listing 5.18 packages the complete box execution step, which drives the execution of locally declared boxes, and those nested within further logical instances. The features of note are the function parameters traceNestBoxesGenFunc and traceMemBFGenFunc. These are instantiated in the fixpoint and cofixpoint in which they are used by supplying the name of the function itself. We have by convention named this sort of function with the suffix ‘NFP’ for ‘no fixpoint’ to emphasize that the definition does not by itself set up complete recursion over the coinductive type of traces. The mtiNest parameter carries the new input values for nested boxes. If the code is modified to update nested inputs in-place, this argument may be removed.

The match of the exist constructor of the σ-type and the deconstruction of the instance object are identical to those in section 5.5.2.

Listing 5.19: Frequency and time proofs

```
let fLocalDivPrf : FreqDivide fl f := _
  in let instsigInDivPrf :
    FreqDivide (InstSigFreqMemIn (‘linstsig)) f := _
    in let fmiDivPrf : FreqDivide fmil f := _
      in let fmoDivPrf : FreqDivide fmol f := _
in let prfnxtim : isNextTimeStep fl (‘ (ttimeConv fl f t fLocalDivPrf))
  (tNext (‘ (ttimeConv fl f t fLocalDivPrf))) :=
  tNextIsNextTimeStep _ _
```

The first five terms in Listing 5.19, suffixed `DivPrf`, are proof terms required by time conversion functions. Time values are represented as natural numbers of ticks of a clock running at the frequency that is a dependent argument to the time. When a time must be supplied that is on a different frequency base, it must be converted to a different number of ticks in order to represent the same time. This conversion function requires a proof that the one frequency divides the other exactly; if it did not, the conversion could not be guranteed to give sensible results. `prfnxtim` certifies that `tNext t` is the next tick after `t`. This follows from the definition of `tNext`.

Listing 5.20: Box step predicate

```
in let boxesStepPred : BoxesStep.StepSSOPred
  fl fmil fmol mfi mfo hboxmap := _
```

The box step predicate strengthens the input and output maps for the box by providing proof that the input and output memory maps for the local boxes are coherent in the context of the current time. This is to be inferred from the coordination state dynamic predicate, which is present at this scope.

Listing 5.21: Instance time and frequency consistency proofs

```
in let lissoLeibLoc : FreqMemsInLissoLeib
  fl fmil fmol mfi mfo
  (‘ lisso) := _ in
  let lissoLeibNest :
    FreqNestInLissoLeib fn
    (‘ lisso) := _ in
```

The proof terms of Listing 5.21 are needed to show the equivalence of the consistency of the frequencies in the environment, as *indexed* by frequency as a dependent argument, and the concrete frequency matched in the instance. This inelegance is needed for the convenience of matching on a frequency that is implicit in the structure of the instance definition object, but would be computationally expensive to extract. We would consider its removal in a future reference interpreter.

Listing 5.22: Invocation of nested boxes evaluation

```
let nb := (traceNestBoxesGenFunc f
  (currStateNestBoxes t currState f)) mtiNest
in
```

In Listing 5.22, the parameter containing the nested box function is invoked, producing a trace which is in a coinductive type, but of a finite length: the nested instance can only run until it blocks on input from the enclosing instance. Coq's syntactic guard conditions force us to provide this function in a fixpoint version.

Listing 5.23: Memory map time conversions

```
let tfl := (‘ (ttimeConv fl f t fLocalDivPrf)) in
  let tfl' := (tNext (‘ (ttimeConv fl f t fLocalDivPrf))) in
    let tti := (‘ (ttimeConv fmil f t fmiDivPrf)) in
      let tti' := (‘ (ttimeConv fmil f (tNext t) fmiDivPrf)) in
        let tto := (‘ (ttimeConv fmol f t fmoDivPrf)) in
          let tto' := (‘ (ttimeConv fmol f (tNext t) fmoDivPrf)) in
```

Listing 5.23 handles the time conversions for the frequencies of input and output maps.

Listing 5.24: Inner map extraction

```
let mti :=
  (mtiCurrStateMti
    fl fmil fmol t mfi mfo currState lissoLeibLoc)
  in let mto :=
    (mtiCurrStateMto
      fl fmil fmol t mfi mfo currState lissoLeibLoc) in
    let nestCSR := (currStateNestBoxes t currState fn)
```

The definitions of Listing 5.24 access the input, output and nested maps from the coordination object.

Listing 5.25: Execution of local boxes

```
in let outp :=
  BoxesStepImpl.StepFunc fl fmil fmol
  tfl tfl' tti tti' tto tto'
  mfi mfo (exist _ _ boxesStepPred)
  (tNextIsNextTimeStep _ _) (exist _ _
    (boxesPre fmil tti mfi mti))
  in
```

Listing 5.25 invokes the local box execution function, producing output memories. We return to it in Listing 5.30.

Listing 5.26: Prepending the box output

```
let outpPrepended :=
  mtoPrepend fmol fmol (‘ (ttimeConv fmol f t _))
  mfo mfo mto (‘ (‘outp))
  in
```

Listing 5.26 prepends the outputs generated by Listing 5.25 to the relevant memories in the local maps. Some of these memories — the ones that were not scheduled to run in this tick — will be unaffected by this procedure.

Listing 5.27: Construction of the new strong map of nested instance states

```
let nestCSR' := (exist _ (` (fst nb)) _) in
```

Listing 5.27 shows the map that results from Listing 5.22 being strengthened with the necessary predicate to make it a valid nested map of the current instance.

Listing 5.28: Construction of the coordination state object at current scope

```
let cstateNext := (buildCurrStateMemBF fl fn fmil
  fmol t tfl' tti' tto' mfi mfo mti outpPrepended
  nestCSR') in
```

Listing 5.28 makes the new coordination state for the current instance out of the original local input map, the new local output map, and the new nested map.

Listing 5.29: Construction of the new trace object

```
let traceBF :=
  let mtoResolved :=
    exist _ (outMemBoxMapKeys _ (resolveManifInstQualRev manif)

      (` (` (snd nb)))) _
    in
    traceMemBFGenFunc cstateNext (` (snd nb))
    in let bsteppre := _ in let bsteppost := _ in

      TraceBoxesStep
      f t linstsig instTypeScopeMap lissolib lisso fl fn
      fmil fmol tfl tfl' t
      (` (ttimeConv (InstSigFreqMemIn (`linstsig)) f t
        instsigInDivPrf)) tti tti' tto tto' mfi mfo
      (exist _ _ boxesStepPred) mti mto mti outpPrepended
      nestCSR nestCSR' prfnxtim bsteppre bsteppost currState cstateNext
_ _ traceBF
```

Listing 5.29 makes a new trace by wrapping `cstateNext` from Listing 5.28 around the old trace, using the inductive constructor of the coinductive box trace type. This is the only constructor of this type: the only terminal constructor is in the FIFO trace type, but this is accessible from the box trace type because the trace types are mutually coinductive, and could be reduced to a single non-coinductive type if we dispensed with building separate trace types for the explict substeps of our super-step.

Listing 5.30: Local box step function

```
Definition StepFunc(f fmi fmo : Freq)
  (t t' : TTime f CoordPhase )
  (ti ti' : TTime fmi)(to to' : TTime fmo)
  (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
  (stepSSO : BoxesStep.StepSSO f fmi fmo mfi mfo)
  (nexttimeprf : isNextTimeStep f t t')
  (instate : sig (InMemModInst.MDatMapModeReadPred _ ti mfi) ) :
  sig (BoxesStep.StepPred
    f fmi fmo t t' ti ti' to to' mfi mfo stepSSO nexttimeprf instate).
```

The local box step function executes all of the harmonic boxes local to the logical instance. We now review its arguments. `f`, `fmi` and `fmo` are the familiar overall frequency, input memory map frequency and output memory map frequency respectively. They are again conveniences, these frequencies being a function of the memory maps in question. The times `t`, `ti` and `to` all refer to the time of the present slice, couched in terms of their respective frequencies. The primed versions of these variables correspond to the next time slice: they are needed because the time of the memories updated after execution is incremented to the next time slice, and the predicates over these memories need to be qualified by these new times. Again, these arguments are conveniences, since they can be derived from the current time. We prefer this approach rather than using the `tNext` function, since the former approach is in better accord with the declarative style suggested by a predicate-driven style of programming. `mfi` and `mfo` are the input and output specification maps for the memories that are connected to the boxes being stepped. `stepSSO` is a composite σ-type, that provides a map of harmonic box specifications, together with identifier-mapping information to bind their interfaces to their environment. `nexttimeprf` ensures the relationship between the time values in scope is what we have just described. Finally, `instate` is the state of the input memories before execution. Boxes do not in general keep internal state between invocations, although this is a possible extension project. The only exception to this is the nested box situation, which is treated in this formalization as a separate class of object. See section 5.5.5 for a description of how this works.

Listing 5.31: Box-processing fixpoint

```
let fix processBoxesMapAsList
  (dl : list (ipm.boxidPred.PredidDecidable.t ×
    (HBoxSSONonDep × InMemModBox.otm.t Varid × VaridMapMod.t HBCL_OidMemBF)))
  (inclprf : SetoidList.inclA (@BoxTypeIdMapMod.eq_key_elt _ ) dl
    (BoxTypeIdMapMod.elements
      ( (‘ (stepSSO))))
  ) { struct dl }
  : OutMemModBox.MDatTimeMapRaw :=
  match dl as dl return _ = dl → _ with
    | nil ⇒ fun _ ⇒ OutMemModBox.otm.empty _
    | (hbx :: dl′)%list ⇒ fun J : dl = (hbx :: dl′)%list ⇒
      let
        accumMap : OutMemModBox.MDatTimeMapRaw :=
        processBoxesMapAsList dl′ _
        in
        OutMemBoxWPties.update
        (accumMap)
        (stepOneBox _ _ _ t ti to mfi mfo instate hbx)
  end eq_refl
  in
```

The box-processing function renders the map of boxes as an (arbitrarily ordered) list, and goes through each box in turn, updating the overall memory map to reflect each box execution. This use of lists satisfies the well-foundedness requirement, because recursive calls decrease structurally on the size of the list. The drawback of this approach is that it loses logical generality of the operation, which is more satisfactorily viewed as belonging to the 'map reduce' paradigm of functional programming: this could be a future refinement, using the unordered fold function of the Standard Library map interface. However, this solution breaks down when we try and use the same technique with nested maps. The computation for each box is triggered in `processBoxesMapAsList` by the invocation of `stepOneBox`. We start to review this function shortly in Listing 5.34. The term `inclprf` allows us to conclude that properties of the map in which each list element must be found, and which imply things about each such element, are also properties that hold over each sub-list derived from that map.

Listing 5.32: Raw output subset construction

```
let outputSubsetRaw : OutMemModInst.MDatMapTime fmo mfo := exist _
  (processBoxesMapAsList (BoxTypeIdMapMod.elements
    (
      (‘ (stepSSO))))

  (inclReflBoxid _ _)) _

  in
```

`outputSubsetRaw` is built by invoking `processBoxesMapAsList`.

Listing 5.33: Strong output subset construction

```
let outputSubset := exist _ outputSubsetRaw _ in
  exist (BoxesStep.StepPred
    f fmi fmo t t' ti ti' to to' mfi mfo stepSSO nexttimeprf instate)
  outputSubset _
```

Listing 5.33 shows the requirement to establish the temporal correctness of the memories following their updates by box execution. In common with many such book-keeping propositions, we have admitted them in the Coq code without proof.

Listing 5.34: Single box step variables

```
Variables f fmi fmo : Freq.
Variable t : TTime f.
Variable ti : TTime fmi.
Variable to : TTime fmo.
Variable mfi : InMemModInst.MDatFreqMap fmi.
Variable mfo : OutMemModInst.MDatFreqMap fmo.
Variable stepSSO : BoxesStep.StepSSO f fmi fmo mfi mfo.
Variable instate : sig (InMemModInst.MDatMapModeReadPred _ ti mfi).
Variable hbx : boxidPred.PredidDecidable.t ×
  (HBoxSSONonDep × (InMemModBox.otm.t ipm.Varid)
    × (ipm.VaridMapMod.t opm.HBCL_OidMemBF)).
Hypothesis inprf : SetoidList.InA (@BoxTypeIdMapMod.eq_key_elt _) hbx
  (BoxTypeIdMapMod.elements (‘stepSSO)).
```

These variables form the arguments to `stepOneBox` and its supporting helper functions and lemmas. They have the same meanings as in `StepFunc`, although the variable `hbox` is new. This is the specification of the particular harmonic box that `stepOneBox` is to process.

Listing 5.35: Single box step harmonic box matches

```
Definition stepOneBox : OutMemModBox.MDatTimeMapRaw.
  match (snd hbx) as hbxt return _ = hbxt → _ with
    | (Build_HBoxSSONonDep hbf hbfi hbfo hbinmem hboutmem hbsso,
      inmemmap, outmemmap) ⇒
    fun J ⇒

      match hbsso as hbsso return _ = hbsso → _ with
        | exist hbraw hbpred ⇒ fun J1 ⇒
```

The first match of Listing 5.35 unpacks the indexed form of the harmonic box static semantic object into the arguments doing the indexing and the dependent type that those arguments index. It also deconstructs the product type that holds the information for binding the harmonic box's variables to its environment. This paradigm of unpacking the indexing arguments to a dependent type built from a parametrized σ-type occurs repeatedly in this development. It is a convenient way to build a prototype. These indexing arguments are implicit in the harmonic box itself, and it would be neater to recover them (insofar as they are needed only as arguments to other predicates) from a single predicate that was constant across the map, reducing the number of redundant matches in the exported code. Even so, there are substantial benefits to containing harmonic boxes in a heavily indexed type such as this, since any inhabitant of this dependent type (parametrized on the properties of its environment) can more easily be guaranteed to produce terminating functions when they are supplied to the evaluation machinery. It is this that makes worthwhile the apparent inconveniences of heterogeneous dependent types in maps. This sort of indexing is in a similar spirit to that used by Chlipala in his intrinsic embeddings of lamda calculi in Coq [46, 48]. The second match of Listing 5.35 extracts the raw data object from the σ-type in the ordinary way, making the predicate accessible to proofs involved in constructing the next σ-type.

Listing 5.36: Single box step time conversions

```
let thb := ‘ (ttimeConv hbf _ t _)
  in
  let thfi := ‘ (ttimeConv hbfi _ ti _)
    in
    let thfo := ‘ (ttimeConv hbfo _ to _)
      in
```

The time conversions of Listing 5.36 recast the current time into the clock frequencies of the subsets of the input and output memory maps that are used by the harmonic box in question (thfi and thfo), and the overall harmonic box frequency (thb).

Listing 5.37: Strong input map recovery

```
let inp :
  InMemModBox.MDatBoxTime MDataInst.ReadEnabled
  (InMemModBox.otm.map MDataTypeInst.MDFE_boxMemDat (‘mfi)) hbfi thfi :=
  exist _ (‘ ( (‘instate))) _
  in
```

Listing 5.37 recovers the σ-type version of the input map, so it is suitable for feeding as an argument to the invocation of a harmonic box execution.

Listing 5.38: Single box step computation result

```
let hboxresult := HBoxStep hbf _ _ thb thfi
  thfo

  hbinmem hboutmem inmemmap outmemmap
  (InMemModBox.otm.map (MDataTypeInst.MDFE_boxMemDat) (‘mfi))
  (OutMemModBox.otm.map (MDataTypeInst.MDFE_boxMemDat) (‘mfo))
  hbsso inp
  in (‘ (‘ hboxresult))
```

Listing 5.38 constructs the result of the harmonic box computation by calling HBoxStep. HBoxStep is a helper function of the harmonic box that binds arguments in the form specified by the harmonic box language given in the harmonic box static semantic object. In the current implementation, this in turn immediately binds the types of arguments and results to an untimed box language, which in turn wraps our basic expression language. Temporal data is stripped off when an untimed box language is used, and re-added according to static rules when the output is bound back into the harmonic box environment. Further detail of the harmonic box formalization can be found in section 5.4 and appendix D.1.7.

5.5.5 Nested box step

Rule C.139 is realized in Coq in Listing D.25. We now look at this in more detail.

Listing 5.39: The FIFO function variable

```
Variable traceFIFOsGenFunc : ∀
  f linstsig instTypeScopeMap lissolib
  lisso t


  csInnerFIFOsEnabled
  (mti : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
    (‘ (InstSigInputMems (‘linstsig))))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
    (‘ (InstSigOutputMems (‘linstsig))))
    ,
        TraceFIFOsEnab f t linstsig instTypeScopeMap
        lissolib lisso
        csInnerFIFOsEnabled.
```

The FIFO-processing function is set out as a section variable in Listing 5.39. This is instantiated by the *fixpoint* version of that function, since in a nested context, instance executions must always terminate. In the present version of the formalization, where input memories are given as parameters in nested cases (as opposed to being modified in-place by an enclosing instance), the stream of input data given is restricted to finite instances of the input data stream coinductive type.

Listing 5.40: The nested box processing entry point

```
Definition traceNestBoxesGenNFP(fn : Freq)(t : TTime f)
  (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
    lisso CoordStateInnerFIFOsEnabled))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn ('linstsig))
    (' (InstSigInputMems ('linstsig)))) :
  (sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
    lisso CoordStateOuterFIFOsEnabled) ×
  sig (mtoExternalNestPred (InstSigFreqMemOut ('linstsig))
    ((InstSigOutputMems ('linstsig))))
  )%type.
```

The nested box function takes two interesting arguments, as well as the familiar frequency and time parameters, and the FIFO-processing variable of Listing 5.39. These are `nestCSR`, which is the map of nested coordination objects, and `mtiNest`, which is the input data for each of these coordination states. The return type is the Cartesian product of the new coordination state map (wrapped by a suitable predicate) and the output data that will be used by the next FIFO step at the scope of the parent instance. The latter term is a convenience that can be derived entirely from the former.

Listing 5.41: Set-up of the nested box processing fixpoint

```
  let fix processNestedInst
    (l dl : list (HBCL_OidLInst × (CSTempCorrectND × TTFL)))
    (inclprf : SetoidList.inclA (@LInstMapMod.eq_key_elt _ ) dl l)
    (inprf : boxesEnabNestPred t l)
    (mapsofar : LInstMapMod.t CSTempCorrectND)
    (outdatsofar : OutMemModBox.MDatTimeMapRaw)
    : ((LInstMapMod.t CSTempCorrectND) × OutMemModBox.MDatTimeMapRaw)%type :=
    match dl as dl return _ = dl → _ with
      | nil ⇒ fun _ ⇒ (mapsofar, outdatsofar)
      | cons (v, (cs, ttfl)) m' ⇒ fun J : dl = cons (v, (cs, ttfl)) m' ⇒
```

Listing 5.41 sets up the inner fixpoint that traverses the map of nested instances and processes each one in turn. The same issues of list independence that we discussed in Listing 5.31 apply here.

Listing 5.42: Recursive call of nested box processing fixpoint

```
        let mnew := processNestedInst
              l m′ (inclNest _ _ _ _ _ _ inclprf J)
inprf mapsofar outdatsofar in
```

Listing 5.42 invokes the processing of the nested boxes and assigns the updated map to mnew.

Listing 5.43: Setting up the new nested coordination state

```
let csnew :=
  match TTseqb _ _ (ttimeTTFLAdjust f (tPrev _ t) ttfl)
    (CSTempCorrectND_t cs)
    as ttseqb return
      _ = ttseqb → _ with
```

Listing 5.43 declares the new nested coordination state object being dealt with on this recursive call (its type is inferred from the definition that follows). The execution path immediately splits according to whether the current time slice enables the nested instance or not. This is an equality test. We could equally well derive this information statically from the instance specification object by comparing the clocks in the frequency domain.

Listing 5.44: The beginning of the active match clause

```
    | true ⇒ fun J0 : TTseqb _ _
      (ttimeTTFLAdjust f (tPrev _ t) ttfl)
      (CSTempCorrectND_t cs) = true ⇒
      let csbp := (inprf (v, (cs, ttfl)) _ _)
        in
```

Listing 5.44 is the beginning of the case of the match where the nested object *is* enabled in the current temporal scope. It sets up some proof terms that capture the context of the match: this is essential in order to be able to build a strong object in this context.

Listing 5.45: Input map filter

```
        let mtiNest′′ :=
          FilterInputMapNestedInst v _ _
          (‘ (CSTempCorrectND_linstsig cs)) mtiNest
          in
```

The input map filter selects those nested inputs that are qualified by the identifier of the present nested coordination object, extracts only those and removes the prepended indentifier, thus transforming it into an input map at the local scope of the nested instance.

Listing 5.46: Inner trace instantiation

```
let fTrace :=
  (traceFIFOsGenFunc
    (CSTempCorrectND_f cs)
    (CSTempCorrectND_linstsig cs)
    (CSTempCorrectND_instTypeScopeMap cs)
    (CSTempCorrectND_lissolib cs)
    (CSTempCorrectND_lisso cs)
    ( (CSTempCorrectND_t cs))
    (exist (CoordStateInnerFIFOsEnabled (CSTempCorrectND_f cs)
      ( (CSTempCorrectND_t cs))
      (CSTempCorrectND_linstsig cs)
      (CSTempCorrectND_instTypeScopeMap cs)
      (CSTempCorrectND_lissolib cs)
      (CSTempCorrectND_lisso cs))
    (CSTempCorrectND_CS cs) csbp)
  )
  mtiNest''
  (exist _ (OutMemModBox.otm.empty _) _) in
```

Listing 5.46 shows the definition of `fTrace`, which is the section of execution trace that describes the behaviour of the nested coordination object for the period until its next invocation. This may include the generation of further levels of nested traces, although the trace information is actually thrown away when each level of nesting returns, giving just the final coordination state reached at the end of the trace. The accessors suffixed '`ND`' are used to restore the dependent type from the non-dependent container (a map cannot store unindexed heterogeneous dependent types). This method is convenient, but as explained in the discussion of Listing 5.35, it is not optimal.

Listing 5.47: Extraction of result from inner trace

```
let newcs := (processSubInstFinite
  cs (inprf (v, (cs, ttfl))
    (inprfNest _ _ _ _ _ _ inclprf J)
    (ttseqCorr v cs ttfl t J0 ))
  fTrace) in
```

Listing 5.47 calls `processSubInstFinite`, whose single job is to mine to the bottom of the nested trace fragment and retrieve the final coordination state and the extracted output map.

Listing 5.48: Non-dependent nested instance construction

```
({| CSTempCorrectND_f := CSTempCorrectND_f cs;
  CSTempCorrectND_t :=
  (starvationLongstop cs);
  CSTempCorrectND_linstsig :=
  CSTempCorrectND_linstsig cs;
  CSTempCorrectND_instTypeScopeMap :=
  CSTempCorrectND_instTypeScopeMap cs;
  CSTempCorrectND_lisso := CSTempCorrectND_lisso cs;
  CSTempCorrectND_CS := (' (fst newcs))
|}, (snd newcs))
```

In Listing 5.48, `newcs` from Listing 5.47 is re-established as the necessary σ-type, dependent in the necessary arguments of its predicate.

Listing 5.49: Match for non-active nested instances

```
  | false ⇒ fun _ ⇒
    (cs, exist _ (OutMemModBox.otm.empty _) _)
end eq_refl
```

Listing 5.49 deals with the case where the nested coordination instance being considered is not scheduled to run during this time slice. It therefore outputs the original coordination state object and an empty output map. Finally, in Listing 5.49, the match structure for selecting active nested objects ends and the usual equality proof, `eq_refl`, is supplied.

Listing 5.50: Transformation of nested keys

```
in let vLiblessStrong := exist liblessInst v _
  in let extKeysLifted := outMemBoxMapKeys
    (concatInstMemBF vLiblessStrong) (‘ (snd csnew))
    in
```

The definitions of Listing 5.50 recast the identifier space of the nested instance back into the identifier space of the enclosing instance.

Listing 5.51: Accession of new nested instance to map

```
    ((LInstMapMod.add v (fst csnew) (fst mnew)),
      (OutMemBoxWPties.update (snd mnew) extKeysLifted)
    )
  end eq_refl
```

Listing 5.51 adds the new nested map to the nested coordination state, overwriting the old state. This is the last operation of the map-processing fixpoint.

Listing 5.52: Invocation of the map-processing fixpoint

```
  in let processRaw :=
    let linstttflmap :=
      match (‘lisso) with
        LInstSSORaw_make _ _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ ttflinstmap _ ⇒ ttflinstmap
      end in
      let linstttflmapOK := _ in
        processNestedInst
        (LInstMapMod.elements (combineLInstMapModMaps (‘nestCSR) linstttflmap
          linstttflmapOK))
        (LInstMapMod.elements (combineLInstMapModMaps (‘nestCSR) linstttflmap
          linstttflmapOK))
        (inclRefl _ _)
        (coordStateMapBoxesNest t linstttflmap fn (‘nestCSR) (“nestCSR) _)
        (LInstMapMod.empty _)
```

```
(OutMemModBox.otm.empty _)
in
```

In Listing 5.52, `processNestedInst` is invoked following the preparation of some subsidiary definitions. `ttfllinstmap` is the `insttshiftmap` argument we described with reference to Listing 5.7. The `linstttflmapOK` predicate states that the map extracted from the static instance object is well formed. This is necessary in order to be able to convince Coq that the map is of the correct type to be considered a Standard Library map. This is an instance of the paradigm we described in section 5.2.2.3: predicates and raw types have to be stored separately and recombined where necessary for functions that expect composite arguments.

Listing 5.53: New predicate for nested coordination objects

```
let processPred : InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib lisso
  CoordStateOuterFIFOsEnabled (fst processRaw) := _ in
```

Listing 5.53 ensures that the resulting map, after all update operations, is well-formed.

Listing 5.54: Construction of the nested step completion object

```
let mtoExtTimeStrong := exist _ (snd processRaw) _ in
  let mtoExtStrong := exist (mtoExternalNestPred (InstSigFreqMemOut ('linstsig))
    (InstSigOutputMems ('linstsig))) mtoExtTimeStrong _ in
  (exist _ (fst processRaw) processPred, mtoExtStrong)
```

Listing 5.54 does the same as Listing 5.53, but for the assisting output map. Both are then packaged up in a Cartesian product and returned.

5.5.6 Box-FIFO memory step

Rule C.141, which transforms the box-FIFO memory from a writable into a readable form and throws away stale values, is realized in Coq in Listing D.24. It is almost identical to the FIFO-box step discussed in section 5.5.3, in that its essential task is to reverse the list and throw away stale values. We therefore do not examine it again here.

5.5.7 Super-step assembly

In appendix C.3.1, we have given an account of the operation of the program super-step in terms of formal semantic rules. We now show how the component substeps we have reviewed in this section are combined into a single mutually recursive fomalization. In appendix C.3.1, we have described how the five parts of the super-step are defined individually, and in section 5.5.2, section 5.5.3, section 5.5.4, section 5.5.5 and section 5.5.6 we showed how these are rendered in Coq in such a way as to allow us to convince the type-checker that the structure is well-founded. In our explanation of Listing 5.46 in section

5.5.5, we introduced the reason why we need both cofixpoint and fixpoint versions of the same structures. These are instantiated in Listing D.26 by supplying other member functions of the (co)fixpoint as arguments to the individual substeps. In checking that the whole (co)fixpoint meets the required guard conditions, Coq first makes the necessary substitutions of definitions (δ-reductions). In the case of the cofixpoint, Coq checks that recursive calls are under a constructor; in this case, the accessibility predicates are not needed and are trivially satisfied. In the fixpoint version, the accessibility predicates are needed, and the nested case requires nested fixpoints. The resulting function is listed in Listing D.27. The instantiated fixpoint is itself a parameter to the instantiation of the cofixpoint.

5.6 Formalization of the example expression language

The expression language uses the same approach to the compilation of a 'static' semantic object, though at present we do not have a compiler and perform this process by hand. The static semantic object has a dependent type that, when Curried as an argument to a reduction function, yields a Coq function that reduces the main function of that static semantic object (as applied to its intrinsically typed data) according to the semantics of Coq.

The key to formalizing the expression language is in handling structures that are expressive enough to produce non-termination, but nevertheless compile into Coq functions that guarantee to terminate in Coq's primitive recursive, strongly normalizing executable logic. As we saw with the coordination language, it is often not trivial to identify how a fixpoint will reduce structurally in its argument. HBCL's basic expression language repeatedly uses a design pattern in which higher order map functions are used, both on lists and record types. This occurs in the look-up of data values and functions by their names from environment scopes, and also in the processing of patterns and the construction of new types over tuples and records. The solution to finding a termination proof term with an appropriately decreasing size lies in determining ordering relations on sized types and computational potential, which can then be used in producing an accessibility predicate. The full power of sized types is not needed in the present implementation, since data types have sizes that are fixed at compile-time.

5.6.1 Sized types

The idea of 'sized types' was coined by Hughes *et al.* [112]. When we first formalized our type system, we envisaged sized types would be necessary in order to show termination. Since then, we have found it *is* possible to define a total ordering on the sizes of inductive predicates that are parametrized by associative arrays.[8] This implies that sized types are strictly unnecessary for the present type system, and some of our later predicates in the same paradigm reflect this new method. However, sized types would still be necessary for any extensions to types of indeterminate size, where they would be required to make linkages between the computational costs of functions, these costs being parametrized in the sizes of types. For simplicity, we have used concrete Coq types for sizes. Since sizes do not have a computationally useful content (and neither do costs) it would be preferable in future to move them into the `Prop` sort.

5.6.2 Cost functions

Our need for cost functions is very limited, given that our demonstration expression language does not have control flow structures. Our cost structures are thus toys compared to the power tools that a language such as Hume has available to it [114,115,131]. However, the ability of our expression language freely to call named functions means that, without cost data, there is no way to prove termination. There is a limitation of Coq in play here. It is difficult to satisfy well-foundedness over nested calls to a map function, unless we modify the map interface to pass a proof term that reduces in size with each call and shows well-foundedness. An alternative approach is to modify the type of the function itself on recursive calls, so that it operates on a σ-type with a reduced size ceiling. This involves re-casting the data on each recursive call to a lower 'potential ceiling', which is inefficient, and utterly redundant when code is exported from Coq. We would prefer the former option for a map function with a richer interface. However, since it was impractical and incidental to our subject matter to produce modified versions of Coq's standard libraries, we have settled for the latter approach, which though inefficient, works adequately.

Finally, we observe that in the binding of our expression language to the coordination language, cost data is not preserved. This is left to further work.

5.6.3 Structure of the formalization

The structure of the expression language formalization follows the rules given in appendix C.4. We provide comprehensive listings of the definitions in appendix D.6 and *via* the table in appendix D.3. The number of admitted lemmas in the expression language is low, with all of the most important proof terms fully proven. We do not print most proof scripts, but supply a sample in Listing D.36.

[8]We solved this problem when we later implemented nested coordination state objects in the coordination language.

5.7 Provable properties

The kinds of properties that we can prove in this system fall into two categories. First, there are proofs that a program (that is, instance specification) in HBCL satisfies some predicate over the coordination state trace space, quantified over all possible input streams. Second, we could prove that a generic transformation of an HBCL program preserves these properties, both in the case of a transformation that is reflexive in HBCL, such as a replication transformation, and in the case of a physical implementation. We shall illustrate all of these points using the replicated multiplier example that we gave in section 3.6, using a category diagram and some Coq code.

In Listing 5.55, we have written down a simple specification of what a multiplier does over a Peano axiomatization of natural numbers. We arrived at this by adapting the executable (fixpoint) definitions of addition and multiplication from the Coq Standard Library. In this purely propositional specification, we do not say anything about executability: to prove that the relation is computable, the simplest way would be to use the fixpoint version of the same specification as a witness relating the arguments to the result.

Listing 5.55: An axiomatization of multiplication

```
Inductive Plus(n m l : nat) : Prop :=
| Plus0 : n = 0 → l = m → Plus n m l
| PlusS : (∃ p : nat, ∃ q : nat,
   S p = n → l = S q → Plus p m q) → Plus n m l.
Inductive Mult(n m l : nat) : Prop :=
| Mult0 : n = 0 → l = 0 → Mult n m l
| MultS : (∃ p : nat, ∃ q : nat,
   S p = n → Plus m q l → Mult p m q) → Mult n m l.
```

To understand how this kind of specification can be related to an HBCL specification, we make use of the category diagram in Figure 5.2. The diagram is laid out as a chain of morphisms. The digram is in two rows: this is purely so that it can be printed large enough to see what is happening. Within each of these rows, the diagram has a ladder-like structure. Along the top edge of the ladder, we see a number of different axiomatizations of natural numbers. As we move from left to right, these become decreasingly intuitive. On the left-hand, most intuitive side, we have a Peano axiomatization. n and m are the two numbers we want to multiply, and l is the result. The injective relation *Mult* specifies an injection whose (non-unique) result l is the product of each (unique) pair of arguments n and m. The next vertical arrow to the right represents the same relation, but over a binary axiomatization of natural numbers, which again are an integral part of the Coq Standard Library. The question is, how do we know the predicate that we could write called *Mult(binary)* is the same relation as *Mult(nat)*? The answer lies in what we claim with the double-ended horizontal arrows. These arrows represent another relation (both top and bottom happen to be the same one). It is a bijection between the Peano axiomatization of natural numbers and the binary axiomatization of natural numbers. To use the category- or type-theoretic terminology, the injections of the vertical arrows are monomorphisms, while the vertical arrows are isomorphisms. The *double*-ended arrow signifies the *bi*jection.

The diagram gives us an easy way to visualize the key property that we are interested in, which is difficult to read in Coq. The property is that the relation *Mult(nat)* is the same relation as the transitive concatenation of *Eq(Peano, binary)*, *Mult(binary)* and *Eq(binary, Peano)*. We can therefore state as a theorem that there is a structure morphism between the two *Mult* relations with respect to the equivalence relationship between the axiomatizations over which they are defined. A proof of this theorem then allows us to conclude that any inhabitant of the *Mult(binary)* relation has an equivalent inhabitant of the *Mult(nat)* relation. This is an extremely important property, because it allows us to make conclusions about 'mathematically pure' *statements* of problems from the ugly but mechanistically efficient and reliable implementation of axiomatizations such as the binary one of natural numbers: it would be completely impractical to multiply natural numbers by adding n to itself by repeated applications of the succession constructor m times, even though this is a much better way of axiomatizing what is *meant* by natural number multiplication. In the logic domain, this is analogous to the way that we can become convinced that a theorem is true by witnessing the mechanistic application of a proof-checking algorithm.

The rest of Figure 5.2 shows how this argument proceeds transitively to HBCL, including a reflexive transformation in HBCL to a replicated version. The equivalence relation between replicated and non-replicated versions is an interpretation function — in this case, voters — of the three replicated states. For completeness, it also shows how this can be carried into the physical domain, whereupon the difference in time instants between the arguments of the multiplication and results (which are added at the HBCL stage) are changed to *intervals* in continuous time, matching the rubber sheet abstraction we discussed in chapter 3. The further to the right in the diagram we look, the less the relation contained in the vertical arrow has to do with multiplication, and the more it has to do with the semantics in physics, until at the far right of the equation, all of the 'multiplicationness' of the relation has been pushed into the arrangement of the state of matter in space that represents the initial conditions of the multiplication and the interpretation of the result. There is more than a little structuralist philosophy driving this way of looking at things: the structure of multiplication is directly linked within a logic between quasi-Platonic mathematical entities on the left to physical entities on the right, linked only by laws of Nature.

Finally, we observe that the horizontal arrows in Figure 5.2 which are only leftwards-pointing are *sur*jective relations, or *epi*morphisms. They are a formalized version of

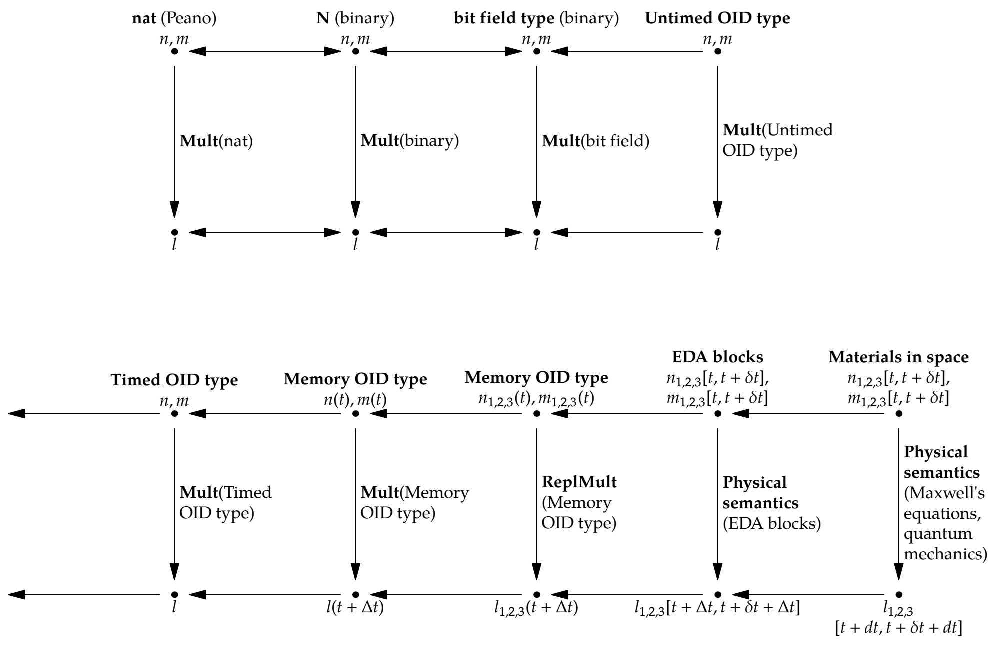

nat (Peano)
n, m
N (binary)
n, m
bit field type (binary)
n, m
Untimed OID type
n, m
Mult(nat)
Mult(binary)
Mult(bit field)
Mult(Untimed OID type)
l
l
l
l
Timed OID type
n, m
Memory OID type
$n(t), m(t)$
Memory OID type
$n_{1,2,3}(t), m_{1,2,3}(t)$
EDA blocks
$n_{1,2,3}[t, t + \delta t]$, $m_{1,2,3}[t, t + \delta t]$
Materials in space
$n_{1,2,3}[t, t + \delta t]$, $m_{1,2,3}[t, t + \delta t]$
Mult(Timed OID type)
Mult(Memory OID type)
ReplMult (Memory OID type)
Physical semantics (EDA blocks)
Physical semantics (Maxwell's equations, quantum mechanics)
l
$l(t + \Delta t)$
$l_{1,2,3}(t + \Delta t)$
$l_{1,2,3}[t + \Delta t, t + \delta t + \Delta t]$
$l_{1,2,3}$ $[t + dt, t + \delta t + dt]$

Figure 5.2: Multiplier category diagram

what we have previously called 'interpretation functions'. Whereas the multiplication of binary-represented natural numbers is the same relation as the concatenation of the *Eq(binary, Peano)*, *Mult(Peano)* and *Eq(Peano, binary)*, one cannot recover the semantics of physics by going *via* a multiplication relation in the same way.

The most important practical implication of this approach for the kinds of properties we can show with HBCL specifications is that, with HBCL axiomatized in a general logic, we can show equivalences of this kind with *any* axiomatization we choose, be it an axiomatization of the laws of Nature (physics), or an axiomatization of *a priori* synthetic objects (mathematics, at least from a Kantian perspective). This is not possible with monolithic specification tools such as Event-B or TLA+, which contain their own specialized logics and deductive systems that are not general enough to axiomatize arbitrary concepts. In chapter 6, we consider a small part of this category diagram, showing emperically the surjective interpretation relation between the ordinary and replicated multipliers — a relation that is reflexive in HBCL. We also see how similar relations can be constructed over specifications composed into more complex systems, such as the parallel composition example.

5.8 Summary

The formalization of HBCL's structural operational semantics as a Coq function enables us to test HBCL programs by supplying suitably encoded static semantic objects as Curried arguments to it. The resulting functions can then be executed by applying them to input streams: the semantics in this form have become an executable HBCL interpreter. The formalization produces traces which are amenable to inspection. It would be possible to take a predicate defined over some application-specific domain and a Pre-HBCL signature and attempt to prove that the trace, as produced by the interpreter, satisfied the predicate. In practice, it would be easier to use the predicate characterization of the semantic domain of HBCL to prove that a particular static semantic object gave rise to an execution that satisfied the predicate, because it is much easier in Coq to prove things by induction over predicates than to prove things over the structure of functions. The structure of functions is only evident in a Coq proof by studying the inductive structure of the data on which they operate, and it is easier to do this when this structure is closely matched with an inductive predicate which can be pattern-matched by a proof. It is therefore desirable in further work to strengthen the predicates of the semantic domain so that the interpreter witnesses an injective predicate of that domain. This would involve repeated application of the technique of parametrized σ-types that we developed in the context of the predicate calculus example in appendix B, and which we have partially applied in the context of our coordination and expression languages.

We have advanced two of the hypotheses of the thesis in this chapter. We have obtained soundness of HBCL up to the soundness of Coq, essentially for free, by supplying for each inductive predicate characterization of a semantic rule, a Coq function. This proceeds inductively over our rules, such that the top level rule has an executable Coq analogue. If we disregard admitted lemmas, then the strong normalization of Coq guarantees that the static semantic object can only be built from conforming abstract syntax trees; it also ensures that dynamic executions of HBCL terminate, through the existence of the Coq function that proves the constructability of the dynamic semantics, as given by induction over the predicate characterization of the dynamic semantic rules. We have also seen in our discussion of Figure 5.2 that our choice of a deep embedding style for HBCL over an axiomatization of its ontology gives us a very flexible way to express morphisms of arbitrary properties with relation to arbitrary deep embeddings of languages or physical laws.

Chapter 6

Results of the canonical examples and an illustrative case study

In this chapter, we describe how our first HBCL programs have been built, and present the results of HBCL simulating them with the reference interpreter developed in chapter 5.

The canonical examples have been compiled manually, constructing the relevant static semantic objects by hand. This involved translating the text of HBCL programs into Coq source code, providing an inhabitant of the static semantic object in the semantic domain for each program. Most of the lemmas required to build the static semantic object are omitted. As a consequence, the termination guarantee given by the interpreter function and static semantic object does not strictly hold, since there is always a possibility that the static semantic object may be inconsistent. In the interests of proceeding to executable examples with reasonable speed, we have exported the static semantic objects coded in Coq into OCaml, along with the entire coordination language and expression language. We have then proceeded to debug the resulting programs using the OCaml debugger. The Coq extraction tool is for the most part extremely reliable[1]. However, the semantic mapping of Coq to OCaml module semantics is problematic, unpleasantly verbose, and causes the debugger to have problems with type unification, because the Coq module system is more expressive than the OCaml module system.[2] The conversion of terms in Coq's dependent type system into terms in OCaml's non-dependent type system requires Coq's code export function to use the '`Obj.magic`' cast of OCaml. Although the calls Coq makes to these casts are safe, their presence is another symptom of semantic mismatch. The result of these mismatches is that debugging is more painful than it should be, although we have developed methods to cope with this. The issue is discussed further in section 6.1 and section 7.3.

The process of specifying, formalizing and executing HBCL is shown in Figure 6.1.

[1]Although we did find a bug, which we note in appendix D.1

[2]The Haskell code export facility cannot handle some of our module uses at all.

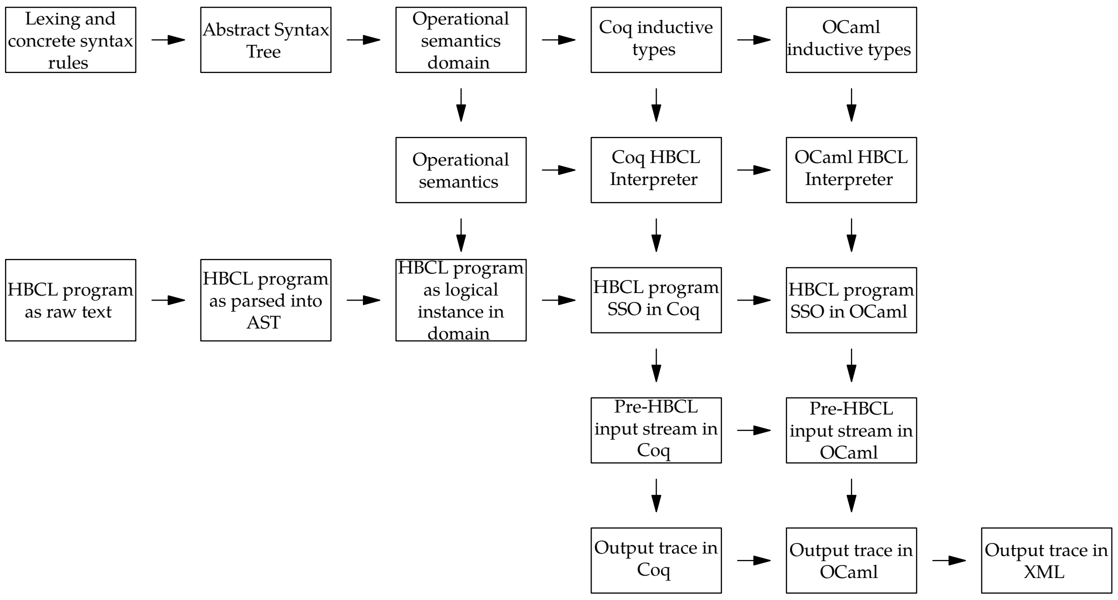


Figure 6.1: Commuting diagram of HBCL formalization and execution

The diagram flows from top left to bottom right. Each column contains the full set of HBCL structures formalized in a particular way. Each row shows the full set of formalization structures as it applies to a single HBCL structure. The HBCL programs in the middle row are hand-compiled as far as the Coq column, before everything in the Coq column is exported to OCaml and linked into a test rig.

Before designing HBCL, we experimented with finite versions of similar example scenarios using the SPIN model-checker [25]. The reference implementation we have produced here works in a similar way to a (naïve) model checker, in that it processes and considers the whole state space on each time step. The crucial difference with a model-checker is that by formalizing HBCL in Coq, we can use Coq to quantify statically over arbitrary infinite domains of HBCL programs using deductive reasoning. Using our deep embedding, we can obtain the soundness of HBCL directly from the soundess of Coq. If we represent particular programs and their static semantic objects in Coq, then we might also reason about the behaviour of these particular programs. If we wanted, we could even reason about programs using model-checking paradigms formalized within Coq, but we could never reason about deductive systems from within model-checkers. Our implementation can always be made more efficient (and just as robust) by proving morphisms, and any deductive conclusions reached about a program with respect to the operational semantics-inspired reference interpretation will then follow for any such implementation refinement.

6.1 Code export and compilation

A small amount of OCaml code has been written to print the values of traces. Since traces are produced by a cofixpoint generating a coinductive type, Coq's code export facilities wrap this with OCaml's special 'lazy' constructors. The print function takes an argument as to how long a trace is required (essential with an infinite input stream) and has to deconstruct the result object carefully to avoid precipitating a non-terminating execution.

The OCaml printing functions are necessary to provide a way to inspect the traces. Coq is a pure logic, and does not handle such side-effects as causing state to be output to a screen. The printing functions produce output data in two modes: a crude text mode and an XML mode. The XML mode is required so that the results can be programmatically filtered using standard tools, since the most complex examples produce extremely large (gigabyte) files, which clearly cannot be printed as they are. The provision of printing functions is particularly difficult for the purposes of debugging the OCaml output by Coq. Since the implementation still contains many admitted lemmas, there were inevitably bugs. Unfortunately, the way that Coq's module system mapped to OCaml's module system made accessing this data extremely difficult. Coq's module system is richer than OCaml's, and OCaml cannot automatically unify types under module functor application in the way Coq can.

This had two difficult consequences for debugging. First, the facility in OCaml's debugger that registers printing functions could not be used for any definition under a functor. Second, to build these printing functions at all, it was necessary to generate patch files to add printing functions to the exported Coq files, as this was the only way to achieve the necessary type unification. This was a time-consuming process, since the debugger could not be used for the task. Every change to a printing function call had to be applied to a patch file, and any change to the Coq code necessitated re-exporting to OCaml and applying these patches. All of this suggests that, if Coq's module system is to be used to its full power (and we have suggested in the previous chapter that the module system may not actually confer all the advantages that first appear), then OCaml is not a very suitable target extraction language. Unfortunately, it is the best supported extraction language in Coq.[3]

The most significant problem we found in exporting code from Coq to OCaml was that we had to treat the coordination and expression languages separately, and instantiate the expression language in the coordination language with OCaml glue code. This seems impossible to excise without either removing the use of modules (the pre-compilation of which reduces the ability of an invoking Coq module to adjust the universe hierarchy), or using explicit type universe polymorphism. This has been proposed by Sozeau

[3]Exporting into Haskell produces a 'not yet implemented'-type exception.

et al. [174], but the idea has yet to be integrated into the main Coq distribution.

The rest of this section presents the traces produced when the interpreter function of chapter 5 is evaluated using the examples' static semantic objects and suitable input streams. We illustrate this hand-compilation process in relation to the examples of chapter 3 in appendix E.

6.2 Trace table production

The trace tables are produced directly from executions of the extracted OCaml, with the traces post-processed by XSLT, TeXML and LaTeX. In the case of the multiplier and replicated multiplier, the traces were so large that a hybrid XSLT and Java StAX approach was adopted. We summarize the examples below.

Program scenario	**Listing**	**Purpose**
Negator of section 3.5.1	Listing 6.1	Tests negator box program: a single box
Parity of section 3.5.2	Listing 6.2	Another test of a single box, needed for the composite examples.
Parallel composition of section 3.5.2	Listing 6.3	A test of the composition of negator and parity boxes without interaction. We consider what the structure of the resulting trace, compared with each trace of the negator and parity boxes individually, tells us about the composition and timing properties of HBCL.
Pipeline of section 3.5.4	Listing 6.4	A test of negator and parity boxes, with negator output fed to parity input
Checksum of section 3.5.3	Listing 6.5	A test of an example with a single box with feedback
Checksum pipeline of section 3.5.5	Listing 6.6	A test of an example with box feedback passing through a further box.

Table 6.1: Conformational examples

We have shown traces in a diagramatic notation, as a series of snapshots of coordination state. Boxes are colour-coded to show which super-step coordination state they are in, according to the following pattern

Colour	Preceding step	Succeeding step
Blue	Box-FIFO memory step	FIFO step
Green	FIFO step	FIFO-box memory step
Yellow	FIFO-box memory step	Box step
Red	Box step	Box-FIFO memory step

Table 6.2: Coordination state colour code

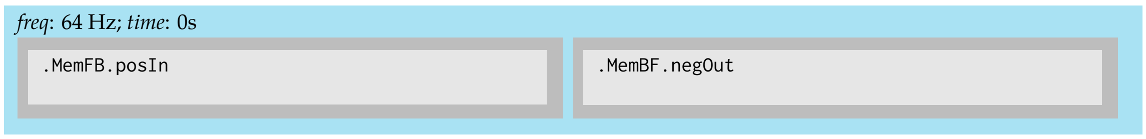
Inside each coordination state, the dark grey box on the left contains input memories, that on the right contains output memories, and a dark grey box straddling the whole width of a coloured box contains nested memories. If, in the case of a particular example, one of these boxes is empty, it is omitted. Coloured boxes show the aggregate (lowest common multiple) frequency of the components it contains, and the current time, at the scope of the box, as a rational number with the denominator corresponding to this frequency. Individual memories are shown as light grey boxes, with their names in the top left. Sequences of time slices within these memories are shown as white boxes within them. They give their frequency, the time to which the value relates, and the data payload. In the nested case, nested instances are shown as white boxes, with the name of the instance in the top left. They contain coordination state boxes which recursively follow the same colour and layout scheme.

All sequences start and end with pre-FIFO step (blue) coordination states. The initial state of the system is a pre-FIFO state because nothing can happen before some input is received. The sequence ends on such states because it is the last state that is fully determinable by the previous FIFO. The coinductive trace object is terminated with the empty post-FIFO step coordination state constructor: this is triggered by a match on a similar end marker for the input stream. While both these data types are coinductive and thus can go on infinitely, this does not rule out finite cases, given that we have in both instances provided a non-inductive final constructor.

6.3 Single box examples

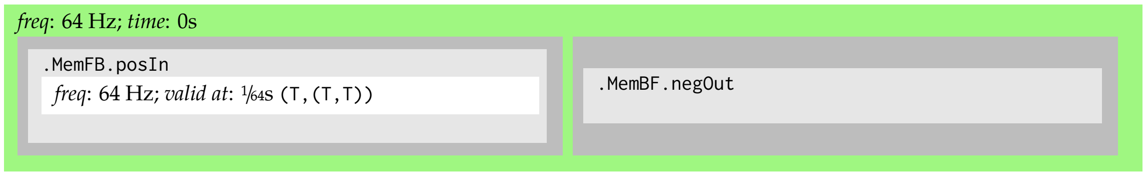
We first consider the negator box and parity box. The parity box is very similar to the negator box, except that it introduces a memory operating at a different frequency.

Listing 6.1(i): The single negator box trace



The blue box shown in Listing 6.1(i) indicates the coordination state at time 0 seconds, for the state between a box-FIFO memory step and a FIFO step. The frequency and time value are shown at the top left of the box. This type of step is always shown as a blue box. Within the blue box are two dark grey boxes: the box on the left is where all the input memories are displayed; the box on the right is where all the output memories appear. Individual memories are shown as lighter grey boxes, with their names appearing at the top left of each memory. Both of these memories are empty during this time slice. The coordination state object also carries a container for nested instances, but we do not show it when it is empty: we shall meet this structure shortly.

Listing 6.1(ii): The single negator box trace (cont.)



freq: 64 Hz; *time*: 0s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $\frac{1}{64}$s `(T,(T,T))`

`.MemBF.negOut`

The green box shown in Listing 6.1(ii) advances the time to the state between a FIFO step and a FIFO-box memory step. The wall clock time has not advanced: it is still at 0 seconds. There is no need to allow time to pass between the four sub-steps of the execution cycle since they are notionally separated by infinitessimally small amounts of time: it is the sequence of steps that is important, and that sequence is always the same. Within Listing 6.1(ii), the `.MemFB.posIn` memory has advanced during the FIFO step, and a new piece of data has been added. This data slice was supplied by the input stream that parametrizes the whole execution. I/O streams are processed on the FIFO step. Where nested instances occur, these exposed streams are either re-exported to the environment or hooked up to new FIFOs outside the scope of the instance. Within the instance, the semantics are the same regardless of whence the input comes or whither the output flows. The memory slice is shown within the memory that accommodates it as a white box on the grey background of the memory of which it is part. Again, the frequency is given, along with a *time of validity* for the piece of data. This time is at a static offset from the current instance time, as determined by the declared time to or from live of the memory, and in the case of this memory, this offset is nil. For data such as Booleans, which do not have an obvious 'meaning' outside an instance, the choice of these offsets is not important, but the relative offsets between these times of validity for a local instance *is* important. The situation is rather different when the data value is strongly typed in the physical environment: in this case times *from* live naturally relate to observed quantities at the system boundary, which become further from 'live' with every passing tick of the clock; times *to* live relate to quantities destined for some actuator where the offset can be thought of as representing a deadline. In the former case, the labelled time of validity is the time at which the observation nominally took place, and in the latter case, the time of validity is the time at which the value should nominally be manifested to the part of the system environment labelled by the relevant OID.

Listing 6.1(iii): The single negator box trace (cont.)

freq: 64 Hz; *time*: 0s

`.MemFB.posIn`
freq: 64 Hz; *valid at*: $\frac{1}{64}$s `(T,(T,T))`

`.MemBF.negOut`

Listing 6.1(iii) shows the coordination state between the FIFO-box step and the box step. These steps are always coloured yellow. Nothing happens at the moment for this step, because there is only one value in the input memory and nothing in the output memory yet. We shall shortly see what happens during this step when the memories are fully populated.

Listing 6.1(iv): The single negator box trace (cont.)

freq: 64 Hz; *time*: $\frac{1}{64}$s

`.MemFB.posIn`
freq: 64 Hz; *valid at*: $\frac{1}{64}$s `(T,(T,T))`

`.MemBF.negOut`
freq: 64 Hz; *valid at*: $\frac{3}{64}$s `(F,F,F)`

Listing 6.1(iv) shows the coordination state between the box step and the next box-FIFO memory step. The clock has advanced to one second for this step, since the box step is notionally the first thing to happen in each time slice. We can also see that there is now some output data. As expected, each output bit is the logical negation of the corresponding input bit. The time of validity is two cycles *beyond* the current time, reflecting the fact that the memory has a time-to-live of two cycles. The temporal semantics of this particular box are fixed such that the output always determines a value whose validity is two cycles in advance of the box's current time.

Listing 6.1(v): The single negator box trace (cont.)

freq: 64 Hz; *time*: $\frac{1}{64}$s

`.MemFB.posIn`
freq: 64 Hz; *valid at*: $\frac{1}{64}$s `(T,(T,T))`

`.MemBF.negOut`
freq: 64 Hz; *valid at*: $\frac{3}{64}$s `(F,F,F)`

In Listing 6.1(v) we can see that nothing has happened, since we only have one value in each memory.

Listing 6.1(vi): The single negator box trace (cont.)

freq: 64 Hz; *time*: $^{1}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,(F,F))`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

In Listing 6.1(vi) a new data slice has been added, with the time of the next box execution as its time of validity (since the time to or from live for this memory is zero). The new value is *prepended* to the list.

Listing 6.1(vii): The single negator box trace (cont.)

freq: 64 Hz; *time*: $^{1}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,(T,T))`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,(F,F))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

Listing 6.1(vii) shows the result of having executed the FIFO-box memory. Now that we have two values in the memory, something can be seen to happen: the order of the values in the memories has changed (this is only visible in the input memory here since the output memory still only has one value). This indicates that the memory has flushed anything that happened to it in the FIFO step and is now ready to be read by a box step. There is nothing canonical about using list reversal to indicate this: it just happens to be convenient to show it this way for the purposes of the reference implementation.

Listing 6.1(viii): The single negator box trace (cont.)

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,(T,T))`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,(F,F))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,T,T)`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

Listing 6.1(viii) shows the coordination state between the box step and the next box-FIFO memory step. The negated value from the latest piece of data has been prepended to the output list with a time of validity of the fourth cycle, which again respects the static time-to-live of two cycles.

Listing 6.1(ix): The single negator box trace (cont.)

freq: 64 Hz; *time*: $^2/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^2/_{64}$s `(T,(F,F))`

freq: 64 Hz; *valid at*: $^1/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^3/_{64}$s `(F,F,F)`

freq: 64 Hz; *valid at*: $^4/_{64}$s `(F,T,T)`

In Listing 6.1(ix) both memories have been reversed. This means that the box-FIFO memory is now ready to be read by a FIFO and the FIFO-box memory is ready to be written by a FIFO.

Listing 6.1(x): The single negator box trace (cont.)

freq: 64 Hz; *time*: $^2/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^3/_{64}$s `(F,(T,T))`

freq: 64 Hz; *valid at*: $^2/_{64}$s `(T,(F,F))`

freq: 64 Hz; *valid at*: $^1/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^3/_{64}$s `(F,F,F)`

freq: 64 Hz; *valid at*: $^4/_{64}$s `(F,T,T)`

In Listing 6.1(x) another new data slice has been added from the input stream. There are now two stale values in the input memory. When the current time goes past the time to or from live-adjusted time of validity of a value, plus the (also statically specified) retention time, then such values are lopped from the list. The need to specify a retention time is a consequence of the passive nature of data transfer in the execution model. Data are observed as opposed to messages passed, the latter being an *active* process on the part of both sender and recipient. The memory declares for how long values will be observable and the FIFO or box implementation is constructed accordingly. The reason we do not just assume that any value will be read immediately it apparently becomes valid is that, in a fully general case, and with state variables for FIFOs,[4] a box or FIFO may execute with a different frequency from the memories to which it is connected, leading to a phase difference in the length of buffer required. In this case, lengths can be statically inferred where both connecting boxes and FIFOs are in scope, while memories on an instance interface would require a kind of temporal polymorphism to accommodate the different frequency range of FIFOs that might be connected. See section 6.8.5 for further discussion on this point.

[4]See the discussion of Listing 6.4(iv) and section 6.9.3.

Listing 6.1(xi): The single negator box trace (cont.)

```
freq: 64 Hz; time: 2/64s
  .MemFB.posIn
    freq: 64 Hz; valid at: 1/64s (T,(T,T))
    freq: 64 Hz; valid at: 2/64s (T,(F,F))
    freq: 64 Hz; valid at: 3/64s (F,(T,T))
  .MemBF.negOut
    freq: 64 Hz; valid at: 4/64s (F,T,T)
    freq: 64 Hz; valid at: 3/64s (F,F,F)
```

Listing 6.1(xi) shows the result of having executed the FIFO-box memory. As in Listing 6.1(vii), this is shown by list reversal.

Listing 6.1(xii): The single negator box trace (cont.)

```
freq: 64 Hz; time: 3/64s
  .MemFB.posIn
    freq: 64 Hz; valid at: 1/64s (T,(T,T))
    freq: 64 Hz; valid at: 2/64s (T,(F,F))
    freq: 64 Hz; valid at: 3/64s (F,(T,T))
  .MemBF.negOut
    freq: 64 Hz; valid at: 5/64s (T,F,F)
    freq: 64 Hz; valid at: 4/64s (F,T,T)
    freq: 64 Hz; valid at: 3/64s (F,F,F)
```

Listing 6.1(xii) shows the results of another box step. The observations follow the same pattern as for Listing 6.1(viii): another value (that valid at three cycles) has been negated and appended to the output memory, with a time value two steps in advance (valid at five cycles). This static offset of 2 cycles reflects the difference in the input memory's time-from-live value (0) and the output memory's time-to-live value (2), as given in the HBCL program.

Listing 6.1(xiii): The single negator box trace (cont.)

```
freq: 64 Hz; time: 3/64s
  .MemFB.posIn
    freq: 64 Hz; valid at: 3/64s (F,(T,T))
    freq: 64 Hz; valid at: 2/64s (T,(F,F))
    freq: 64 Hz; valid at: 1/64s (T,(T,T))
  .MemBF.negOut
    freq: 64 Hz; valid at: 3/64s (F,F,F)
    freq: 64 Hz; valid at: 4/64s (F,T,T)
    freq: 64 Hz; valid at: 5/64s (T,F,F)
```

The sequence ends with Listing 6.1(xiii), where we can see by the list reversal that another box-FIFO memory execution has taken place.

Listing 6.2(i): The single parity box trace

```
freq: 64 Hz; time: 0s
  .MemFB.datIn
  .MemBF.parOut
```

The situation in Listing 6.2(i) is identical to that in Listing 6.1(i), where no data has yet been processed from the input stream.

Listing 6.2(ii): The single parity box trace (cont.)

freq: 64 Hz; *time*: 0s

`.MemFB.datIn`
freq: 64 Hz; *valid at*: 1/64s `(T,T,T)`

`.MemBF.parOut`

The situation in Listing 6.2(ii) is very similar to that in Listing 6.1(ii), except that we see the data type is now a triple of Booleans as opposed to a pair of one Boolean and an inner Boolean pair.

Listing 6.2(iii): The single parity box trace (cont.)

freq: 64 Hz; *time*: 0s

`.MemFB.datIn`
freq: 64 Hz; *valid at*: 1/64s `(T,T,T)`

`.MemBF.parOut`

The situation in Listing 6.2(iii) is identical to that in Listing 6.1(iii).

Listing 6.2(iv): The single parity box trace (cont.)

freq: 64 Hz; *time*: 1/64s

`.MemFB.datIn`
freq: 64 Hz; *valid at*: 1/64s `(T,T,T)`

`.MemBF.parOut`
freq: 128 Hz; *valid at*: 7/128s `((T,T,T),T)`
freq: 128 Hz; *valid at*: 6/128s `((T,T,T),T)`

Listing 6.2(iv) reveals some differences when compared with the situation in Listing 6.1(iv). First, the output data type is clearly different, with the result being a pair containing the original input together with a computed parity bit. Second, the frequency of the data is double that of the input memory and of the box (although the frequency of the *memory* is the same), so the box has to generate two pieces of data (on average, although in this case the number is constant). We have defined the box binding in this instance to put the value of the same expression obtained from the box's input into two consecutive temporal slices of output. The time-to-live requirement is given for the whole memory, and is again two cycles to 'live', so the earliest value must be valid at the current time (1 cycle) plus the time-to-live (two cycles). The value valid at 6 cycles on the higher frequency clock meets this requirement at 3 cycles on the clock of its owning memory.

Listing 6.2(v): The single parity box trace (cont.)

```
freq: 64 Hz; time: 1/64s
  .MemFB.datIn
    freq: 64 Hz; valid at: 1/64s (T,T,T)
  .MemBF.parOut
    freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
    freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
```

Listing 6.2(v) shows the usual list reversal of memory execution for box-FIFO memories, with an effect visible in the output memory, now that it has two values.

Listing 6.2(vi): The single parity box trace (cont.)

```
freq: 64 Hz; time: 1/64s
  .MemFB.datIn
    freq: 64 Hz; valid at: 2/64s (T,F,F)
    freq: 64 Hz; valid at: 1/64s (T,T,T)
  .MemBF.parOut
    freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
    freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
```

A new value is added in the FIFO step of Listing 6.2(vi).

Listing 6.2(vii): The single parity box trace (cont.)

```
freq: 64 Hz; time: 1/64s
  .MemFB.datIn
    freq: 64 Hz; valid at: 1/64s (T,T,T)
    freq: 64 Hz; valid at: 2/64s (T,F,F)
  .MemBF.parOut
    freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
    freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
```

Again, Listing 6.2(vii) shows the expected list reversals, this time happening on the FIFO-box memory step.

Listing 6.2(viii): The single parity box trace (cont.)

```
freq: 64 Hz; time: 2/64s
  .MemFB.datIn
    freq: 64 Hz; valid at: 1/64s (T,T,T)
    freq: 64 Hz; valid at: 2/64s (T,F,F)
  .MemBF.parOut
    freq: 128 Hz; valid at: 9/128s ((T,F,F),T)
    freq: 128 Hz; valid at: 8/128s ((T,F,F),T)
    freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
    freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
```

Another box step is visible in Listing 6.2(viii). The two new output values are again identical, save in their times of temporal validity, which cover the period of the slower owning memory.

Listing 6.2(ix): The single parity box trace (cont.)

freq: 64 Hz; *time*: 2/64s

`.MemFB.datIn`
- *freq*: 64 Hz; *valid at*: 2/64s `(T,F,F)`
- *freq*: 64 Hz; *valid at*: 1/64s `(T,T,T)`

`.MemBF.parOut`
- *freq*: 128 Hz; *valid at*: 6/128s `((T,T,T),T)`
- *freq*: 128 Hz; *valid at*: 7/128s `((T,T,T),T)`
- *freq*: 128 Hz; *valid at*: 8/128s `((T,F,F),T)`
- *freq*: 128 Hz; *valid at*: 9/128s `((T,F,F),T)`

Listing 6.2(ix) shows list reversal, in the same way as Listing 6.2(v) and Listing 6.1(ix).

Listing 6.2(x): The single parity box trace (cont.)

freq: 64 Hz; *time*: 2/64s

`.MemFB.datIn`
- *freq*: 64 Hz; *valid at*: 3/64s `(F,T,T)`
- *freq*: 64 Hz; *valid at*: 2/64s `(T,F,F)`
- *freq*: 64 Hz; *valid at*: 1/64s `(T,T,T)`

`.MemBF.parOut`
- *freq*: 128 Hz; *valid at*: 6/128s `((T,T,T),T)`
- *freq*: 128 Hz; *valid at*: 7/128s `((T,T,T),T)`
- *freq*: 128 Hz; *valid at*: 8/128s `((T,F,F),T)`
- *freq*: 128 Hz; *valid at*: 9/128s `((T,F,F),T)`

Listing 6.2(x) shows that another value has been obtained from the input stream and prependend to the input data list, following the pattern of Listing 6.2(vi) and Listing 6.1(x).

Listing 6.2(xi): The single parity box trace (cont.)

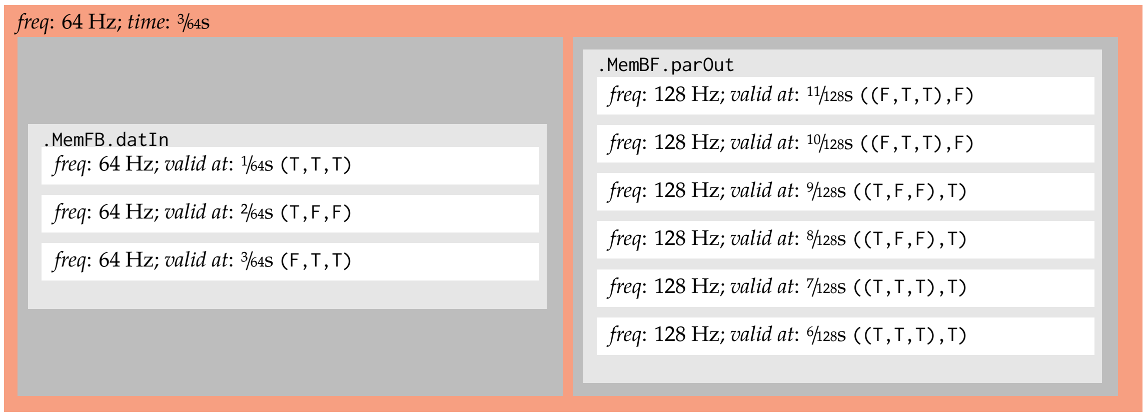
freq: 64 Hz; *time*: 2/64s

`.MemFB.datIn`
- *freq*: 64 Hz; *valid at*: 1/64s `(T,T,T)`
- *freq*: 64 Hz; *valid at*: 2/64s `(T,F,F)`
- *freq*: 64 Hz; *valid at*: 3/64s `(F,T,T)`

`.MemBF.parOut`
- *freq*: 128 Hz; *valid at*: 9/128s `((T,F,F),T)`
- *freq*: 128 Hz; *valid at*: 8/128s `((T,F,F),T)`
- *freq*: 128 Hz; *valid at*: 7/128s `((T,T,T),T)`
- *freq*: 128 Hz; *valid at*: 6/128s `((T,T,T),T)`

Listing 6.2(xi) shows a further memory reversal on FIFO-box memory execution.

Listing 6.2(xii): The single parity box trace (cont.)

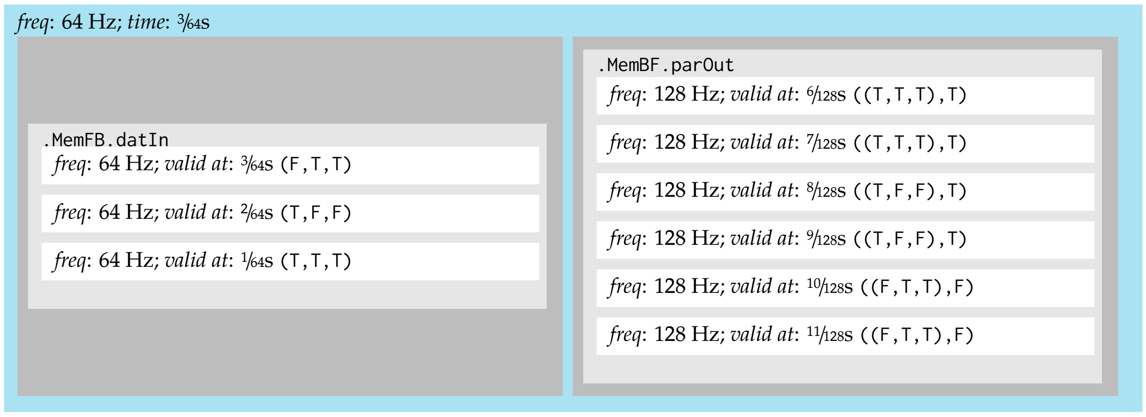

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.MemFB.datIn`
freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,T,T)`
freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,F,F)`
freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,T,T)`

`.MemBF.parOut`
freq: 128 Hz; *valid at*: $^{11}/_{128}$s `((F,T,T),F)`
freq: 128 Hz; *valid at*: $^{10}/_{128}$s `((F,T,T),F)`
freq: 128 Hz; *valid at*: $^{9}/_{128}$s `((T,F,F),T)`
freq: 128 Hz; *valid at*: $^{8}/_{128}$s `((T,F,F),T)`
freq: 128 Hz; *valid at*: $^{7}/_{128}$s `((T,T,T),T)`
freq: 128 Hz; *valid at*: $^{6}/_{128}$s `((T,T,T),T)`

Listing 6.2(xii) shows another set of output values produced as a result of box execution. Again, the time has been incremented immediately before this step.

Listing 6.2(xiii): The single parity box trace (cont.)


freq: 64 Hz; *time*: $^{3}/_{64}$s

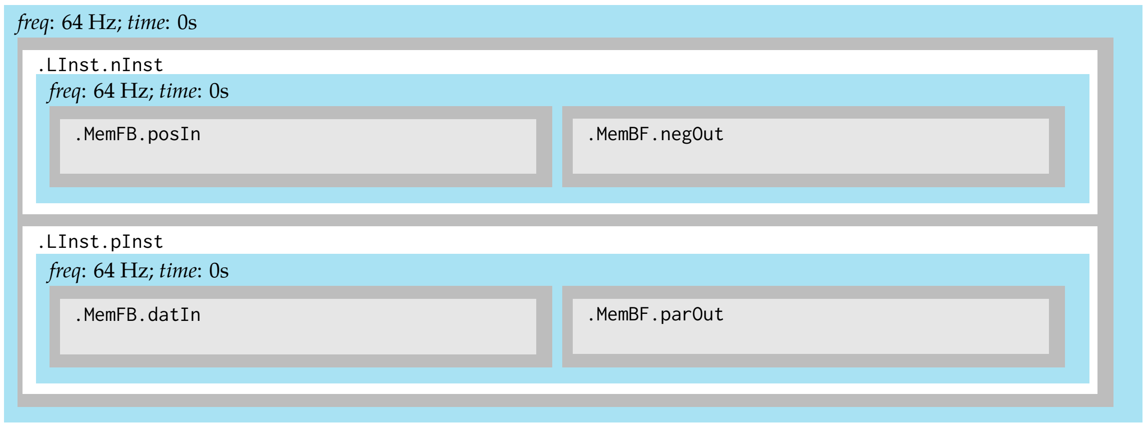
`.MemFB.datIn`
freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,T,T)`
freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,F,F)`
freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,T,T)`

`.MemBF.parOut`
freq: 128 Hz; *valid at*: $^{6}/_{128}$s `((T,T,T),T)`
freq: 128 Hz; *valid at*: $^{7}/_{128}$s `((T,T,T),T)`
freq: 128 Hz; *valid at*: $^{8}/_{128}$s `((T,F,F),T)`
freq: 128 Hz; *valid at*: $^{9}/_{128}$s `((T,F,F),T)`
freq: 128 Hz; *valid at*: $^{10}/_{128}$s `((F,T,T),F)`
freq: 128 Hz; *valid at*: $^{11}/_{128}$s `((F,T,T),F)`

The sequence ends with Listing 6.2(xiii).

6.4 Parallel composition example

Listing 6.3(i): The parallel composition of a negator and parity box

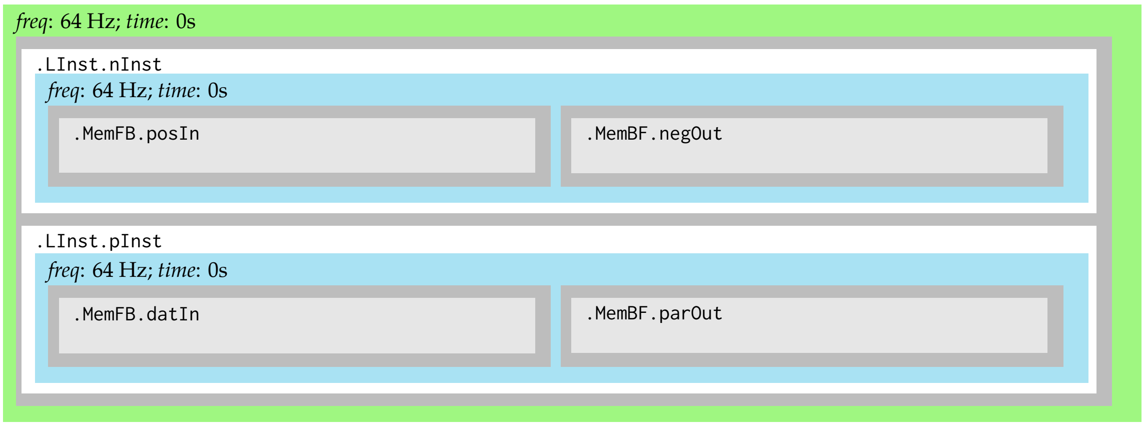

freq: 64 Hz; *time*: 0s

`.LInst.nInst`
freq: 64 Hz; *time*: 0s
`.MemFB.posIn`
`.MemBF.negOut`

`.LInst.pInst`
freq: 64 Hz; *time*: 0s
`.MemFB.datIn`
`.MemBF.parOut`

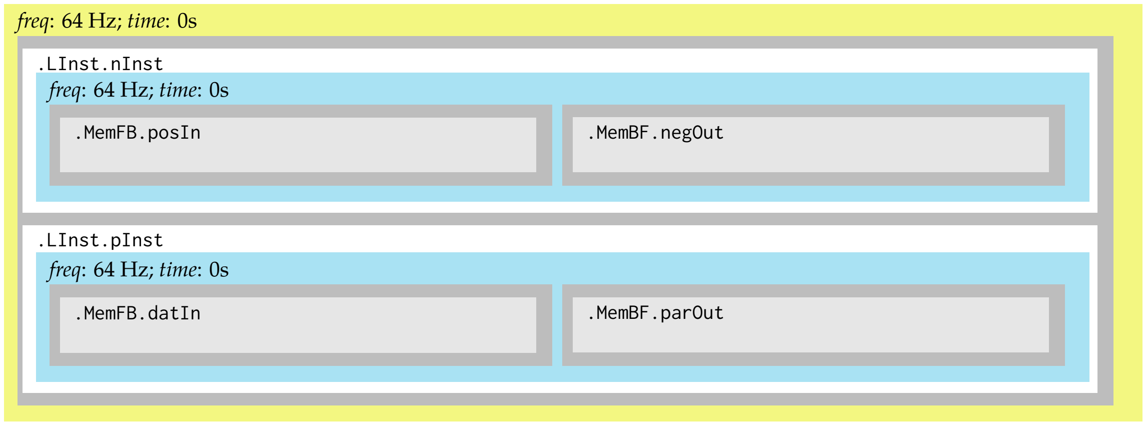
In Listing 6.3(i) we see for the first time a nested instance (or two of them). The columns for input and output memories for the enclosing instances are not shown, since they would be empty. The dark grey box immediately inside the blue box accommodates all the nested instances for the enclosing instance. Each instance appears in a white box, labelled with its local name. Inside the white box is a new nested coordination state object: it has a colour just like the top level boxes. The blue colour highlights the fact that the nested coordination state is also between box-FIFO memory and FIFO steps. The memories inside the negator and parity instances shown are familiar from the stand-alone negator and parity examples: in fact, we reused precisely the same code. The local times of the inner and outer instances can both be seen to be zero.

Listing 6.3(ii): The parallel composition of a negator and parity box (cont.)


freq: 64 Hz; *time*: 0s
.LInst.nInst
freq: 64 Hz; *time*: 0s
.MemFB.posIn
.MemBF.negOut
.LInst.pInst
freq: 64 Hz; *time*: 0s
.MemFB.datIn
.MemBF.parOut

Listing 6.3(ii) shows that the outer step has advanced to the post-FIFO step state, but the states of the inner instances are still at the pre-FIFO stage. Inner instances are only *ever* shown at the pre-FIFO stage. The input stream to the global cofixpoint (at 'enclosing scope') contains data for both nested instances, but it is carried as a collateral argument to the top-level execution steps until the box step. It is during the box step that all four steps of the inner instances are processed. There are two ways to avoid these collateral arguments. First, we could have cached the memory from the environment locally and re-read it when the inner instance executed. This would clutter the state space with duplicate values and extra memory reads and writes that are not part of the semantics, so such an approach is not well fitted to a reference interpreter that is trying directly to implement the operational semantics. Second, we could have updated the inner memory immediately. This is a more attractive solution, but further qualitatively different coordination states would be needed to differentiate between, on the one hand, when all memories have executed *but for* those that are exported to the environment, and on the other, when *all* memories have executed. If we were later to implement mutually recursive heterogeneous coordination languages as we discuss in section 7.4.2, then the requirement for data opacity would win out and we would adopt the first approach. For the sake of clarity, both of these added complexities have been avoided, and we settle for collateral arguments and deferred input memory updates.

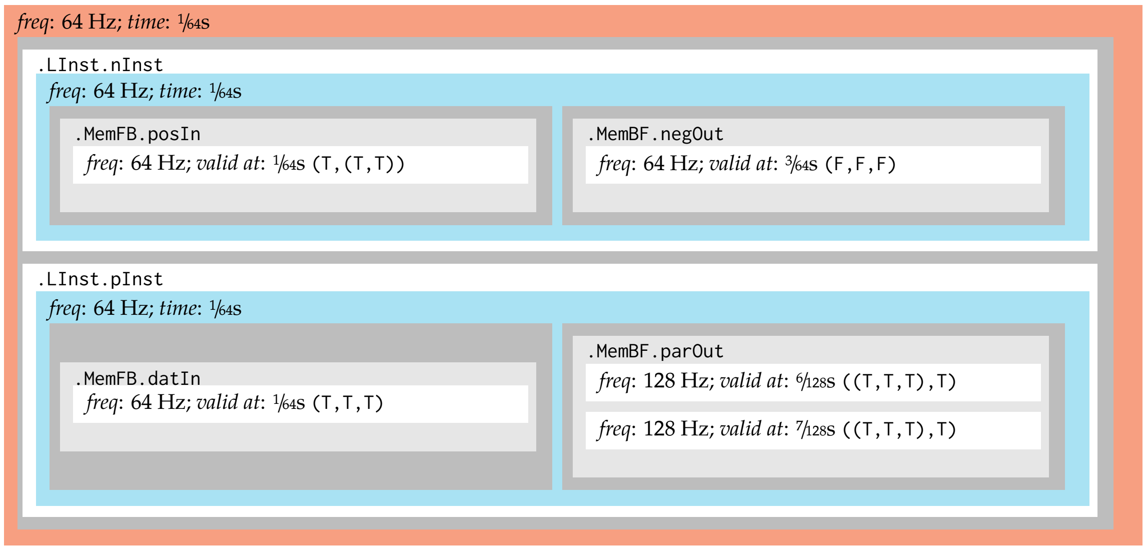
Listing 6.3(iii): The parallel composition of a negator and parity box (cont.)



freq: 64 Hz; *time*: 0s
.LInst.nInst
freq: 64 Hz; *time*: 0s
.MemFB.posIn
.MemBF.negOut
.LInst.pInst
freq: 64 Hz; *time*: 0s
.MemFB.datIn
.MemBF.parOut

In Listing 6.3(iii), it can be seen that nothing happens during the local FIFO-box memory execution step, because there are no local memories. Again, execution of the nested memories is deferred until the nested instances are run.

Listing 6.3(iv): The parallel composition of a negator and parity box (cont.)



freq: 64 Hz; *time*: $1/64$s
.LInst.nInst
freq: 64 Hz; *time*: $1/64$s
.MemFB.posIn
freq: 64 Hz; *valid at*: $1/64$s (T,(T,T))
.MemBF.negOut
freq: 64 Hz; *valid at*: $3/64$s (F,F,F)
.LInst.pInst
freq: 64 Hz; *time*: $1/64$s
.MemFB.datIn
freq: 64 Hz; *valid at*: $1/64$s (T,T,T)
.MemBF.parOut
freq: 128 Hz; *valid at*: $6/128$s ((T,T,T),T)
freq: 128 Hz; *valid at*: $7/128$s ((T,T,T),T)

Listing 6.3(iv) shows the result of the first boxes step at the enclosing scope. There are no local boxes to execute, but the boxes step triggers the execution of the nested instances, which, from the outside, have the same behaviour as any other boxes, save that we allow them to keep state.[5] The result of this execution is that the inner instances advance until they run out of input. Given that the inner instances are operating at the same

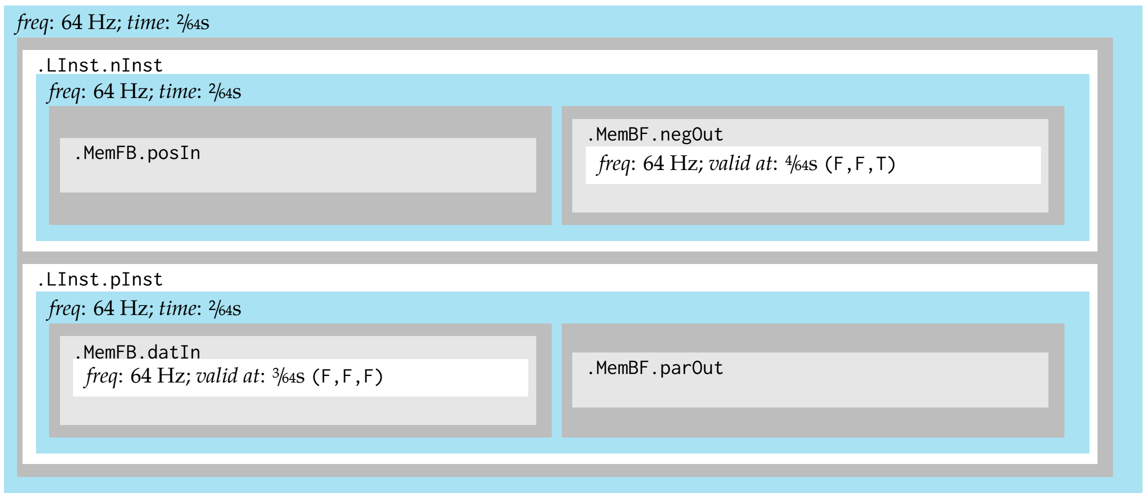
[5]In a fully general case we would allow any boxes to keep state between executions, but we avoid this complication at present.

frequency as the enclosing instance, this means that in this case they run for one fourfold cycle. The state of the inner instances is now identical to the state of the negator box when it was at enclosing scope (at Listing 6.1(v)) and the parity box when it was at enclosing scope (at Listing 6.2(v)). The boxes are not connected, hence no internal FIFOs are declared or have operated in the parallel composition example.

Further cycles of this execution are shown in appendix F.1. By comparing the construction of this trace, and that of the individual traces of the negator and parity boxes, we can see how the parallel composition example maps to each of the negator box example and the parity box example individually. To consider the negator box, we can construct a coinductive predicate over both traces, where there is one propositional constructor, which takes the current coordination state (say, for $t = 0$), for both scenarios, and a proposition that the state of the negator example is equal to the nested negator box in the parallel composition example. We can see by inspection that this property holds for each time slice that we show, and that it also holds for the parity box example.

6.5 Two pipelined boxes example

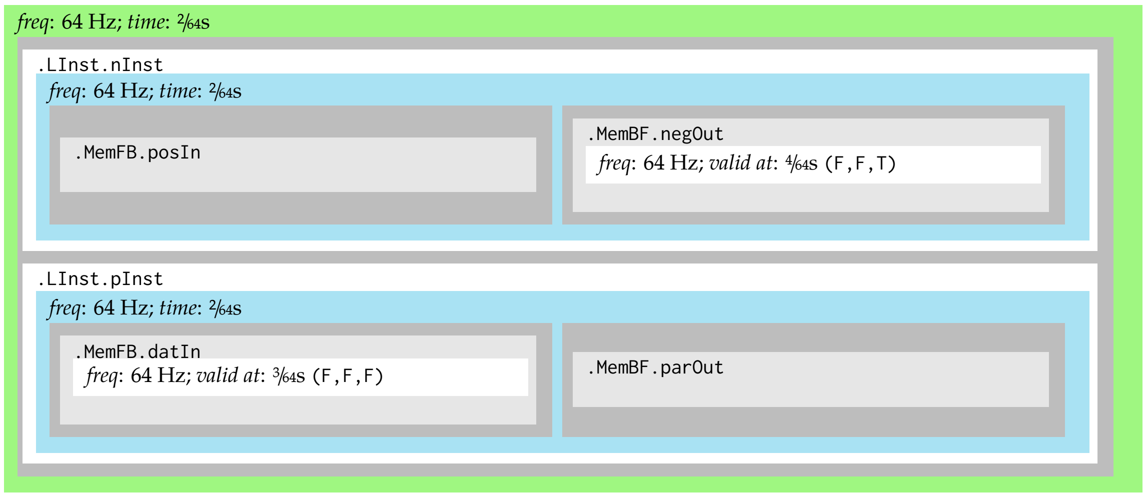
Listing 6.4(i): The pipeline example



freq: 64 Hz; *time*: $^2/_{64}$s
.LInst.nInst
freq: 64 Hz; *time*: $^2/_{64}$s
.MemFB.posIn
.MemBF.negOut
freq: 64 Hz; *valid at*: $^4/_{64}$s (F,F,T)
.LInst.pInst
freq: 64 Hz; *time*: $^2/_{64}$s
.MemFB.datIn
freq: 64 Hz; *valid at*: $^3/_{64}$s (F,F,F)
.MemBF.parOut

In Listing 6.4(i) we see the first example of a FIFO. The scenario has been initialized with some pre-existing data to prevent data starvation. The semantics can deal with starvation, producing null temporal data, but we do not explore this until we reach the realistic examples in section 6.8. We can see that the output of the negator instance contains a value that will be copied to the FIFO on the next FIFO execution, while the input of the parity instance contains a value that will be used by the next parity box execution step. We avoided using data with negative time values, and so we have started this execution with the wall clock set to two elapsed cycles. Nevertheless, negative time values *are* allowed in the model, and we see why they are in fact essential in section 6.8.

Listing 6.4(ii): The pipeline example (cont.)


freq: 64 Hz; *time*: $^2/_{64}$s

.LInst.nInst

freq: 64 Hz; *time*: $^2/_{64}$s

.MemFB.posIn

.MemBF.negOut

freq: 64 Hz; *valid at*: $^4/_{64}$s (F,F,T)

.LInst.pInst

freq: 64 Hz; *time*: $^2/_{64}$s

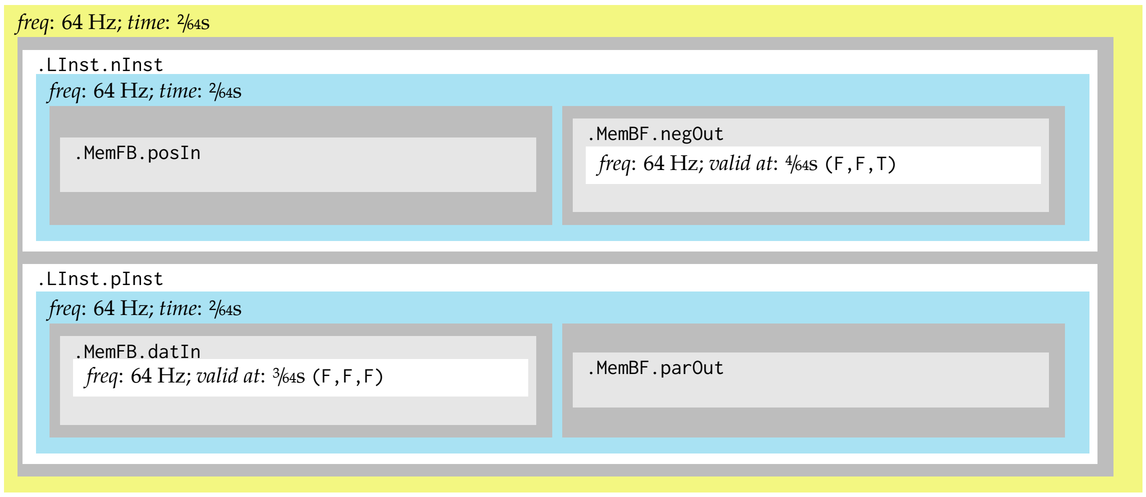
.MemFB.datIn

freq: 64 Hz; *valid at*: $^3/_{64}$s (F,F,F)

.MemBF.parOut

Listing 6.4(ii), like Listing 6.3(ii) shows no change because there are no local boxes and execution of the inner instances' FIFO steps occurs within the nested box execution. Again, input values from the environment are being passed as collateral arguments, but this is not visible in the trace.

Listing 6.4(iii): The pipeline example (cont.)


freq: 64 Hz; *time*: $^2/_{64}$s

.LInst.nInst

freq: 64 Hz; *time*: $^2/_{64}$s

.MemFB.posIn

.MemBF.negOut

freq: 64 Hz; *valid at*: $^4/_{64}$s (F,F,T)

.LInst.pInst

freq: 64 Hz; *time*: $^2/_{64}$s

.MemFB.datIn

freq: 64 Hz; *valid at*: $^3/_{64}$s (F,F,F)

.MemBF.parOut

Again, in Listing 6.4(iii) we do not expect to see anything change.

Listing 6.4(iv): The pipeline example (cont.)

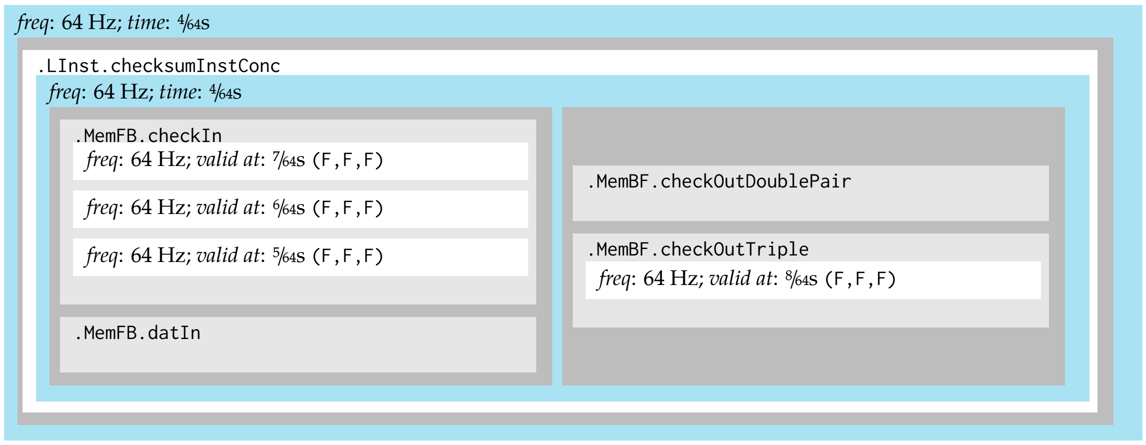
```
freq: 64 Hz; time: 3/64s
  .LInst.nInst
  freq: 64 Hz; time: 3/64s
      .MemFB.posIn
        freq: 64 Hz; valid at: 3/64s (T,(T,T))
      .MemBF.negOut
        freq: 64 Hz; valid at: 4/64s (F,F,T)
        freq: 64 Hz; valid at: 5/64s (F,F,F)
  .LInst.pInst
  freq: 64 Hz; time: 3/64s
      .MemFB.datIn
        freq: 64 Hz; valid at: 4/64s (F,F,T)
        freq: 64 Hz; valid at: 3/64s (F,F,F)
      .MemBF.parOut
        freq: 128 Hz; valid at: 10/128s ((F,F,F),F)
        freq: 128 Hz; valid at: 11/128s ((F,F,F),F)
```

Listing 6.4(iv) shows all the changes that have taken place as a result of the four inner sub-steps of the inner instance executions. The FIFO has copied the next available value from the output of the negator box to the input of the parity box, while a new input value has been read from the environment to feed the input of the negator box. The boxes in both instances have executed, with new computed values being added to the output memories. FIFOs are implemented in a stateless way, with any new value from the FIFO input being immediately copied to the FIFO output upon FIFO execution. This is equivalent to a more realistic semantics that keeps state for FIFOs, thus dealing with the fact that some values may be in transit in the middle of the FIFO during a FIFO tick, and not manifest at either input or output memory. We would anticipate converting the reference semantics to this model in further work, so that resource reasoning about the size of memories can be respected: in the present implementation, input memories have a size without bound depending on the length of the feeding FIFO — a coupling between instance and external FIFO implementation that is undesirable.

Further cycles of this execution are shown in appendix F.2. The pipeline example makes a further useful point about timing. The reason two apparently arbitrary boxes can be composed is because their outward-facing memories that are connected by a FIFO are callibrated according to the *same* clock, regardless of how each of the negator and parity box might be implemented. In constructing an implementation in a discrete substrate, there is perfect temporal alignment. In a more physical implementation, we would need to use the rubber sheet abstraction and interval arithmetic over the probability distributions of clock divergence to construct the FIFO. Nevertheless, because there is a single temporal frame of reference, it is only the timing of this FIFO that an implementation of the two composed instances must concern itself with: the pairwise partial orders of any other memories can be discarded.

6.6 Simple feedback example

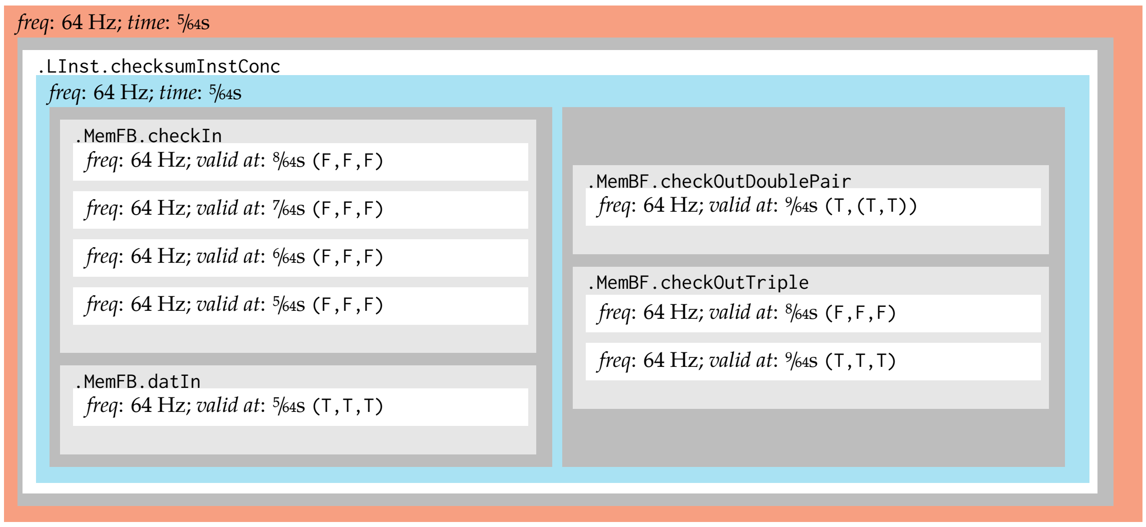
Listing 6.5(i): The simple checksum example



freq: 64 Hz; *time*: $^{4}/_{64}$s
.LInst.checksumInstConc
freq: 64 Hz; *time*: $^{4}/_{64}$s
.MemFB.checkIn
freq: 64 Hz; *valid at*: $^{7}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{6}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{5}/_{64}$s (F,F,F)
.MemFB.datIn
.MemBF.checkOutDoublePair
.MemBF.checkOutTriple
freq: 64 Hz; *valid at*: $^{8}/_{64}$s (F,F,F)

In Listing 6.5(i) we see the initial state of the checksum example. This is the first example with feedback, and has a FIFO connecting `checkOutTriple` to `checkIn`. As with the example in section 6.5, the FIFO state has been pre-initialized, while the other two memories are empty, since no input has been read. The FIFO in this example is longer, as is implicit in the larger difference in times to and from live in the corresponding HBCL program's memory specifications (the difference is now four rather than two cycles). This extra latency allows us to use the same checksum box when we later insert a negator box into the pipeline, without rewriting the boxes.

Listing 6.5(ii): The simple checksum example (cont.)



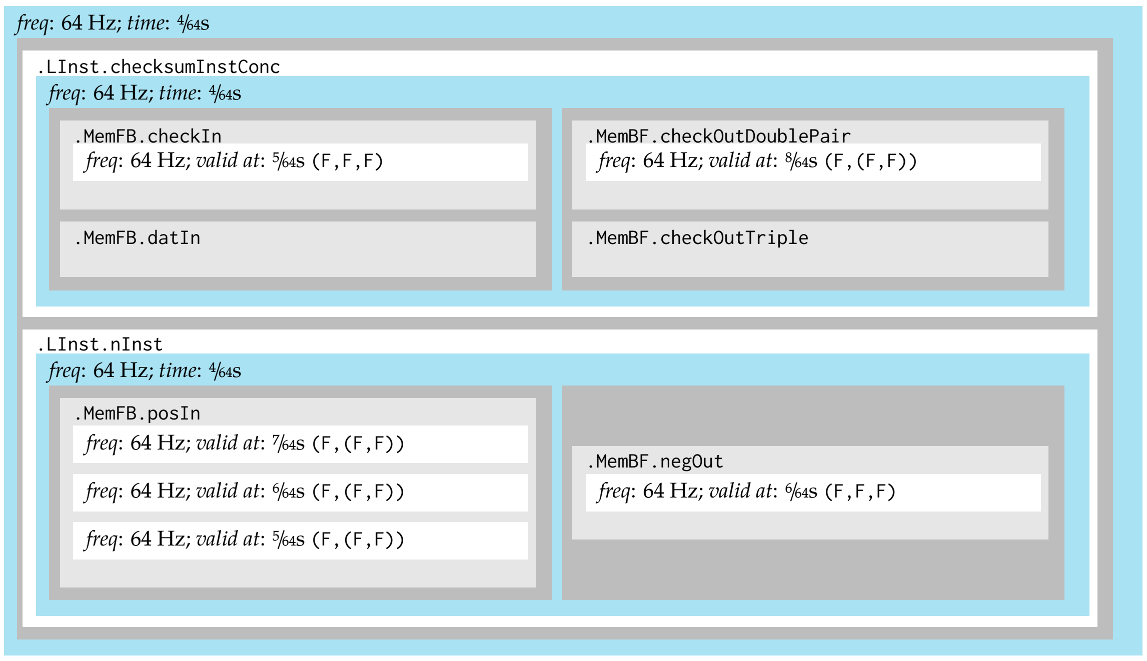
freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.checksumInstConc
freq: 64 Hz; *time*: $^{5}/_{64}$s
.MemFB.checkIn
freq: 64 Hz; *valid at*: $^{8}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{7}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{6}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{5}/_{64}$s (F,F,F)
.MemFB.datIn
freq: 64 Hz; *valid at*: $^{5}/_{64}$s (T,T,T)
.MemBF.checkOutDoublePair
freq: 64 Hz; *valid at*: $^{9}/_{64}$s (T,(T,T))
.MemBF.checkOutTriple
freq: 64 Hz; *valid at*: $^{8}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{9}/_{64}$s (T,T,T)

Listing 6.5(ii) shows the result of a box execution at top level, containing the result of a full four-sub-step cycle at the nested level. It can be seen that the feedback FIFO has operated, with the value that was in the `checkoutTriple` memory being copied to the `checkIn` memory. In the same step, input was read from the environment into `datIn`, and the box executed, producing more checksummed data.

Further execution rounds following the same pattern can be seen in appendix F.3.

6.7 Feedback with pipeline example

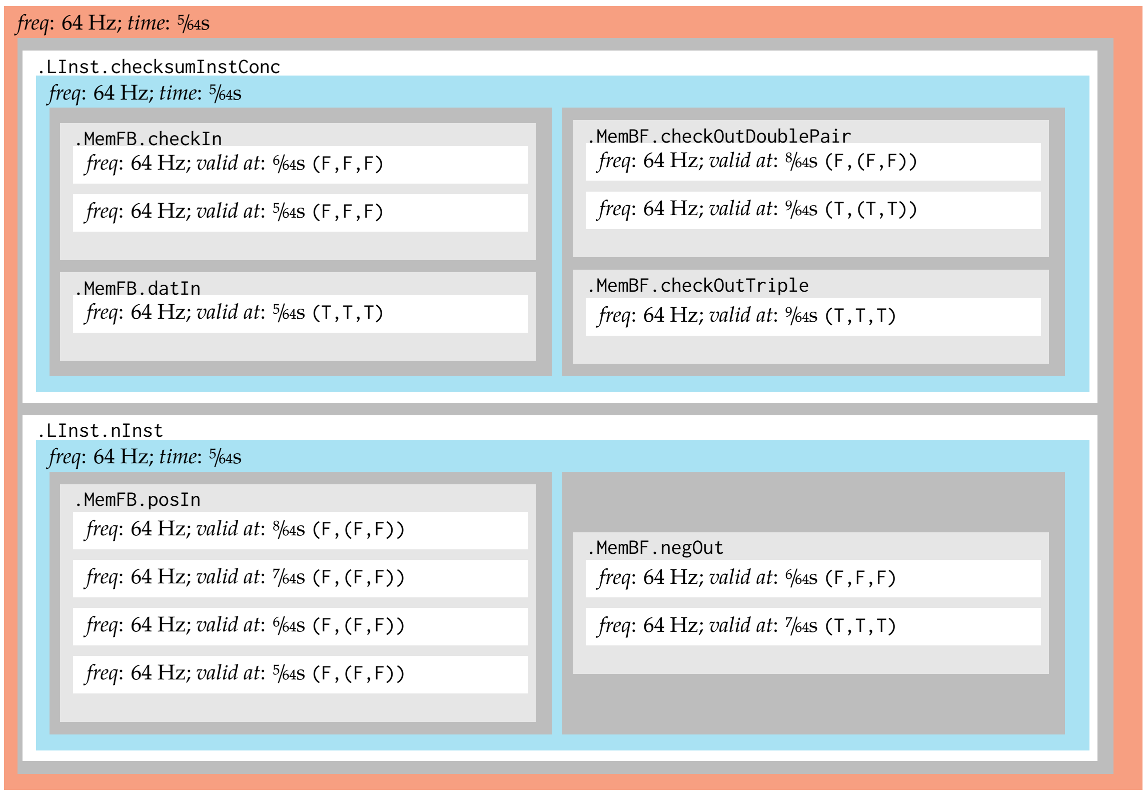
Listing 6.6(i): The checksum pipeline example



freq: 64 Hz; *time*: 4/64s
.LInst.checksumInstConc
freq: 64 Hz; *time*: 4/64s
.MemFB.checkIn
freq: 64 Hz; *valid at*: 5/64s (F,F,F)
.MemBF.checkOutDoublePair
freq: 64 Hz; *valid at*: 8/64s (F,(F,F))
.MemFB.datIn
.MemBF.checkOutTriple
.LInst.nInst
freq: 64 Hz; *time*: 4/64s
.MemFB.posIn
freq: 64 Hz; *valid at*: 7/64s (F,(F,F))
freq: 64 Hz; *valid at*: 6/64s (F,(F,F))
freq: 64 Hz; *valid at*: 5/64s (F,(F,F))
.MemBF.negOut
freq: 64 Hz; *valid at*: 6/64s (F,F,F)

In Listing 6.6(i), we see the first example with two FIFOs. One takes an output of the checksum box and sends it to the negator; the other connects the output of the negator to the feedback input of the checksum. The result is a different checksum to that seen in section 6.6. Both FIFOs have been initialized.

Listing 6.6(ii): The checksum pipeline example (cont.)


freq: 64 Hz; *time*: 5/64s
.LInst.checksumInstConc
freq: 64 Hz; *time*: 5/64s
.MemFB.checkIn
freq: 64 Hz; *valid at*: 6/64s (F,F,F)
freq: 64 Hz; *valid at*: 5/64s (F,F,F)
.MemFB.datIn
freq: 64 Hz; *valid at*: 5/64s (T,T,T)
.MemBF.checkOutDoublePair
freq: 64 Hz; *valid at*: 8/64s (F,(F,F))
freq: 64 Hz; *valid at*: 9/64s (T,(T,T))
.MemBF.checkOutTriple
freq: 64 Hz; *valid at*: 9/64s (T,T,T)
.LInst.nInst
freq: 64 Hz; *time*: 5/64s
.MemFB.posIn
freq: 64 Hz; *valid at*: 8/64s (F,(F,F))
freq: 64 Hz; *valid at*: 7/64s (F,(F,F))
freq: 64 Hz; *valid at*: 6/64s (F,(F,F))
freq: 64 Hz; *valid at*: 5/64s (F,(F,F))
.MemBF.negOut
freq: 64 Hz; *valid at*: 6/64s (F,F,F)
freq: 64 Hz; *valid at*: 7/64s (T,T,T)

In Listing 6.6(ii), both boxes and both FIFOs have executed, in just the same way as in section 6.6. In these FIFO examples, it is apparent that the times to and from live for each memory can become difficult to manage, since to have them correspond with what we intuitively expect, we either have to hard-code offsets to ensure that values flow without having their temporal component abruptly reset, or have an output that mysteriously produces a value that is less old than its input. The latter approach has been adopted so far and is not entirely satisfactory; nor is the former solution completely adequate, because it means that boxes cannot be reused without trivial re-writes. The resolution of this problem is time-shifting semantics, whereby instances have their wall-clock times permanently displaced from the enclosing instance. This has been implemented for HBCL, and we meet this behaviour in the scalable examples of section 6.8. One can envisage a further development of this whereby time can also be *multiplied* by constants, compressing the latency difference between input and output. We leave this to further work.

Further execution rounds following the same pattern can be seen in appendix F.4.

6.8 More involved examples

We now present a set of progressive examples leading to a reasonably complex case involving three replicated multipliers operating in parallel, with fan-outs and voting. The use of HBCL to represent hardware in this way specifies an architecture in which gate propagation delay is amortized by a clocked flip-flop. In this sense, HBCL in combination with the bit field expression language might be thought of as an architecture-neutral FPGA specification mechanism, where we restrict the logic cells of the FPGA to synchronous interactions. We also meet the time-shifting semantics of whole instances for the first time. This is observable in the differing wall-clock time of nested instances when compared to their enclosing instances. Internally to an instance, semantics are invariant with respect to this local wall-clock time. Externally, the temporal expectations of a memory are shifted by the static displacement of the instance. In printing the time values of nested instances, the printing functions re-construct the time using the same OCaml code (exported from Coq) that resolves displacement within the interpreter. Time is not stored in the coordination object itself because it is superfluous information: it is statically and intrinsically typed into the dependent type of the coordination object, and can be fully inferred from the static semantic object for the program and the wall-clock time of the top-level instance.

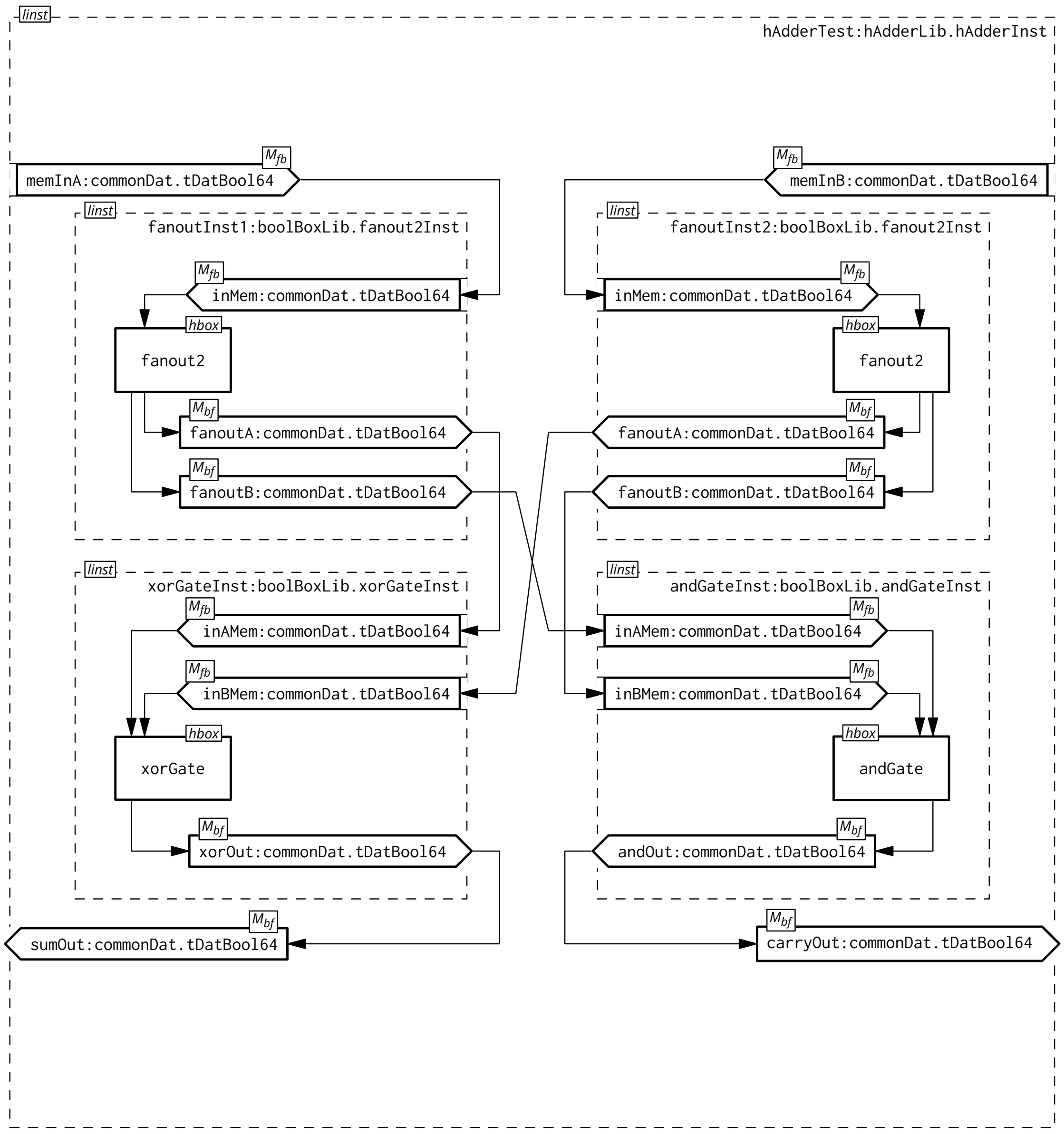
The examples below all use a library of boxes implementing some basic gates and fan-out operations. The instances with logical operations such as AND and OR in their names implement this operation with their inputs as the arguments to the operations in question and their single output representing the outcome of the function. The fan-outs copy their inputs to each set of outputs. We list the library containing these instances in appendix E.1.

6.8.1 Half adder

The HBCL code for the half adder is given in Listing 6.7. It is shown diagramatically in Figure 6.2. The adder sums binary digits, and has the following truth table:

`memInA`	`memInB`	`sumOut`	`carryOut`
0	0	0	0
0	1	1	0
1	0	1	0
0	1	1	1

Table 6.3: Half adder truth table



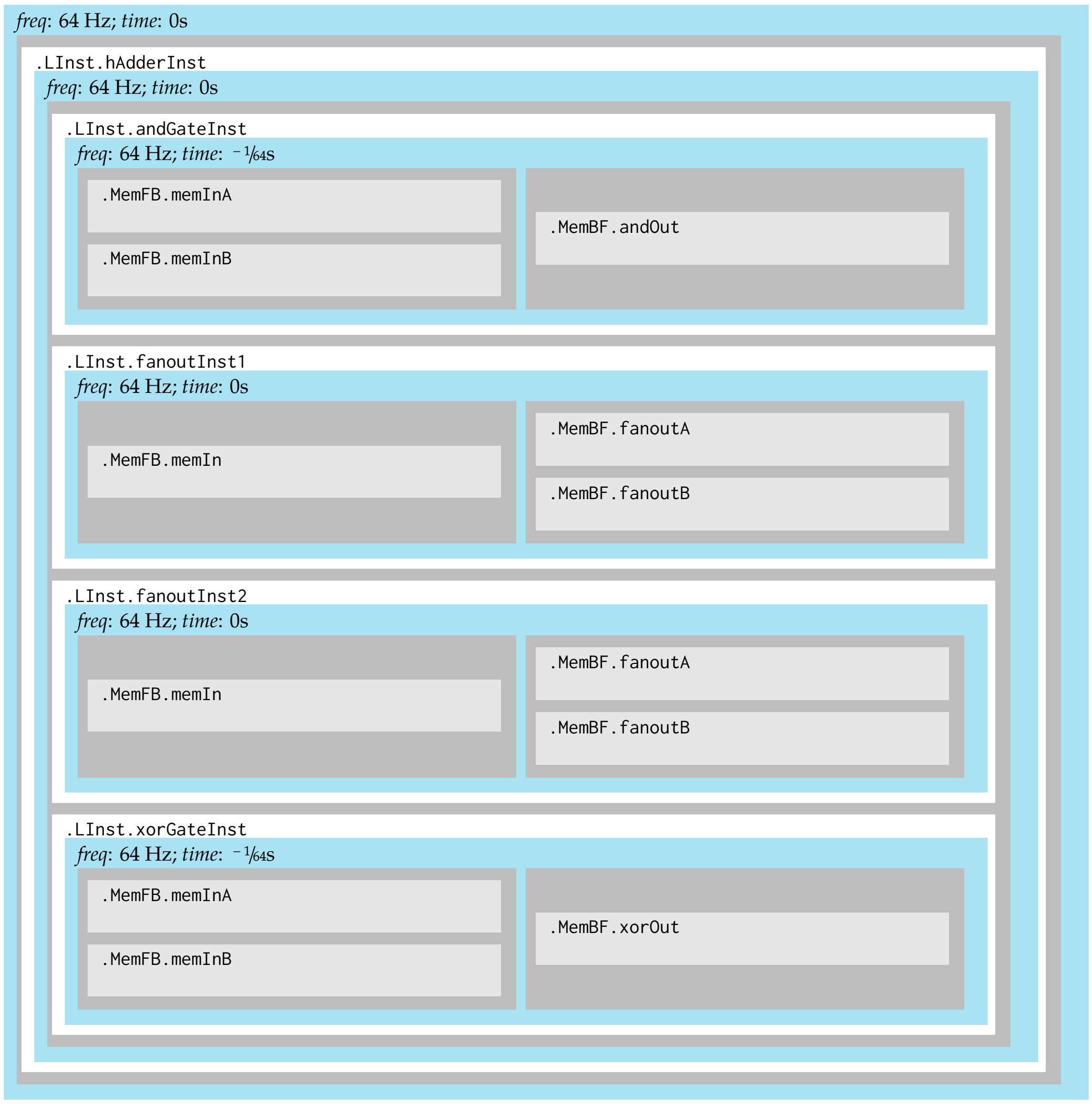
linst
hAdderTest:hAdderLib.hAdderInst
M_{fb}
memInA:commonDat.tDatBool64
memInB:commonDat.tDatBool64
fanoutInst1:boolBoxLib.fanout2Inst
fanoutInst2:boolBoxLib.fanout2Inst
inMem:commonDat.tDatBool64
hbox
fanout2
M_{bf}
fanoutA:commonDat.tDatBool64
fanoutB:commonDat.tDatBool64
xorGateInst:boolBoxLib.xorGateInst
andGateInst:boolBoxLib.andGateInst
inAMem:commonDat.tDatBool64
inBMem:commonDat.tDatBool64
xorGate
andGate
xorOut:commonDat.tDatBool64
andOut:commonDat.tDatBool64
sumOut:commonDat.tDatBool64
carryOut:commonDat.tDatBool64

Figure 6.2: Half adder

Listing 6.7: The half adder

```
llib hAdderLib

  linst hAdderInst {

    linst fanoutInst1 : boolBoxLib.fanout2Inst;
    linst fanoutInst2 : boolBoxLib.fanout2Inst;
    linst xorGateInst : boolBoxLib.xorGateInst(1);
    linst andGateInst : boolBoxLib.andGateInst(1);

    observe {
      fanoutInst1.memIn as memInA;
      fanoutInst2.memIn as memInB;
    }

    manifest {
      xorGateInst.xorOut as sumOut;
      andGateInst.andOut as carryOut;
    }

    fifo fanoutInst1.fanoutA to xorGateInst.memInA;
    fifo fanoutInst2.fanoutA to xorGateInst.memInB;
    fifo fanoutInst1.fanoutB to andGateInst.memInA;
    fifo fanoutInst2.fanoutB to andGateInst.memInB;

  }
}
```

Listing 6.8(i): The half adder example



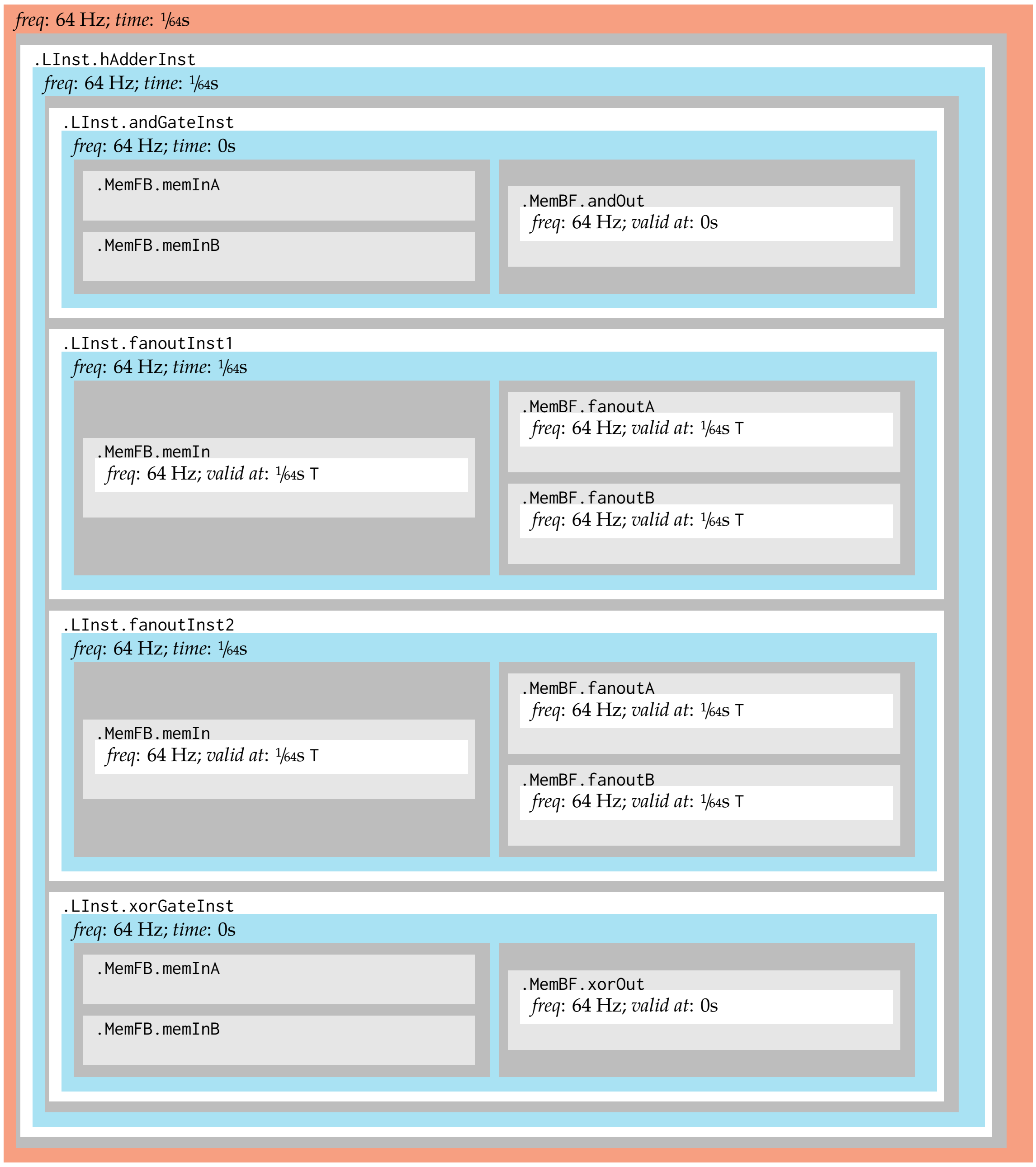
In Listing 6.8(i) can be seen the entire state space of the half adder. For the first time in this example, we have not initialized the state space, so it appears that starvation will occur when boxes and FIFOs operate. This does not happen, because FIFOs tolerate missing data by ignoring it, and boxes generate null outputs if inputs are missing or null.[6] This gives rise to two desirable properties: first, the often complex systems of FIFOs are self initializing; second, such semantics lend themselves to a development where we might simulate faults in which data is lost. There is no danger of such semantics producing nonsensical values, since the production of a null value is not a value at all, and the length of time a scenario takes to initialize or recover from a transient fault is a static

[6]The precise behaviour of boxes in these cases depends on the binding of the relevant box language to the coordination language.

property of the system. This behaviour has a similar effect to the ⋆-matching pattern of Hume. The HBCL semantics have a different emphasis, however, because HBCL only expects to match nulls under system start-up or fault conditions. Our expression language is not aware of nulls. Our binding of the expression language to the HBCL coordination language, through our untimed box language abstraction, does not call the expression if a null input is present, but instead immediately generates a null output, short-circuiting the expression language altogether. The HBCL binding framework is flexible enough to accommodate language bindings where the untimed box language *does* deal with null values, and does so more gracefully. More elaborate embedded untimed box languages could make use of this.

We can also see in Listing 6.8(i) that the AND and XOR gates have *negative* wall-clock times. This follows from the fact that they have a time *from* live, and the static offset for a time-from-live at time zero is negative.

Listing 6.8(ii): The half adder example (cont.)



freq: 64 Hz; *time*: 1/64s
.LInst.hAdderInst
freq: 64 Hz; *time*: 1/64s
.LInst.andGateInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
.MemFB.memInB
.MemBF.andOut
freq: 64 Hz; *valid at*: 0s
.LInst.fanoutInst1
freq: 64 Hz; *time*: 1/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1/64s T
.LInst.fanoutInst2
freq: 64 Hz; *time*: 1/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1/64s T
.LInst.xorGateInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
.MemFB.memInB
.MemBF.xorOut
freq: 64 Hz; *valid at*: 0s

Listing 6.8(ii) shows the situation after one round of sub-steps, including the nested box step with the top-level clock at one cycle. The gates can be seen to have produced null values as expected, while the fan-out boxes have asserted the correct outputs, ready for FIFO execution in the next step.

Listing 6.8(iii): The half adder example (cont.)

```
freq: 64 Hz; time: 2/64s
  .LInst.hAdderInst
  freq: 64 Hz; time: 2/64s
    .LInst.andGateInst
    freq: 64 Hz; time: 1/64s
      .MemFB.memInA
      freq: 64 Hz; valid at: 1/64s T
      .MemFB.memInB
      freq: 64 Hz; valid at: 1/64s T
                                        .MemBF.andOut
                                        freq: 64 Hz; valid at: 1/64s T
    .LInst.fanoutInst1
    freq: 64 Hz; time: 2/64s
                                        .MemBF.fanoutA
      .MemFB.memIn                      freq: 64 Hz; valid at: 2/64s F
      freq: 64 Hz; valid at: 2/64s F
      freq: 64 Hz; valid at: 1/64s T    .MemBF.fanoutB
                                        freq: 64 Hz; valid at: 2/64s F
    .LInst.fanoutInst2
    freq: 64 Hz; time: 2/64s
                                        .MemBF.fanoutA
      .MemFB.memIn                      freq: 64 Hz; valid at: 2/64s F
      freq: 64 Hz; valid at: 2/64s F
      freq: 64 Hz; valid at: 1/64s T    .MemBF.fanoutB
                                        freq: 64 Hz; valid at: 2/64s F
    .LInst.xorGateInst
    freq: 64 Hz; time: 1/64s
      .MemFB.memInA
      freq: 64 Hz; valid at: 1/64s T
                                        .MemBF.xorOut
                                        freq: 64 Hz; valid at: 1/64s F
      .MemFB.memInB
      freq: 64 Hz; valid at: 1/64s T
```

Listing 6.8(iii) shows the next execution round, in which the FIFO step at one cycle and boxes step at two cycles have taken place. The fan-out outputs have been copied to the gate inputs, and this time the gates have been able to operate on some real data, to produce non-null outputs. The XOR output gives a sum value of zero, while the AND output gives a carry output of one. The half adder thus gives the expected output of the input data (namely both bits one), that is zero in the units and carry one. The answer to the sum has a 'valid at' time of one cycle, the same as the input data two cycles earlier with the same time of one cycle. The time-shifting semantics of boxes have enabled us to give sensible meanings to these abstract data. While different parts of a coherent set of output data may emerge from different sub-instances and at different times, the offsets between these times are constant. In the case of our arithmetic examples, these outputs are the various significant bits of a binary number. Since the input memories of harmonic boxes are also statically determined, we know immediately how these outputs relate to previous inputs.

Further execution rounds are shown in appendix F.5. Data from following time slices produces the expected results. For example, we can see for the data valid at two cycles that zero and zero make zero with zero carry.

6.8.2 Full adder

Listing 6.9: The full adder

```
llib fAdderLib

  linst fAdderInst {

    linst hAdderInst1 : hAdderLib.hAdderInst;
    linst hAdderInst2 : hAdderLib.hAdderInst(2);
    linst orGateInst : boolBoxLib.orGateInst(4);

    observe {
      hAdderInst1.memInA as memInA;
      hAdderInst1.memInB as memInB;
      hAdderInst2.memInB as memInCarry;
    }

    manifest {
      hAdderInst2.sumOut as sumOut;
      orGateInst.orOut as carryOut;
    }

    fifo hAdderInst1.sumOut to hAdderInst2.memInA;
    fifo hAdderInst1.carryOut to orGateInst.memInA;
    fifo hAdderInst2.carryOut to orGateInst.memInB;

  }
}
```

The full adder is the last example for which it is practical to show the full state space in graphical form: it can be found in appendix F.6. The arithmetic works as expected, according to the following truth table.

memInCarry	memInA	memInB	sumOut	carryOut
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 6.4: Full adder truth table

The headings in the truth table correspond to the observed and manifested values of the HBCL program. The full adder is the first example having nested instances that are themselves nested. We show the conformation in Figure 6.3, with half adders instantiated. We do not show the inside of the half adders: only its functional specification is important. The time shift semantics (shown by the figures in brackets after the logical instance definitions of lines 6 and 7) produce the expected results, with each instance appearing to exist in a world with its own definitive wall-clock time. The instance has knowledge of the time shifts of instances nested within itself, but is ignorant of how its own time frame is mediated by any intervening boxes between itself and the real wall-clock of international atomic time (or the simulated global time of our simulation). This effect can be seen in the `hAdderInst2` instance within each full adder coordination object.

This example is only well-formed when inside a closure in which a half adder with the required signature is defined. The whole example is compiled at the same library scope as the half adder library, which has the effect that the implementation resolution routine finds the half adder implementation of section 6.8.1.

Other implementations would be possible, for example, one in which the entire computation was done in a single expression step in only one box. If such an implementation were placed between any other half adder implementation in the library hierarchy and the gates-as-boxes example, then it would be the implementation that would be implicit in the program.

This kind of resolution technique is quite unusual in that it does not follow the 'import'-style directives of many programming languages for dealing with structure and selecting implementations. The choice is deliberate, because it is envisaged that once a program is uploaded to an ontology of programs, this (monotone) process results in a

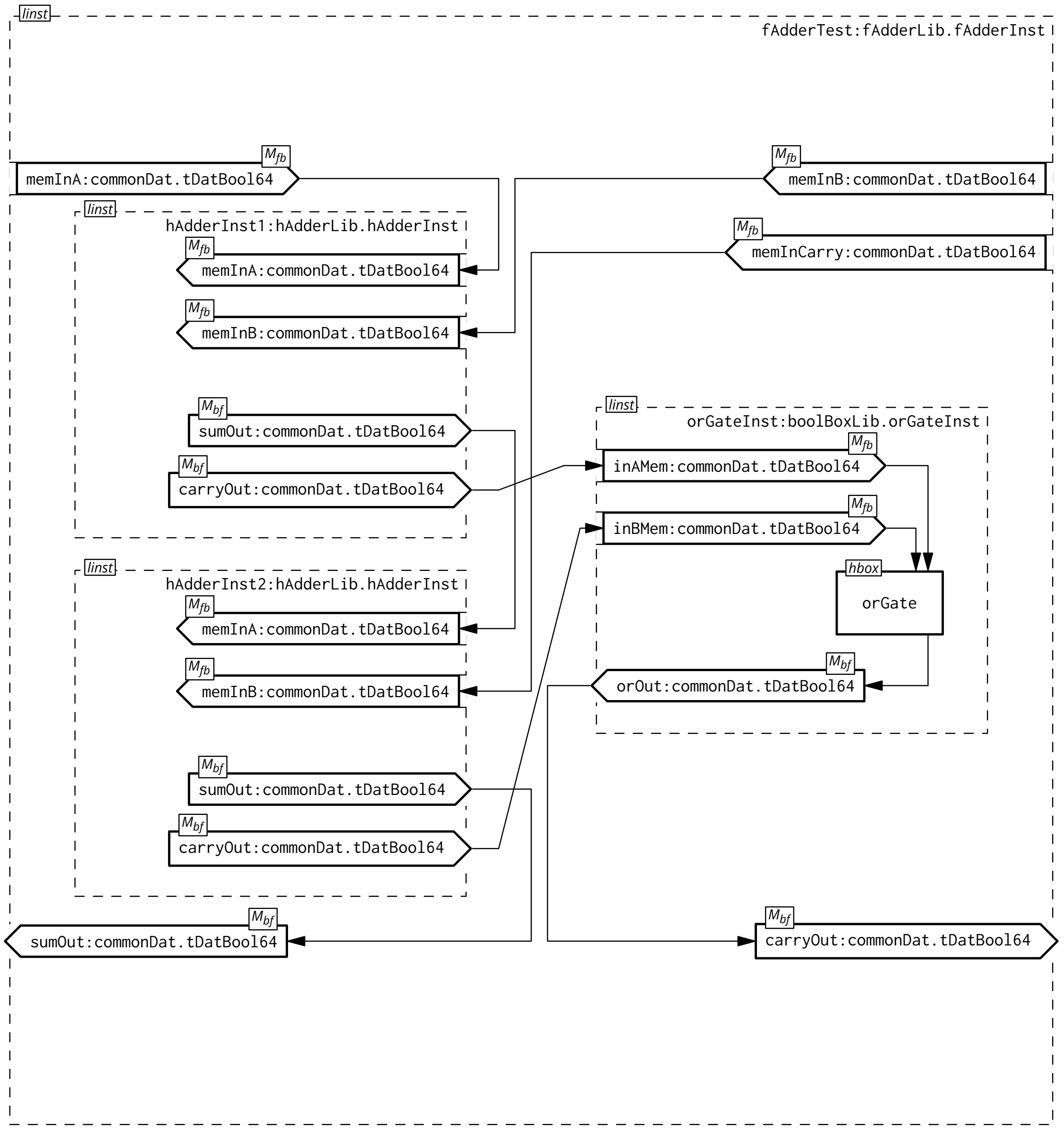

linst
fAdderTest:fAdderLib.fAdderInst
Mfb
memInA:commonDat.tDatBool64
Mfb
memInB:commonDat.tDatBool64
Mfb
memInCarry:commonDat.tDatBool64
linst
hAdderInst1:hAdderLib.hAdderInst
Mfb
memInA:commonDat.tDatBool64
Mfb
memInB:commonDat.tDatBool64
Mbf
sumOut:commonDat.tDatBool64
Mbf
carryOut:commonDat.tDatBool64
linst
orGateInst:boolBoxLib.orGateInst
Mfb
inAMem:commonDat.tDatBool64
Mfb
inBMem:commonDat.tDatBool64
hbox
orGate
Mbf
orOut:commonDat.tDatBool64
linst
hAdderInst2:hAdderLib.hAdderInst
Mfb
memInA:commonDat.tDatBool64
Mfb
memInB:commonDat.tDatBool64
Mbf
sumOut:commonDat.tDatBool64
Mbf
carryOut:commonDat.tDatBool64
Mbf
sumOut:commonDat.tDatBool64
Mbf
carryOut:commonDat.tDatBool64

Figure 6.3: Full adder

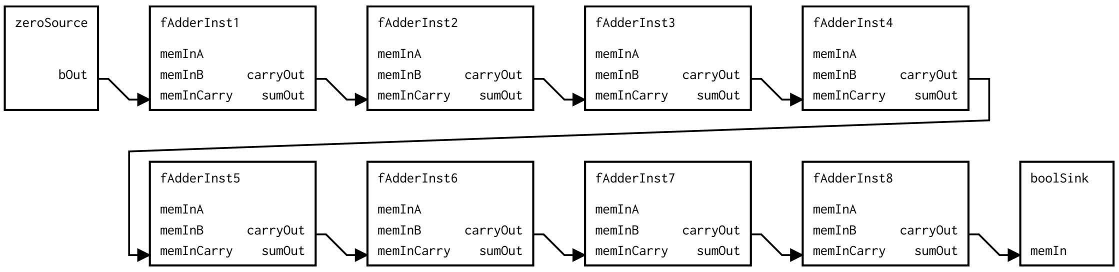


Figure 6.4: Structure of a cascade adder

program that is both immutable and not reliant on finding a root namespace in which to resolve something. Library closures are formed when no instances within that library refer to definitions that require any ancestors in the library tree.

6.8.3 Cascade adder

The cascade adder strings together eight full adders in the fashion illustrated in Figure 6.4. We can see how each of the eight bits corresponding to an eight-bit binary number feed their output carry bit to the input carry bit of the next (and more significant) bit. The first carry is supplied with a source of zeros, and the last carry is connected to a sink of Boolean values. HBCL memories must always be connected on both sides. The least significant bits are in the full adder instances with the lowest numbers. The aim is not to construct an efficient adder (cascade adders have poor latency) but to show an eightfold increase in the scale of a recognizable example. Important parts of the trace for the cascade adder are shown in appendix F.7. The listing shows the input states after one cycle. These can be seen as the two input memories of the first half adder of each full adder. These values are followed by the states of the sum and carry outputs as they emerge at five cycle intervals over 40 cycles. These static offsets can be seen in the full adder instantiations from lines 5 to 14 in Listing 6.10. The names correspond with the boxes in Figure 6.4. The observed and manifested memories specified in lines 16 to 29 show the aliases with which each instance is observed or manifested. To maintain the clarity of Figure 6.4, we have not added these export aliases to the diagram. As with the half and full adders, matching of inputs to outputs is evident in the fact that 'valid at' values are the same for a particular calculation notwithstanding the cascaded delays of more significant bits as they wait for the less significant carry bit to become available. It can be seen in Listing F.7 that the calculation for the value input after one cycle is 01111111 + 10000000 = 11111111, or 127 + 128 = 255 in base 10. The outputs are read from the XOR gate outputs of the second half adder of each full adder. The selection of boxes shown in Listing F.7 was obtained by filtering the boxes of interest from the output trace using an XSLT stylesheet, followed by generation of the tables using further stylesheets to produce LaTeX *via* TeXML .

Listing 6.10: The cascade adder

```
llib cAdderLib

  linst cAdderInst {

    linst fAdderInst1 : fAdderLib.fAdderInst;
    linst fAdderInst2 : fAdderLib.fAdderInst(5);
    linst fAdderInst3 : fAdderLib.fAdderInst(10);
    linst fAdderInst4 : fAdderLib.fAdderInst(15);
    linst fAdderInst5 : fAdderLib.fAdderInst(20);
    linst fAdderInst6 : fAdderLib.fAdderInst(25);
    linst fAdderInst7 : fAdderLib.fAdderInst(30);
    linst fAdderInst8 : fAdderLib.fAdderInst(35);
    linst zeroSource : sourceSinkLib.source0Inst(1);
    linst boolSink : sourceSinkLib.sinkInst(40);

    observe {
      fAdderInst1.memInA as memIn1A;
      ...
      fAdderInst8.memInA as memIn8A;
      fAdderInst1.memInB as memIn1B;
      ...
      fAdderInst8.memInB as memIn8B;
    }

    manifest {
      fAdderInst1.sumOut as sumOut1;
      ...
      fAdderInst8.sumOut as sumOut8;
    }

    fifo zeroSource.bOut to fAdderInst1.memInCarry;
    fifo fAdderInst1.carryOut to fAdderInst2.memInCarry;
    ...
    fifo fAdderInst7.carryOut to fAdderInst8.memInCarry;
    fifo fAdderInst8.carryOut to boolSink.memIn;

  }
}
```

6.8.4 Multiplier

The structure of the multiplier is illustrated in Figure 6.5. Each arrow represents a separate FIFO. The multiplier makes use of source and sink boxes, given in the library of Boolean source and sink boxes listed in appendix E.2. The multiplier also uses a feature that we have not yet seen, namely a library nested within the scope of an instance. This has the effect on making that library visible only to the local instance and any other libraries or instances declared within it. We make use of this feature for 8-fold fan-out

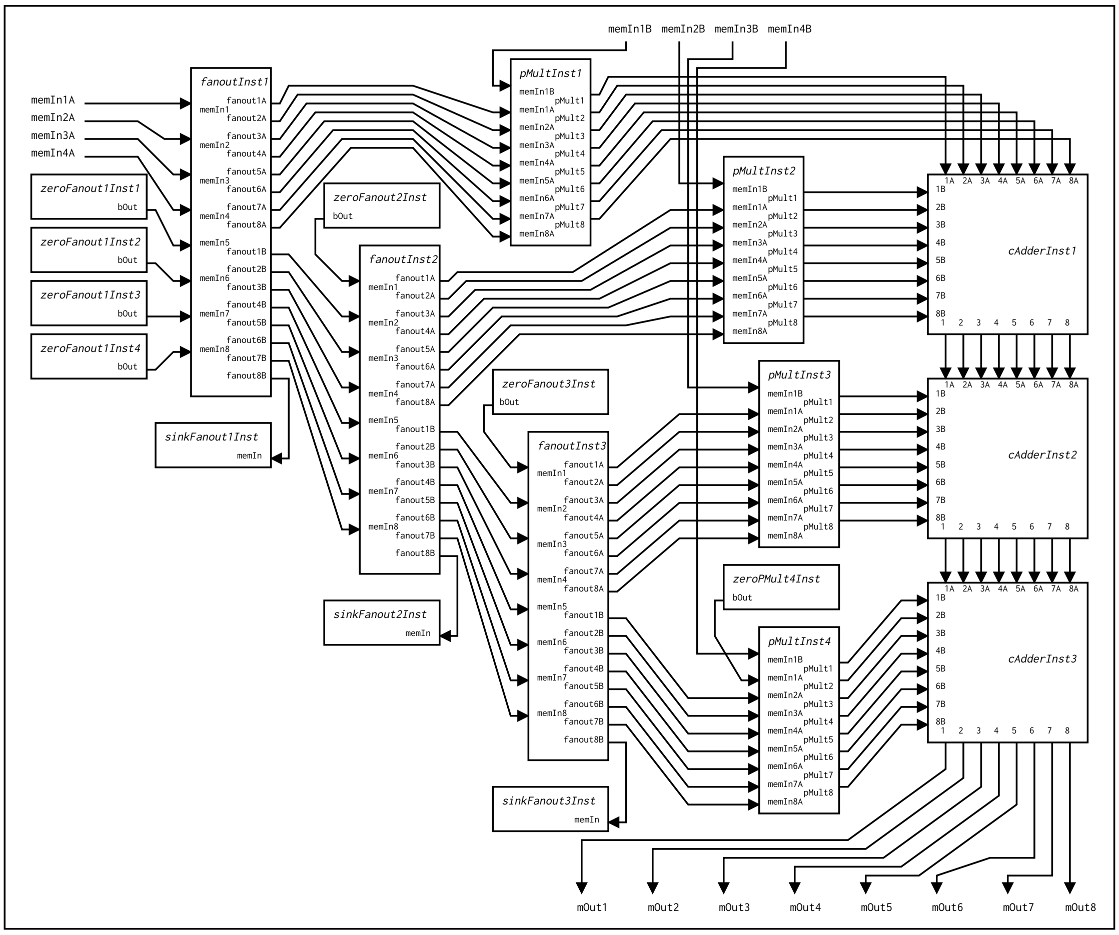


Figure 6.5: Structure of multiplier

and partial multiplier instances, which we do not intend to re-use anywhere else. Had we declared these libraries directly within the `multLib` library rather than inside the `multInst` instance, then these inner libraries would have been accessible from outside the instance definition, and outside the `multLib` library by qualifying the library name as `multLib.fanout8Lib.fanout8Inst` or (had we dispensed with the intervening library container) `multLib.fanout8Inst`.

Listing 6.11: The four-bit multiplier: inline fan-out library

```
llib multLib {

  linst multInst {

    llib fanout8Lib {

      linst fanout8Inst {

```

```
    linst bFanout2Inst1 : boolBoxLib.fanout2Inst;
    ...
    linst bFanout2Inst8 : boolBoxLib.fanout2Inst;
  }

  observe {

    bFanout2Inst1.memIn as memIn1;
    ...
    bFanout2Inst8.memIn as memIn8;
  }

  manifest {

    bFanout2Inst1.fanoutA as fanout1A;
    ...
    bFanout2Inst8.fanoutA as fanout8A;

    bFanout2Inst1.fanoutB as fanout1B;
    ...
    bFanout2Inst8.fanoutB as fanout8B;
  }
}
```

Listing 6.11 gives the definition of the 8-bit two-fold fan-out as the parallel composition of eight 1-bit two-fold fan-outs.

Listing 6.12: The four-bit multiplier: inline partial multiplier library

```
llib pMultLib {

  linst pMultInst {

    linst fanout2Inst1 : boolBoxLib.fanout2Inst;
    linst fanout2Inst2 : boolBoxLib.fanout2Inst(1);
    linst fanout2Inst3 : boolBoxLib.fanout2Inst(1);
    linst fanout2Inst4 : boolBoxLib.fanout2Inst(2);
    ...
    linst fanout2Inst7 : boolBoxLib.fanout2Inst(2);

    linst andInst1 : boolBoxLib.andGateInst(3);
    ...
    linst andInst8 : boolBoxLib.andGateInst(3);

    observe {

      andInst1.memInA as memIn1A;
      ...
      andInst8.memInA as memIn8A;

      fanout1Inst1.memIn as memIn1B;
    }

    manifest {

          andInst1.andOut as pMult1;
          ...
          andInst8.andOut as pMult8;
        }

        fifo fanout2Inst1.fanoutA to fanout2Inst2.memIn;
        ...
        fifo fanout2Inst3.fanoutB to fanout2Inst7.memIn;
        fifo fanout2Inst4.fanoutA to andInst1.memInB;
        ...
        fifo fanout2Inst7.fanoutB to andInst8.memInB;
      }
    }
```

The multiplier's inline partial multiplier library is shown in Listing 6.12. It carries out a 1 bit eight-fold fan-out using a pipeline of two-fold fan-outs. For each output from the fan-out assembly, the pipeline is three fan-outs long. The whole fan-out system uses seven two-fold fan-outs in total.

The resulting eight copies of the single input bit are then sent to eight AND gates, which multiply this number by each of the eight bits of the second argument to the partial multiplier. The partial multiplier is instantiated four times, and a set of source and sink boxes together with 8-bit parallel fan-outs then arrange the necessary bit-shifting of the first multiplier arguments, so that it can be partially multiplied with each of the four bits of the other multiplier argument in turn. The results of these partial multipliers are then added together in a cascaded pipeline of three cascade adders.

Listing 6.13: The four-bit multiplier: concrete part of definition

```
    linst fanoutInst1 : fanout8Lib.fanout8Inst(1);
    linst fanoutInst2 : fanout8Lib.fanout8Inst(2);
    linst fanoutInst3 : fanout8Lib.fanout8Inst(3);

    linst zeroFanout1Inst1 : sourceSinkLib.source0Inst;
    ...
    linst zeroFanout1Inst4 : sourceSinkLib.source0Inst;
    linst sinkFanout1Inst : sourceSinkLib.sinkInst(2);

    linst zeroFanout2Inst : sourceSinkLib.source0Inst(1);
    linst sinkFanout2Inst : sourceSinkLib.sinkInst(3);

    linst zeroFanout3Inst : sourceSinkLib.source0Inst(2);
    linst sinkFanout3Inst : sourceSinkLib.sinkInst(4);

    linst pMultInst1 : pMultLib.pMultInst(2);
    linst pMultInst2 : pMultLib.pMultInst(3);
    linst pMultInst3 : pMultLib.pMultInst(4);
    linst pMultInst4 : pMultLib.pMultInst(4);
    linst zeroPMult4Inst : sourceSinkLib.source0Inst(3);

    linst cAdderInst1 : cAdderLib.cAdderInst(7);
    linst cAdderInst2 : cAdderLib.cAdderInst(12);
    linst cAdderInst3 : cAdderLib.cAdderInst(17);

    observe {

      fanout1Inst.memIn1 as memIn1A;
      ...
      fanout1Inst.memIn4 as memIn4A;
      pMultInst1.memIn1B as memIn1B;
      ...
      pMultInst4.memIn1B as memIn4B;
    }

    manifest {

      cAdderInst3.sumOut1 as mOut1;
      ...
      cAdderInst3.sumOut8 as mOut8;
    }

    fifo zeroFanout1Inst1.bOut to fanoutInst1.memIn5;
    ...
    fifo zeroFanout1Inst4.bOut to fanoutInst1.memIn8;

    fifo zeroFanout2Inst.bOut to fanoutInst2.memIn1;
    ...
    fifo fanoutInst1.fanout7B to fanoutInst2.memIn8;
    fifo fanoutInst1.fanout8B to sinkFanout1Inst.memIn;

    fifo zeroFanout3Inst.bOut to fanoutInst3.memIn1;
    ...
    fifo fanoutInst2.fanout7B to fanoutInst3.memIn8;
    fifo fanoutInst2.fanout8B to sinkFanout2Inst.memIn;

    fifo fanoutInst1.fanout1A to pMultInst1.memIn1A;
    ...
    fifo fanoutInst1.fanout8A to pMultInst1.memIn8A;

    fifo fanoutInst2.fanout1A to pMultInst2.memIn1A;
    ...
    fifo fanoutInst2.fanout8A to pMultInst2.memIn8A;

    fifo fanoutInst3.fanout1A to pMultInst3.memIn1A;
    ...
    fifo fanoutInst3.fanout8A to pMultInst3.memIn8A;

    fifo zeroPMult4Inst.bOut to pMultInst4.memIn1A;
    ...
    fifo fanoutInst3.fanout7A to pMultInst4.memIn8A;
    fifo fanoutInst3.fanout8A to sinkFanout3Inst.memIn;

    fifo pMultInst1.pMult1 to cAdderInst1.memIn1A;
    ...
    fifo pMultInst1.pMult8 to cAdderInst1.memIn8A;

    fifo pMultInst2.pMult1 to cAdderInst1.memIn1B;
    ...
    fifo pMultInst2.pMult8 to cAdderInst1.memIn8B;

    fifo cAdderInst1.sumOut1 to cAdderInst2.memIn1A;
    ...
    fifo cAdderInst1.sumOut8 to cAdderInst2.memIn8A;

    fifo pMultInst3.pMult1 to cAdderInst2.memIn1B;
    ...
    fifo pMultInst3.pMult8 to cAdderInst2.memIn8B;

    fifo cAdderInst2.sumOut1 to cAdderInst3.memIn1A;
    ...
    fifo cAdderInst2.sumOut8 to cAdderInst3.memIn8A;

    fifo pMultInst4.pMult1 to cAdderInst3.memIn1B;
    ...
    fifo pMultInst4.pMult8 to cAdderInst3.memIn8B;
  }
}
```

Listing 6.13 gives the rest of the definition of the multiplier. The labelled memories and instances correspond to the labels in Figure 6.5. The instance computes the multiplication by summing, using three cascade adders, four partial multiples of the 'A' input bits by each of the 'B' input bits in turn. The output of each fan-out is progressively bit-shifted so that each partial multiple has the correct significance in the sum. The source and sink boxes provide that no memories are left dangling as a result of the skew between inputs and outputs introduced by the bit-shifting.

The inputs and outputs of the multiplier trace have been filtered into appendix F.8. This example is very substantially more complicated than any of the preceding ones, instantiating several previous instance definitions as well as defining new ones. The resulting state space gives rise to very large (hundreds of megabytes) raw XML traces. This is too large to be processed easily by a document object model (DOM) driven stylesheet.[7] We therefore extended the filtration system using a small Java Class. The Java Class broke up each trace serially into top-level coordination state objects using the StAX interface. It then converted each coordination state object into a DOM object, before finally applying an XSLT stylesheet to each such object and reassembling the results of each transformation into a filtered trace. This filtered trace was suitable to be converted back into LaTeX through TeXML . This process is somewhat *ad hoc*, but in performing these tasks programmatically, we can offer a high level of assurance that the graphical representation properly shows what is in the original trace without fear of the errors that would invevitably occur in performing this process manually. To do any better than this would involve verifying a type-setting program.

We see from Listing F.8 that the result of the input presented with one cycle on the

[7] XSLT 3.0 contains features for dealing with streaming XML data, but there are not yet any open source XSLT 3.0 processors available.

top-level wall-clock is 1011 × 0111 = 01001101, or 11 × 7 = 77 in base 10.

6.8.5 Triple modular redundancy (TMR) replicated multiplier

The replicated multiplier is illustrated in Figure 6.6.

The first part of the instance definition gives another inline library. This is a four-bit three-fold fan-out library, instantiating four of the single bit three-fold fan-outs in a parallel composition. It is shown in Listing 6.14.

Listing 6.14: The replicated four-bit multiplier: inline fan-out library

```
llib replMultLib {

  linst replMultInst {

    llib fanout4x3Lib {

      linst fanout4x3Inst {

        linst bFanout3Inst1 : boolBoxLib.fanout3Inst;
        ...
        linst bFanout3Inst4 : boolBoxLib.fanout3Inst;

        observe {

          bFanout3Inst1.memIn as memIn1;
          ...
          bFanout3Inst4.memIn as memIn4;
        }

        manifest {

          bFanout3Inst1.fanoutA as fanout1A;
          ...
          bFanout3Inst4.fanoutA as fanout4A;

          bFanout3Inst1.fanoutB as fanout1B;
          ...
          bFanout3Inst4.fanoutB as fanout4B;

          bFanout3Inst1.fanoutC as fanout1C;
          ...
          bFanout3Inst4.fanoutC as fanout4C;
        }
      }
    }
```

The replicated multiplier makes use of the single-bit three-input Boolean voter library, listed in appendix E.3, whose truth table is as follows:

inMemA	inMemB	inMemC	voter3Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 6.5: Voter truth table

We do not have a means to simulate faults to put this voter to the test: we leave that to further work.

This voter is instantiated eight times in the three-input voter library given in Listing 6.15.

Listing 6.15: The replicated four-bit multiplier: inline voter library

```
llib voter8x3Lib {

  linst voter8x3Inst {

    linst bVoter3Inst1 : votersLib.voter3Inst;
    ...
    linst bVoter3Inst8 : votersLib.voter3Inst;

    observe {

      bVoter3Inst1.memInA as memIn1A;
      ...
      bVoter3Inst8.memInA as memIn8A;

      bVoter3Inst1.memInB as memIn1B;
      ...
      bVoter3Inst8.memInB as memIn8B;

      bVoter3Inst1.memInC as memIn1C;
      ...
      bVoter3Inst8.memInC as memIn8C;
    }

    manifest {

      bVoter3Inst1.voter3Out as voter1;
```

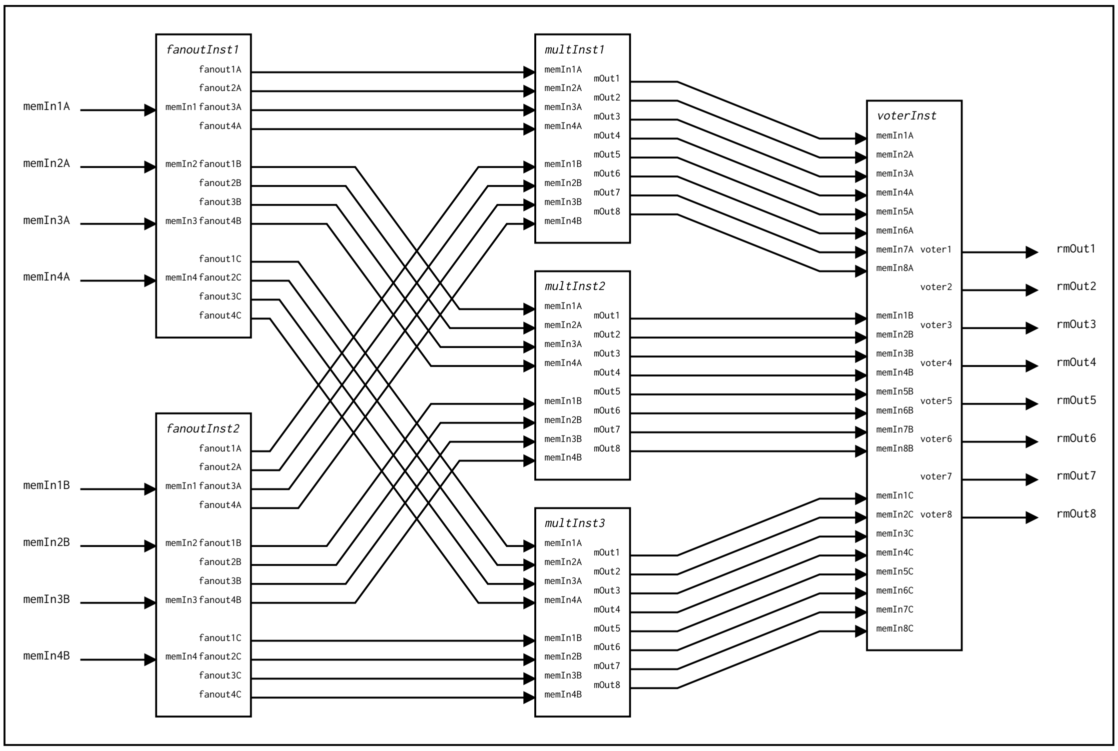


Figure 6.6: Structure of replicated multiplier

```
          ...
          bVoter3Inst8.voter3Out as voter8;
        }
      }
    }
```

The replicated multiplier instantiates the multiplier of section 6.8.4 three times, and declares two new local instance types to construct an 8-bit voter and two 4-bit three fold fan-outs for each of the three multipliers. This can be seen in Listing 6.16. It is all linked together with yet more FIFOs. We remark that, while our voter is a single point of failure, all single points of failure inside an HBCL design may be eliminated if we apply overlapping replication transformation to a program section so that voters are replicated and attached to the next stage of replicating logic. This is not a new observation: it goes back to von Neumann's idea of 'restoring organs' [189].

Listing 6.16: The replicated four-bit multiplier: concrete part of definition

```
    linst fanoutInst1 : fanout4x3Lib.fanout4x3Inst;
    linst fanoutInst2 : fanout4x3Lib.fanout4x3Inst;

    linst multInst1 : multLib.multInst(1);
```

```
  linst multInst2 : multLib.multInst(1);
  linst multInst3 : multLib.multInst(1);

  linst voterInst : voter8x3Lib.voter8x3Inst(57);

  observe {

    fanoutInst1.memIn1 as memIn1A;
    ...
    fanoutInst1.memIn4 as memIn4A;

    fanoutInst2.memIn1 as memIn1B;
    ...
    fanoutInst2.memIn4 as memIn4B;
  }

  manifest {

    voterInst.voter1 as rmOut1;
    ...
    voterInst.voter8 as rmOut8;
  }

  fifo fanoutInst1.fanout1A to multInst1.memIn1A;
  ...
  fifo fanoutInst1.fanout4A to multInst1.memIn4A;

  fifo fanoutInst2.fanout1A to multInst1.memIn1B;
  ...
  fifo fanoutInst2.fanout4A to multInst1.memIn4B;

  fifo fanoutInst1.fanout1B to multInst2.memIn1A;
  ...
  fifo fanoutInst1.fanout4B to multInst2.memIn4A;

  fifo fanoutInst2.fanout1B to multInst2.memIn1B;
  ...
  fifo fanoutInst2.fanout4B to multInst2.memIn4B;

  fifo fanoutInst1.fanout1C to multInst3.memIn1A;
  ...
  fifo fanoutInst1.fanout4C to multInst3.memIn4A;

  fifo fanoutInst2.fanout1C to multInst3.memIn1B;
  ...
  fifo fanoutInst2.fanout4C to multInst3.memIn4B;

  fifo multInst1.mOut1 to voterInst.memIn1A;
  ...
  fifo multInst1.mOut8 to voterInst.memIn8A;

  fifo multInst2.mOut1 to voterInst.memIn1B;
  ...
  fifo multInst2.mOut8 to voterInst.memIn8B;

  fifo multInst3.mOut1 to voterInst.memIn1C;
  ...
  fifo multInst3.mOut8 to voterInst.memIn8C;
  }
}
```

The inputs and outputs of the replicated multiplier trace have been filtered into appendix F.9. The same results can be seen as in section 6.8.4: we read off a value of 77. This time, the inputs and outputs can be seen in the fan-out and voter instances. For the sake of simplicity, we have realigned the cascade of information coming from the multiplier, so that all input and output is read off with 58 cycles showing on the top-level wall-clock. Long FIFOs exist between the least significant bits coming from the cascade adders at the output of the multipliers and inputs of the voter boxes. It would be possible to optimize this away if the outputs of the replicated multiplier were to be fed to similarly cascaded inputs.

It can be seen that the signature of the replicated multiplier is almost exactly the same as that of the single multiplier: the only difference is that the latency is greater to accommodate the fan-out and voting stages.

To show how the bisimilarity of this trace to the non-replicated version can be established, we can construct a similar argument to that which we gave in the context of the parallel composition example. This time, we have a more interesting interpretation function. As we consider a coinductive predicate over both traces, the propositional term that we need to add in the constructor of this predicate (corresponding to the constructors of each step of the respective traces) is one that axiomatizes the interpretation function. In this simple TMR case, the interpretation function is a two-out-of-three voter, so we can construct the bisimulation predicate if any two of the nested replicated multipliers are equal to the state of the non-replicated multiplier for the coordination states corresponding to the relevant time slice. This is precisely the same logic as implemented in the computable voter function that processes the outputs of the replicated multipliers. We can use this bisimulation predicate to define a set of acceptable witnesses to the unreplicated multiplier that is satisfied even if we inject faults into the underlying implementation, as long as no more than one fault occurs at a time. The memory holding the voter output in the replicated example is the same as the interpretation of the final set of multiplier outputs, because the executable voter logic is identical to a two-out-of-three *predicate*. We observe by inspection of the traces that these properties hold.

We note that nothing in the interpretation function requires that each replicated multiplier be implemented in the same way, or on the same hardware. As long as the FIFOs from the individual multipliers to the fan-outs and voter boxes can be constructed without violating the boundary conditions on the 'rubber sheet' model, a bisimilarity predicate can be constructed in the way outlined, with temporal displacements or an entirely different multiplier design. In terms of Figure 6.6, we can partition the replicated multiplier specification and apply different morphisms to different parts.

It would be desirable in further work to find ways of sharing signatures in such cases that were *exactly* the same, without adding explicit boxes for temporal padding. This might be done by constructing more subtle time-shifting semantics on memories, where a kind of temporal memory polymorphism could be enabled by the use of varying buffer lengths or parametrizable outward-facing buffer capacities. This complexity of passive observation/manifestation semantics is the price we pay for avoiding the transactional drawbacks of message-passing specification approaches.

6.8.6 Scale metrics

Finally, we present some measures of the relative scale of our HBCL programs.

Program scenario	Number of boxes	Number of FIFOS	HBCL program LOC	Coq AST LOC	Execution time (user-mode seconds)	Feedbacks	Deepest nesting level
Negator	1	0	23	1,090	0.04	0	0
Parity	1	0	32	1,184	0.05	0	0
Parallel composition	2	0	20	405	0.10	0	1
Pipeline	2	1	16	375	0.11	0	1
Checksum	1	1	36	1,773	0.11	1 (direct)	1
Checksum pipeline	2	2	17	430	0.18	1 (indirect)	1
Half adder	4	4	26	786	0.31	0	2
Full adder	9	11	25	805	0.71	0	3
Cascade adder	74	97	57	1,882	89.77	0	4
Multiplier	316	381	283	9,641	805.17	0	5
Replicated multiplier	964	1191	190	6,174	5,388.85	0	6

Table 6.6: Program statistics

The timings are user-mode times obtained from running programs under the `time` program on Linux at the `bash` shell. The comparatively low line counts for the examples with very large numbers of boxes are a result of the instantiation by reference of previously defined components.

The most important observation about the program statistics is that the complexity (in terms of total number of boxes and FIFOs in an example) increases rapidly for only modest increases in the amount of code. The ability to do this comes from the nesting semantics. We can see that, if we chose to prove something about a particular HBCL program (embedded in Coq), then nesting would enable us to gain similar productivity in the proof domain. The other interesting observation is that the execution time is not linearly related to the number of boxes and FIFOs, but increases more quickly than that. This is likely to be due in large part to the way the implementation is structured, where the static instance definition object is consulted in its entirity for each time step; as the instance object becomes larger, it takes longer to retrieve the relevant box or FIFO specification for each part of a coordination step increment.

6.9 Possible technical improvements

In writing and using the HBCL implementation, a number of technical matters have been identified that it would be worthwhile to address in further work.

6.9.1 Use of tactic language

Our proofs have in general used scripts of simple tactics and 'tacticals'.[8] Coq has a tactic language called 'Ltac', and in further work we would use this much more, as our proofs using simple tactics are hard to follow and often repetitious. Chlipala [48] has advocated an approach to using tactics in Coq where every proof is generated by a single invocation of one customized higher-order tactic. This is an elegant paradigm, and compensates for one of the disadvantages of Coq compared to Isabelle-like proof assistants: Isabelle encourages proofs to be structured in the way that one would expect to read them in a non-automated setting, while Coq's functions-as-proofs paradigm means that proofs can be almost impossible to understand by inspection. In future work, we would adopt the pattern advocated by Chlipala. Although this method would at first be even more time-consuming than the approach we have employed, we expect that in the end it would make the task quicker and easier. It would also help us to close the remaining admitted lemmas in the expression language, and, after adding more struture to predicate terms in the coordination language, to discharge the resultant proof obligations and purge its admitted lemmas. If we were to attempt to do this without us-

[8]'tacticals' in Coq are simple compositions of tactics.

ing more sophisticated proof tactics, the scripts would become intractably long-winded and difficult to follow.

6.9.2 Compiler

We have compiled HBCL programs by hand in the present work, to produce inhabitants of static semantic objects for each HBCL program in Coq code. These were subsequently extracted to OCaml by Coq and compiled using the OCaml compiler. Hand compilation has been used because the interesting features of the language are contained in the idea of a static semantic object Curried to an interpreter function to create an executable HBCL function deeply embedded in the host logic. It has therefore not been a priority to develop a verified compiler. However, with the most advanced examples we have demonstrated, hand compilation became an extremely repetitious and error-prone process, and any further development will require a compiler. We have envisaged from the beginning that the compiler would be a direct implementation of the static semantics. The static semantic object is correct by construction, so the presence of a 'bottom' static semantic object indicates a failed compilation where the static semantics have been violated and it is thus impossible to produce a correct static semantic object.

A further desirable step after generating a compiler would be the production of a debugging tool allowing real-time tracing and stepping. It is not reasonable to ask a user to identify bugs solely from large trace files.

6.9.3 Refinement of semantics

Separate state variables for FIFOs are a priority, in order to decouple fully the space requirements of memories from the latency of the FIFOs to which they are connected, and in order to remove the need for explicit minimum and maximum buffer length parameters in the specification of memories.

It would also be useful to explore a dual version of the semantics in which FIFOs notionally executed *before* boxes. This would save us from having to increment the clock in the middle of a nested instance invocation, and would therefore be tidier. There would be a bijective mapping between the traces generated by such semantics and those produced by the current version.

Our 'time-shifting' semantics have proved powerful. It would be productive to extend them to perform more general temporal transformations on logical instances, multiplying their clock frequencies by constants, as well as translating them as we do at present.

Finally, we might wish to change how we express the dynamic semantics governing how memories from nested HBCL instances are accessed. The reference semantics allow outer boxes to see inside and perform I/O directly on nested boxes, while the interpreter written in Coq avoids this. The matter is one of trading concision of dynamic semantics for data opacity in nested instances. If we introduce mutually nested coordination languages, the latter consideration would probably win out, resulting in a larger coordination state object in the reference semantics. This would not affect the efficiency of implementations, since they would be engineered to optimize out these referential indirections (bisimilarly), resulting in an interpreter more like our current reference design in Coq in this respect.

6.9.4 Recasting modules into records

We have already observed several drawbacks involved with using Coq's module system. The ugliest problem we have found is that the Standard Library map axiomatization and implementations are *statically* parametric in the type of their keys. In our development, a large number of our data maps have multiple but similar instantiations with different types of keys — in particular, the maps of input and output memories. The Standard Library requires that a Standard Library module be instantiated statically and separately for each type, which is extremely cumbersome. These difficulties could be addressed by recasting our static modules in the paradigm of the Ext-lib [55] library, using dependently typed records and type-classes. A version of Coq that featured universe polymorphism according to the proposals of Sozeau *et al.* [174] would also assist with some of these problems. We would also welcome a version of Coq in which it was syntactically and semantically possible to differentiate between function and dependent inductive types.

6.9.5 Higher-order functions with proofs

We have repeatedly used a paradigm where a property of an associative array implies properties of each member of that array. This has usually been implemented by recursing over a list of `(key, element)` pairs, which is ugly, and we would like to factor this out into a higher-order functional construction. Where we have been able to use map functions from the Standard Library, we have had to write tortuous predicates to convince Coq that the resulting functions terminate. It would be advantageous to develop a new map function that takes care of this design pattern in a way that is hidden from the calling code, being parametrized in the relationship between a predicate over the map and a predicate over its members. This would involve extending the Standard Library associative arrays (maps) or the Ext-lib equivalents.

6.9.6 Removal of convenience arguments

It would be desirable to neaten our reference interpreter by removing those extra arguments to functions that made its creation easier. This process is already evident in the inference of nested clocks from static information. More work in this vein would also make the syntax of the Coq functions used more closely resemble the more informal semantic rules, addressing the issues we identified in section 5.3. Although this would make the reference interpreter more inefficient at run-time, this is a price worth paying, given that the reference interpreter is intended to give conviction to the operational semantic rules by mirroring their structure as closely as possible, rather than to be deployed to real implementations of HBCL instances. The boxes of HBCL are designed to specify parallel computation, and any serialized simulation in a single thread of execution will always be disproportionately slow.

6.9.7 Unification of fixpoint and cofixpoint versions of semantics

The desirable goal of unifying fixpoint and cofixpoint versions of the semantics would only be possible if Coq provided more flexible syntax for the cofixpoint guard condition in terminating cases. However, there is no immediate prospect of any changes to the way Coq formalizes recursion over coinductive structures.

6.10 Summary of points demonstrated

In total, the entire development, including all runnable code, proof and examples amounts to 81,654 lines of Coq code. The hand-compilation of HBCL programs is evident in the Coq files bearing the names of their respective examples. There are several files for each example, with instance signatures and harmonic boxes separated from the main instance declarations. In addition, test rigs are written as functors using Coq's `Section`ing mechanism, permitting instance closures to be expressed conveniently in a shallow embedding style.

This chapter has demonstrated the detailed behaviour of the set of examples that we have been discussing throughout the thesis, as well as explaining how a much more complex example works. It has shown how the time-stamped values being observed and manifested work in practice.

We have shown how bisimilarity predicates can be constructed directly over equivalent traces, advancing one of our main hypotheses. By showing how bisimulation predicates that we discussed in chapter 5 can be constructed by inspection of traces, we have advanced our claim that the use of a single coordinate system for anything written in the language makes the description of these objects much easier than forming similar predicates over more directly physical implementations. We can make transformations such as the replication transformation of the multiplier *reflexively* in HBCL, which leaves the 'rubber sheet' implementation as a final step that can be performed when reflexive transformations have been completed. This transitive argument allows us to defer difficult but uninteresting considerations of partial orders.

We learnt in producing these traces that the export workflow from Coq in the presence of admitted lemmas can be extremely difficult to debug, and that they can be difficult to read. The general limitation of model-checking also applies, in that coverage of the design and state space is limited. However, the traces give a far better intuitive grasp of HBCL and how arguments about the language are constructed than is possible with a a set of theorems over the the predicate characterization of the language.

Chapter 7

Conclusions and further work

7.1 Principal contributions

In realizing a language that meets the aspirations we have set, a number of existing ideas have been extended. It has also been necessary to introduce some novelties and innovations. This section summarizes them against the hypotheses stated in section 1.2.

7.1.1 Absolute time

7.1.1.1 Ontology of global clocking and coordinate system

Hbcl defines a computation as a function of timed observations in a coordinate system in spacetime. Observations are axiomatized over an identifier space, such as iso/itu object identifiers, and a type system. Our conception of 'oid semantics', which we derive from this, is a novelty. Hbcl is a 'harmonic' language, with every component operating at some rational-numbered frequency. The periodic structure provides temporal firewalls between each 'box' containing a computation. This means, that in principle, hbcl lends itself to resource use analyses that are orthogonal between instances.

7.1.1.2 Ontology observation semantics

Ontology observation semantics have a particular importance in tolerating faulty implementations. Pre-hbcl is an axiomatization of observation semantics. It comprises that part of hbcl that defines the names, types and frequencies of periodic observations and manifestations of values: it defines the ontology of the subject matter of the specification.

7.1.2 Deep embedding

7.1.2.1 Dependent type-theoretical semantics

We seek an equivalence class of executable specifications over functions with different structures but which compute the same output for each input: in other words, we want the predicate defining the relation over domain and co-domain to be provably sound and compatible with an appropriate axiomatization of completeness. We also want an inhabitant of the function type derived from this predicate as a witness proving that it is computable, and we use this to discharge the soundness proof. This type-theoretic approach has further advantages in its potential to show interesting morphisms between reflexive transformations in HBCL, the correctness of implementations as witnesses to HBCL specifications, and even, *in extremis*, the equivalence of different axiomatizations, potentially as embedded in different logics.[1] It also becomes trivially possible to give a type for a compiler or interpreter: we would like any inhabitant to be a valid tool of the relevant type.

The use of dependent type-theoretical semantics has enabled the treatment of embedded box languages as first class quantities. The paradigm of forming these types using parametrized σ-types to create semantic objects is extremely powerful. The dynamic semantics are formed from a predicate characterization of legitimate evolutions over a domain, where the semantics of a particular instance are fixed by a dependent type in the static semantic object of a program, which is itself dependent in its input/output signature. A type of implementations is given, which is the type of function that produces evolutions that comply with the predicates, as incorporated into appropriate σ-types. The result of all this is a specification of a coinductive 'type' for part of physical reality, whose single inhabitant, uniquely determined by an observed coinductive input stream, corresponds to the history of real-world events as they unfold.

7.1.2.2 Polarization of specification languages into formal logics and ontology

Through the comparison of Table 2.1 and our arguments regarding predicates and morphisms in chapter 5, we have established that the decoupling of predicate and deductive machinery found in specification languages from a sparse ontology produces an economy of formalisms and a clarity of expression for predicate properties.

[1]Carrying forward section 2.9, we amplify this point in section 7.4.5, asking how the limits of confidence in tools and formalisms developed in this way have been and might be approached in a limit, as constrained by Gödel's incompleteness theorem and limitations of logic. To this end, we propose further research.

7.1.3 Ease of proof

7.1.3.1 Formalization in a proof assistant

To increase our conviction in the effectiveness of the semantics, we have deeply embedded the language in Coq.

7.1.3.2 Working simulation and proof

A partially verified interpreter is provided that takes as arguments the static type of a program and a set of input streams, and which produces execution traces. Two sets of progressive examples are provided and hand-compiled into the necessary static semantic objects. The first set of five examples shows that the basic operations of parallel composition, pipelining and feedback are easy to do with HBCL. Such structures exhibit (deterministic) concurrency, signal processing, memory, and a coordination analogue of recursion. The second set of examples uses even simpler expressions inside execution boxes, but dramatically increases the number and complexity of FIFOs to demonstrate the scalability of the language. The simple expression language provided with the coordination language amounts to a kind of hardware description language that can be viewed as a particularly degenerate subset of HBCL. This set of examples culminates in a replicated pipelined multiplier that is designed to give a taste of how the language is adapted to specifying for fault-tolerant implementations. If all lemmas are discharged, the existence of the interpreter immediately gives the soundness of the language, *up to* the consistency of the formalizing logic. Without discharging all the lemmas, the working interpreter provides a convincing, if not conclusive, means to show consistency properties by an empirical analogue of proof by reflection.

We have produced demonstrations of working HBCL simulations with good coverage of scenarios, both in terms of conformation and complexity. This constitutes good evidence that the language concept that we have presented works in practice.

7.2 Ancillary contributions

In addition to the main contributions, the work has produced the following unusual features and insights:

7.2.1 Separation of coordination and expression constructions: box languages as parameters

The separation of coordination languages and expression languages is not new, but our parametrization of a timed coordination language on another arbitrary language is novel. Our type-theoretical semantics gives a higher-order type of *harmonic box languages* that fit into a harmonic box. Bindings of *untimed box languages* to harmonic box languages can be given, and expression languages can be embedded in untimed box languages. These two abstraction layers between coordination and expression languages are novel. So, too, is the treatment of a self-nested coordination language, obtained when a binding of itself is provided as a harmonic box language. While nested coordination languages are not new, potentially *heterogeneous* nested coordination languages are a new development.

Full HBCL introduces a means to specify the computation of an instance, elaborating on its Pre-HBCL signature. It also provides the full semantics of nesting, whereby internal instances run for short bursts while they will not block on lack of input. Issues of binding of untimed and harmonic box languages are addressed in full HBCL. The specification of these box languages and their bindings are dynamic parameters of HBCL which, when provided, turn full HBCL into an executable language.

A simple box language is provided. Box languages may, in a fully general case, be arbitrary languages[2], which may be coordination languages in their own right. Box languages are first order objects in the formalization of the coordination language; we claim nothing special about the expression language presented, save that it satisfies this novel paradigm. Nested instances can be regarded as a special recursive case of a box language.[3] In this thesis, we provide and bind an expression language as a degenerate case. This language is a simple combinatorial logic language over tuples and records of bits, but is expressive enough to show all the key features of the full HBCL coordination language.

7.2.2 Formalization of FIFOs

The way in which we formalize FIFOs is unusual. They are defined by reference to observations driven by a clock *only*. In accordance with the philosophy of observation semantics, there is no hand-shaking at either end of the FIFO. The length of each FIFO is defined implicitly by the temporal offset of memories at either end of the pipe.

Our formalization of FIFOs is key to our ability to dispense with message passing as a means of inter-box communication. They are clocked independently of the memories they are connected to, and, as long as input data is supplied, offer values for observation at their output ends which have a known 'time of validity' and a known offset compared with the time of validity expected at their inputs. It is the difference in the time-to-live or time-from-live parameters of the HBCL memories at each end of a FIFO that determines its length. FIFOs are able to start without initialization, and recover from transient missing data, because any absent data value on input is converted to a null value on output.

[2]Provided, if expressive enough to admit non-termination, they come with program-by-program proofs of termination.

[3]Indeed, it would be possible to instantiate Pre-HBCL in a version of full HBCL which lacked this nesting, or conversely, involved mutual recursion with some other coordination language. This non-canonicity of full HBCL is one of the reasons it is separated from Pre-HBCL.

This kind of timed FIFO based on *absolute* timings and axiomatically constant latency offers a data transmission mechanism that has a higher level of abstraction and greater temporal coherence than the packetized abstractions used in, say, synchronous optical networking protocols.

7.2.3 Agents and entropy

HBCL is a concurrent language with no 'agents'. The coordination language presented in this thesis advances the idea that 'agency' has no place in a coordination language that is meant to *specify*, as opposed to *model* a concurrent system. This does not mean that it is not possible to program applications in HBCL that behave quasi-autonomously, but that apparent autonomy must always be a consequence of hidden entropy that is a *parameter* to the HBCL program: either explicitly as a stream of binary (Shannon) entropy, or implicitly in the ordering of data in a program's input streams. HBCL programs are deterministic functions from how the world can be observed to what must become observable if the causative relationship specified by the program is to be realized. Entropy is observed like any other input. This approach has some interesting and even alarming corollaries that we touch on later.

Familiar models of concurrency, such as as process calculi [108], the Actor model [106] and — more recently — bigraphs [141] model autonomous agents. In a practical HBCL development, such models would be useful in producing implementations that are witnesses to an HBCL trace, or as a high level modelling tool for designing a parallel application that is implemented on an HBCL substrate. This forms an 'entropy sandwich' that is explored further in appendix A.4.1.

7.2.4 Instance and library closure semantics

HBCL makes extensive use of re-usable components, and makes use of a 'library' abstraction in order to do this. This much is unremarkable. However, an unusual feature of HBCL is that it can definitively express instance functors, nuanced instance visibility, and library resolution without dependence on a root namespace for library invocations. It does all this with only two syntactic constructs for each type of instance (logical, hardware and configuration).

7.3 Critical evaluation

7.3.1 Evaluation of hypotheses

7.3.1.1 Ontology of global clocking and coordinate system

In section 3.1 we introduced the idea of a synchronous language that is not, in general, capable of defining its own local clock along with its programs or hardware descriptions, but instead depends on some pre-existing clock in the real world. The language uses the clock and observable quantities to prescribe the evolution of events as parametrized by what is observed. The resulting specifications are not executable programs in the normal sense, but rather witnesses to 'executions' produced by physics if the specification is accurately implemented. This means that when we 'execute' a program on an 'interpreter' we are doing so in a *simulated* reality — an ontological sandbox. Our claim was that this approach offered advantages in terms of clarity, composability, and that it was especially amenable to replication transformations. We now evaluate these claims.

The point of adding object identifiers to data types and observable quantities that are unique from a practical perspective by grafting them into a generally recognized designation schema is unabiguous: that much is obvious. The question is, is this extreme clarity useful? We have demonstrated this to a limited extent, but when the examples are of components such as arithmetic units, the advantages are largely untapped, because the various components of an arithmetic unit are usually in practical terms under the control of a single clock in any event. It is difficult to construct an implementation of the kind of safety-critical system to which we would like to apply this approach, since such systems are inherently extremely costly, and the time taken to design a realistic application is out of proportion to what one person working alone can achieve in a reasonable time. However, we do note that the importance of knowing exactly which aircraft has which transponder fitted to it, or which nuclear fuel rod a particular calculation concerns, are uncontroversially important: the benefit of an approach like that of HBCL is that by placing all of the logical properties directly in the hands of a proof assistant, that proof assitant can provide a unified semantics of all elements of an application domain.

In spite of these problems, the clarity of HBCL has been shown in the replicated transformation example, in which a reflexive transformation within the deterministic rational-numbered time model that HBCL uses allowed us to produce a replicated transformation very quickly and easily, abstracting over the implementation details. In explaining how predicates can be constructed over output traces, we illustrated our point about witnesses of executions, which can tolerate temporal and spatial skews as long as causation boundary conditions are respected and morphisms can be shown.

Composability is rather easier to demonstrate, as the principles can be shown with simple examples. In the case of the parallel and pipelined composition examples, and in that of the replication example, we showed how predicates can be constructed over traces to demonstrate a one-to-one correspondence between two separate traces of two separate examples and one trace of those instances combined.

7.3.1.2 Use of deep embedding

The use of a deep embedding enabled us to reason in chapter 3 about how our system might be implemented beyond the reference interpreter. In chapter 5, we developed the idea of how morphisms can be used to reason across multiple domains, from natural numbers to physical reality. While to reason about concrete physical implementations and continuous time formalisms went beyond the scope of this project, the fact that a deep embedding enables one to frame these questions in a precise formalism is in itself valuable. This can never be achieved in the monolithic specification systems that HBCL differentiates itself against, because such systems have a logical framework designed around the axiomatization of the problem space that they are trying to address. Event-B or TLA+ are not suitable for linking truths about abstract mathematics or physics with arbitrary computation models or language semantics.

We also note that, while we have factored the traditional domain of specification languages into pure formal logics and ontologies, we are still taking the correctness of the logic's deductive system and implementation on trust. We cannot reap the full benefits of maximum possible certainty (within the limitations of Gödel) in the consistency and correctness of a particular proof assistant's kernel until and unless we use a proof assistant that is fully formalized in itself or another.

7.3.1.3 Soundness and completeness properties

In chapter 5, we showed how the structure of our formalization, using directly executable functions over σ-types, allowed us to infer directly the soundness and completeness of our language *up to* that of Coq. In chapter 6, we showed this empirically.

One weakness that emerges from the deep embedding approach is that, since soundness of the subject language (HBCL) is derived directly from that of the formalizing proof tool (Coq), the semantics of the *statement* of consistency are problematical without axiomatizing Coq in Coq. This has been tried experimentally in the literature, but there are no production systems that formalize Coq itself in this way. Given enough time, our view is that the deep embedding is ultimately superior to a shallow one, because it enables quantification over *everything* of interest, even the proof assistant itself. True, there are incompleteness problems that contain the scope of what can be said about a formalizing logic within itself, but to address the consistency of a target language by addressing the consistency of the formalizing logic is a general approach, that once, obtained, can work for any language.

7.3.2 Use of Coq

Formalizing a programming language in a proof assistant is a very time-consuming process. Unusually, in this project, we opted to formalize the language before using it. This decision has had some positive and some negative consequences. On the positive side, it has allowed the dependent type theory to influence how we approached setting out the semantics: this provided insights into the level of parametrization achievable in specifying semantics that we might not otherwise have gained. It also obviates distracting arguments about which kind of semantic style is preferable. Further, it has driven our investigations more deeply into the issues of what specification is, what a specification language should achieve, and what a host logic can achieve when reasoning about such a language. Most importantly, it has motivated an economy of formalisms in general. We do not try to embed a semantic style or a temporal logic or a formal ontology: we deal directly with a thin axiomatization layer of observation points in space-time. This economy of 'helper' formalisms has led us to an understanding of the business of formalizing logics and ontologies of the physical world in a mutually inductive system. Pre-HBCL provides only this ontology: the proof assitant's logic is adequate to describe any computation based on this ontology; full HBCL is a convenience that helps us do so concisely. The economy of Pre-HBCL presents a narrow interface for any other extended physical ontology that we might like to formalize and reason about in the same proof assistant.

Kernels of individual proof checkers, as well as axiomatizations of the world and of specifications, are a potential weak point in arbitrarily reliable verified systems. This has unearthed the idea of a drive to capture our intuition of a system in a formal logic with an arbitrarily high level of reliability, within the confines of our understanding of Gödel's incompleteness theorems. This is an end point for the present thesis and a departure point for further ideas for research. In the absence of a formalization-driven development, it is unlikely we would have come across these insights.

On a purely practical level, developing a new language with a tool such as Coq has been a very slow process. We describe here the particular difficulties we have had with Coq. Some difficulties with individual tools and languages, such as Coq and OCaml, are peculiar to the systems in question, but the problems of working with partially verified code in a proof assistant are more generally applicable to any deductively verified program in any proof assitant. As a result, the impetus inevitably shifted during the course of the research from fully proving every lemma to producing a working interpreter to develop the language. We believe it is more useful to have a complete partially verified interpeter than a fully verified fragment of one. This, in turn, introduced delays of its own, because we had to debug through the obfuscation of code export into OCaml. As we discovered in chapter 6, OCaml and Coq have syntactic and semantic mismatches (mostly to do with dependent typing and module parameter types) that conspire to make this an exceptionally frustrating process. The OCaml debugger will not work with certain Coq module functors unless the OCaml is heavily patched after extraction. Coq itself currently lacks universe polymorphism (except in some experimental versions), which has forced us to keep harmonic boxes and coordination modules apart until after they have been exported into OCaml. Correcting code in Coq, followed by Coq compilation, export, OCaml patching and looking through OCaml printing functions, is a painful process.

7.4 Further work

In this chapter, we made a number of observations about how, and the extent to which, the work in this thesis advances our main hypotheses. The highest priority would be to fully prove admitted lemmas, and further press the practical points about physical implementations through a selection of fully proven morphisms. This amounts to a very substantial amount of work, with a large overlap with the field of verified compilers. To obtain higher levels of assurance regarding the correctness of a proof assistant itself is an even larger undertaking. Additional static analysis of co-inductive bisimulation predicates is also desirable, along with formalized tools to further underline our points about replication and other reflexive HBCL transformations. Finally, as we discussed in detail in section 6.9, a compiler would be needed in order to write larger examples, and the embedding of at least one more sophisticated functional box language would allow these examples to operate at a higher level of abstraction. We now describe a number of improvements that could be made that further these priorities. The final two sets of headings involving persistent state and multiple axiomatizations would be helpful in making HBCL into a realistic production-prototype system with the maximum amount of interoperability with other proof tools, but do not play a central part in furthering the claims of the present work.

7.4.1 Hardware formalization

7.4.1.1 Hardware simulation

We would like to expand the scope of the HBCL implementation so that `hinsts` and `cinsts` can be simulated, including mobile hardware configurations, physical faults and changing configurations. This is a complex task but one that could be built up incrementally with successive transformations from big-step semantics of box languages to small-step semantics of implementations. It suggests that an eventual connection with deductively verified hardware would be desirable, situating such work in the emerging discipline of verified stacks. The simulation of hardware would enable robust reasoning about the resilience of computations in the presence of faults.

7.4.1.2 Plesiochronous implementation tolerance

Hbcl respects a single unified global clock. Although we can agree that such a clock exists (as a weighted average of Cæsisum reference clocks), the uncertainty principle means that we can never know with complete precision when a tick of the clock occurs, still less arrange that it be propagated with complete accuracy. Thus practical implementations must tolerate clock skew. Communications engineers are familiar with the plesiochronous and synchronous digital hierarchies (pdh [7] and sonet [179]/sdh [8]). These offer ways of arranging the clocking rates of diverse communications networks so that serial links do not produce data at a rate that cannot be accepted by onward links on account of the onward link being clocked at a fractionally slower rate. Plesiochronous networks allow progressively larger amounts of padding in a hierarchy of progressively faster links to deal with these different clock rates, while synchronous networks rely on atomic time standards to ensure universal clock composability. In synchronous networks, the hierarchy of network elements is not needed to ensure consistent clocking, and complex network topologies can be created with ease.

For our purposes, this terminology is very unhelpful. At first sight, our formalism looks a little like a synchronous communications network, but synchronous communications networks are all about harmonizing clock *rates*, not absolute times and latencies. At the same time, the idea that a synchronous network is completely synchronous is a little misleading, as no two clocks can be perfectly synchronized. Two atomic clocks can operate a synchronized network in practice because a second is defined in relation to the properties of the electron quantum states of Cæsisum atoms. These properties are constants of physics, and thus there will be no long term frequency drift between two atomic clocks,[4] although there will be long term drift in absolute *time* recorded because of thermodynamic uncertainties and the 'random walk' effect. Our model requires absolute knowledge of time in our fixed coordinate system in spacetime, and so in practice, our network of clocks must exist in a hierarchy derived from the reference clocks.[5]

This is the same principle as that involved in how the Network Time Protocol (ntp) [163] of ip [5] networks operates, except without the relatively large queueing uncertainties of using a packet-switched network for this purpose. Within a clock distribution network, elements close to each other will have smaller drifts with respect to each other than with respect to remoter elements. This amounts to a plesiochronous network of *absolute time*, as opposed to the usual understanding of a plesiochronous network of *frequencies*. Ensuring that this plesiochronous network has a state space that is a witness to the theoretical synchronous state space of hbcl's coordinate system is an engineering

[4]In the same relativistic frame.

[5]The clock ensemble as a whole will gradually migrate from the time axis in the reference frame, but this drift is unknowable and too small to be of any engineering concern. It does not affect the mutual consistency of systems relying on the absolute time, which all reference the same clock ensemble.

challenge that in practice will require the development of some sophisticated tools. This forms another area of further work.

7.4.2 Mutual recursion of coordination languages

Given that we have a system for binding arbitrary box languages to HBCL boxes, we would like to explore the embedding of a heterogeneous expression or coordination language in an HBCL box. It would then be possible to consider the *mutual* nesting of such languages. For example, if we explored this with Hume, we might investigate embedding Hume in the box language abstraction, and HBCL within a Hume box in hierarchical Hume [93].

7.4.3 Reliability engineering tools

We have asserted that HBCL is an ideal substrate for reliability engineering to an arbitrarily low probability of failure, especially of critical control systems. While we have demonstrated HBCL with a simple expression language that reduces to a hardware description language, if we substitute a full language for this combinatorial Boolean language, HBCL will yield a much higher granularity of computation, in the pure functional paradigm of a language like Haskell. This should allow the synchronous fault tolerance techniques of stand-alone computers to be extended to sychronous distributed computing. We now discuss the tools needed to make this a practical reality.

7.4.3.1 Automated replication

The replication transformation of section 6.8.5, with its insertion of fan-outs and voters, is a simple process and could be easily automated. Through combining this with overlapping replication regions, single points of failure can be completely eliminated by replicating both voters and fan-outs. It is necessary to be mindful of the considerations of section 7.4.4 in doing this: these may add complexity to the replication transformations, involving the insertion of extra HBCL code beyond fan-outs and voters.

7.4.3.2 Fault injection, and stochastic analysis tools

In section 6.8.5, we saw a triple-modularly redundant multiplier with voters. In order to demonstrate that this arrangement provides the degree of resilience we expect, we would need a mechanism to inject faults into the system. Once mapping logic to hardware, we would also require stochastic analysis tools to compute the overall reliability of a system given the mappings to hardware that had been made. If a piece of hardware failed, or a local disturbance such as a data centre burning down occurred, the resilience of the system would depend on the geographical distribution of replicas (with a direct trade-off against latency) and the amount of sharing of physical nodes among logical processes. This would require analysis using Bayesian networks [134], dynamic fault trees [69], Markov models [117] or similar abstractions. Such models would need to be embedded in a proof assistant if we wanted to verify the consistency of the statistical treatment. A further extension to such tools would include a realistic treatment of how multiple sensors of analogue quantities on the system boundary can be reconciled, carrying the removal of single points of failure into the domain of the physical environment. The method by which multiple sensors and actuators are reconciled is an application-specific matter, but one for which it may be possible to provide some generic tools for useful design patterns.

7.4.3.3 Automated evolutionary tuning of fault tolerance transformations

A natural progression from a stochastic analysis tool is an heuristic process for finding a mapping of logic to hardware under reliability constraints. The level and combination of replication processes are factors in this — so too are the mappings of components or use of hardware of different known reliabilities. The dispersal of replicas in space can also be varied. Since there are so many factors with complicated patterns of interaction, an evolutionary approach may well be the most effective at finding efficient configurations for a given target reliability.

7.4.3.4 Interface to asynchronous reliable systems

At the boundary with an aysnchronous reliable system, steps must be taken to ensure that the single points of failure can be removed, just as they can within an HBCL development. Where an interface to a robust system operating by asynchronous consensus is concerned, an aysnchronous consensus protocol must operate on the boundary of the HBCL domain so that an agreement on the arrival ordering of data can be reached. Again, it would be possible to develop tools to do this, perhaps implementing common aysnchronous consensus routines in HBCL.

7.4.3.5 Graphical design tools

It is sometimes difficult to visualize HBCL programs, which makes a three-dimensional visualization tool desirable. Such a tool would be particularly useful in understanding the mappings of logical instances to hardware instances. It would also be helpful to be able to reduce a logical instance to a two-dimensional form, so that a replication transformation could be projected into the third dimension. This would emphasize the fact that the witness of the original program is obtained by a construction function over the various replicas, projected back onto a two-dimensional canvas.

7.4.4 Persistent state HBCL

One of the reasons for the choice of a pure functional paradigm is that it allows the meta-model to enforce very strong state consistency guarantees. This differentiates the present approach to fault tolerance redundancy from that adopted in a functionally impure language like Erlang, or an abstract machine-based methodology, such as the B-method. Where replication is concerned, the presence of persistent state in HBCL will require some extra implementation complexity, described in the following paragraphs, to prevent divergent latent state being built up by accident, but still in a way that hides these issues from the application programmer.

7.4.4.1 Feedbacks

Process feedbacks are essential. As a pure computational issue, they are needed because they are a coordination analogue to expression recursion, and hence essential for coordination languages generally. From a control systems perspective, we can give a less abstract example: without feedbacks it is impossible to integrate with respect to time, something which almost any real-time control system needs to do. However, if we transform cyclic sections of HBCL programs that are then subjected to faults, the states of different replicas will diverge. Compound faults may lead to there being no workable interpretation of the aggregate state.

There are two ways to deal with this. Either we can prohibit replication transformations of cyclic areas of graphs, insisting that at least two overlapping transformations be used instead, or we can introduce a repair mechanism. The former is problematical if we want to cater for the geographical distribution of replicas, as the latencies of synchronizing every cycle may be too great compared to the size of the cycle itself. In the latter case, we could arrange for this to work transparently to the programmer, by monitoring tracer FIFOS automatically generated by the replication transformation. These would vote on and judge supposedly identical feedback values *ex post facto*. If voting produced a null value, the offending cycle could be suspended and reset using the healthy cycles. During the reset period, the faulty cycle would have to cache new input data and then catch up from the image recovered from a healthy cycle, which would always be a few cycles out-of-date. This would add a good deal of complexity, but there are no reasons why it should be impossible; indeed, a pure HBCL solution should be possible as part of a replication transformation.

However many precautions of this sort are taken, there is always a risk that too many failures will mean that state is irrecoverably lost. In this case, a supervisor process for monitoring system health could reset all replicas at the same time with null data. Some fail-safe applications should be designed to recover gracefully from these situations in a way that safely restores control. This is the responsibility of the application programmer.

7.4.4.2 Off-line state extension

The system is also extensible to one that keeps persistent state, even when all replicas are suspended. In this case, a pair of FIFOs may be connected to a persistent storage device with constant-time look-up. Again, repair must be triggered on recoverable divergences, and reset or recalculation must be triggered on irreparable state divergences. It may also be desirable in this case for a supervisor process to run periodic consistency checks that are again transparent to the application programmer. All this adds another very considerable complication, but again, there is no reason why it could not be done, and done reflexively in HBCL as part of a replication transformation.

7.4.5 Multiple axiomatizations

The autonomous agent as an entity controlled by unobservable randomness raises questions about the computability of the brain and what it means to simulate consciousness. Much ink has been spilt on this topic, and we do not have any immediate plans to add to this. We prefer to focus future efforts on how we intuit the subject matter of computations from the viewpoint of multiple axiomatizations, treating consciousness (whatever that is) as a given, which allows us to choose whether we find an axiomatization of a system convincing or not.

There is an inevitable corollary of the elimination of single points of failure, which is that the formalization, formalizing logic and proof-checker should all be replicated as well, and equivalences be shown. This approach ultimately requires the formalization of one logic in another, or itself, and the axiomatization of parts of physics and of hardware descriptions in the same 'common denominator' logic. It then necessitates the proving of very many equivalences and morphisms, before permuting the logics and using a different one as 'common denominator'. Bootstrapping such a process without running into serious Gödel problems is an enormous challenge, whose parameters we discussed when we examined proof assistants in section 2.9.2 and following sections. We believe that by viewing the formalization of logic in physics as the primitive step (as opposed to physics in logic,[6] which is how the subject is always set up) we might situate the locus of incompleteness in a more intuitively comfortable place, namely our *belief* in the *consistency* of the laws of physics. Such a view hypothesizes a dual of Gödel's incompleteness theorems, one in which the inevitable incompleteness of *human* observations of all *possible* observations of the physical world is a limitative result in physics and cosmology, rather than in logic. Such a duality would assert that we cannot *know* that logic

[6]Specifically, first order logic and set theory.

is *any more* consistent than our *experience* allows us to know physics to be consistent and complete. This idea is bold and deserves to be properly investigated.

Multiple axiomatizations would also allow us to formalize fully the type system and semantics of the style in which we presented semantic rules in appendix C. It is difficult to keep such formalizations synchronized with the Coq formalization. Since the informal presentation is easier to understand than the Coq axiomatization, we ought to consider proving the two equivalent: otherwise, there is a risk that a presentation as understood on the page diverges from the formal model we use for reasoning.

7.5 Summary

We have developed HBCL as a recognizable programming language: we have provided syntax and semantics, and developed an implementation, some example programs and tested the results. Yet HBCL is a formulation or a formalization of a paradigm of engineering that is independent of a language or an attempt to formalize it. It is a way of specifying structured logical relationships between what happens in some restricted part of reality and something we would like to *make* happen in reality. It does this by axiomatizing reality in a single consistent frame of reference. Specifications of the mathematical relationships between observations and manifestations of values in such a schema are expressions of human free will. These are expressions to which engineers strive to give effect: we hope that we have provided a compelling new language for describing them.

References

[1] IEEE Standard 1076-2008 VHDL Language Reference Manual.

[2] IEEE Standard 1364-2001 Verilog® Hardware Description Language.

[3] IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual.

[4] ISO/IEC Standard 13568: Information Technology — Z formal specification notation — Syntax, type system and semantics.

[5] Internet Protocol: DARPA Internet Program: Protocol Specification, September 1981. RFC 791.

[6] *occam® 2.1 reference manual*. SGS-Thomson Microelectronics Limited, 1995. First published 1988 by Prentice Hall International (UK) Ltd as the occam 2 Reference Manual.

[7] ITU G.705: Series G: Transmission systems and media, digital systems and networks – Characteristics of plesiochronous digital hierarchy (PDH) equipment functional blocks, 2000.

[8] ITU G.783: Series G: Transmission systems and media, digital systems and networks: Digital terminal equipments – Principal characteristics of multiplexing equipment for the synchronous digital hierarchy, 2006.

[9] Embedding, *n. in* OED *online*, March 2014. Oxford Unviersity Press. Accessed 2 June 2014.

[10] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[11] Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[12] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An Open Extensible Tool Environment for Event-B. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer Berlin Heidelberg, 2006.

[13] Gul Agha and Prasanna Thati. An algebraic theory of actors and its application to a simple object-based language. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 26–57. Springer Berlin Heidelberg, 2004.

[14] Gul A. Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, Massachusetts Institute of Technology, 1985. Published as MIT Artificial Intelligence Laboratory Technical Report 844.

[15] Jesse Alama, Kasper Brink, Lionel Mamane, and Josef Urban. Large Formal Wikis: Issues and Solutions. In JamesH. Davenport, WilliamM. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 133–148. Springer Berlin Heidelberg, 2011.

[16] Kartik Anand, Ginestra Bianconi, and Simone Severini. Shannon and von Neumann entropy of random networks with heterogeneous expected degree. *Phys. Rev. E*, 83:036109, Mar 2011.

[17] Joe Armstrong. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 196–203, New York, NY, USA, 1997. ACM.

[18] Richard T. W. Arthur. *Natural Deduction: An Introduction to Logic with Real Arguments, a Little History, and Some Humour*. Broadview Press, 2011.

[19] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 2005.

[20] Henk P. Barendreght. *The Lamdba Calculus, its Syntax and Semantics*, volume 40 of *Studies in Logic: Mathematical Logic and Foundations*. College Publications, 2012. This is a re-publication of the original out-of-print 1984 North-Holland edition, of which there is also a 1985 revised edition.

[21] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, chapter 18, pages 1149–1238. North-Holland, Amsterdam, 2001.

[22] Bruno Barras. Coq en Coq. Rapport de recherche RR-3026, INRIA, 1996. In French.

[23] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets*. Vieweg Verlag, second edition, 2002.

[24] Timothy Bays. On Floyd and Putnam on Wittgenstein on Gödel. *The Journal of Philosophy*, 101(4):197–210, 2004.

[25] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer-Verlag London, 2008.

[26] Paul Benacerraf. What numbers could not be. *The Philosophical Review*, 74(1):47–73, 1965.

[27] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg, 2009.

[28] Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer Berlin Heidelberg, 2002.

[29] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, Amsterdam, 1986. Centrum voor Wiskunde en Informatica, North Holland.

[30] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, chapter 7, pages 187–210. Springer, 2004.

[31] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Number 53 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[32] Jean-Paul Bodeveix and Mamoun Filali. Type Synthesis in B and the Translation of B to PVS. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 350–369. Springer Berlin Heidelberg, 2002.

[33] Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In *Implementation and Application of Functional Languages*, volume 4449/2007 of *Lecture Notes in Computer Science*, pages 235–252. Springer Berlin / Heidelberg, 2007.

[34] George Boole. *An Investigation of the laws of thought, on which are founded the mathematical theories of logic and probabilities*. London: Walton and Maberly; Cambridge: MacMillan and Co., 1854. Reissued by Cambridge University Press 2009 in the Cambridge Library Collection series.

[35] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[36] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[37] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, June 1984.

[38] J. Richard Büchi. Symposium on decision problems: On a decision method in restricted second order arithmetic. In Patrick Suppes Ernest Nagel and Alfred Tarski, editors, *Logic, Methodology and Philosophy of Science Proceeding of the 1960*

International Congress, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966.

[39] Peter Calvert and Alan Mycroft. Petri-nets as an intermediate representation for heterogeneous architectures. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 226–237. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-23397-5_22.

[40] Luca Cardelli and AndrewD. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Berlin Heidelberg, 1998.

[41] Luca Cardelli, Simone Martini, JohnC. Mitchell, and Andre Scedrov. An extension of System F with subtyping. In Takayasu Ito and AlbertR. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 750–770. Springer Berlin Heidelberg, 1991.

[42] Nicholas J Carriero, David Gelernter, Timothy G Mattson, and Andrew H Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[43] Antonio Cau and Willem-Paul de Roever. A dense-time temporal logic with nice compositionality properties. In Franz Pichler and Roberto Moreno-Díaz, editors, *Computer Aided Systems Theory — EUROCAST'97*, volume 1333 of *Lecture Notes in Computer Science*, pages 123–145. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0025039.

[44] Francesco Cesarini and Simon Thompson. *Erlang Programming*. Sebastopol, CA: O'Reilly Media, Inc., 2009.

[45] Krishnendu Chatterjee, Arkadeb Ghosal, Thomas A. Henzinger, Daniel Iercan, Christoph M. Kirsch, Claudio Pinello, and Alberto Sangiovanni-Vincentelli. Logical reliability of interacting real-time tasks. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 909–914, New York, NY, USA, 2008. ACM.

[46] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[47] Adam Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 93–106, New York, NY, USA, 2010. ACM.

[48] Adam Chlipala. *Certified Programming with Dependent Types*. 2011. Version referenced is that of 17 January 2011, accessed on that date at `http://adam.chlipala.net/cpdt/`.

[49] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[50] Alonso Church. An unsolvable problem of elementary number theory. In Martin Davis, editor, *The Undecidable*, pages 89–107. Dover, 2004. Reprinted from The American Journal of Mathematics, 1936, Vol. 58, pp. 345–363 in collection published by Raven Press Books,1965; re-published in Dover edition.

[51] William Douglas Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, May 1981. Published as Technical Report 633.

[52] Paul Cockshott, Lewis Mackenzie, and Greg Michaelson. *Computation and its limits*. Oxford University Press, 2012.

[53] Mark Colyvan. *An Introduction to the Philosophy of Mathematics*. Cambridge University Press, 2012.

[54] Stephen A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, STOC '79, pages 338–345, New York, NY, USA, 1979. ACM.

[55] coq-ext-lib developers. coq-ext-lib repository. Public git repository. `https://github.com/coq-ext-lib/coq-ext-lib`, accessed 18 June 2014.

[56] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[57] Solange Coupet-Grimal. An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[58] John Darlington and Mike Reeve. ALICE a multi-processor reduction machine for the parallel evaluation CF applicative languages. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 65–76, New York, NY, USA, 1981. ACM.

[59] René David and Hassane Alla. *Discrete, Continuous and Hybrid Petri Nets*. Springer-Verlag, second edition, 2010.

[60] Jim Davies and Steve Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[61] Jared Curran Davis. *A Self-Verifying Theorem Prover*. PhD thesis, The University of Texas at Austin, 2009.

[62] René Descartes. *A discourse on the method of correctly conducting one's reason and seeking truth in the sciences*. Oxford University Press, 2006. Translated with an introduction and notes by Ian Maclean.

[63] D. Deutsch. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

[64] E. Dijkstra. In reply to comments. Letter in EW Dijkstra archive, no. 1058, University of Texas at Austin, 1989. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1058.html` Accessed 3 June 2014.

[65] Edsger Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In Friedrich Bauer, E. Dijkstra, A. Ershov, M. Griffiths, C. Hoare, W. Wulf, and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer Berlin / Heidelberg, 1976.

[66] Emanuele D'Osualdo, Jonathan Kochems, and C.-H. Luke Ong. Automatic Verification of Erlang-Style Concurrency. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 454–476. Springer Berlin Heidelberg, 2013.

[67] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Soter: an automatic safety verifier for erlang. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 137–140, New York, NY, USA, 2012. ACM.

[68] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The Coq Proof Assistant User's Guide, version 5.6. Technical Report 134, Institut National de Recherche en Informatique et en Automatique, December 1991.

[69] J.B. Dugan, S.J. Bavuso, and M.A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *Reliability, IEEE Transactions on*, 41(3):363–377, Sep 1992.

[70] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[71] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1 – 24, 1985.

[72] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33:151–178, January 1986.

[73] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275 – 306, 1987.

[74] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1982.

[75] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[76] Juliet Floyd and Hilary Putnam. A Note on Wittgenstein's "Notorious Paragraph" about the Gödel Theorem. *The Journal of Philosophy*, 97(11):pp. 624–632, 2000.

[77] Juliet Floyd and Hilary Putnam. Bays, Steiner, and Wittgenstein's "Notorious" Paragraph about the Gödel Theorem. *The Journal of Philosophy*, 103(2):pp. 101–110, 2006.

[78] Abraham A. Fraenkel. The notion "definite" and the independence of the axiom of choice. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 284–289. Harvard University Press, 1967.

[79] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 1–82. Harvard University Press, 1967.

[80] N.E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[81] Masahiro Fujita. Model checking and equivalence checking. In D. K. Pradhan and I. G. Harris, editors, *Practical design verification*. Cambridge University Press, 2009.

[82] Robin Gandy. Church's thesis and principles for mechanisms. In Jon Barwise, H. Jerome Keisler, and Kenneth Kunen, editors, *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 123–148. Elsevier North-Holland, Inc., 1980.

[83] Mauro Gaspari and Gianluigi Zavattarom. An Algebra of Actors. Technical Report UBLCS-97-4, Department of Computer Science, University of Bologna, May 1997.

[84] Simon J. Gay and Rajagopal Nagarajan. Communicating Quantum Processes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 145–157, New York, NY, USA, 2005. ACM.

[85] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34:3–25, 2009. 10.1007/s12046-009-0001-5.

[86] Herman Geuvers and Rob Nederpelt. N.G. de Bruijn's contribution to the formalization of mathematics. *Indagationes Mathematicae*, 24(4):1034–1049, 2013. In memory of N.G. (Dick) de Bruijn (1918–2012).

[87] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin Heidelberg, 1995.

[88] Kurt Gödel. On Formally Undecidable Propositions of Principia Mathematica and Related Systems I. In Martin Davis, editor, *The Undecidable*, pages 4–38. Dover Publications Inc., 2004. Corrected republication of 1965 edtion of Raven Press Books, Ltd. Translated by E. Mendelson from original German in *Monatshefte für Mathematik und Physik*, vol 38 (1931) pp. 173–198.

[89] M. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[90] Mike Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical report, University of Cambridge Computer Laboratory, 2001.

[91] A. Gravell and P. Henderson. Executing formal specifications need not be harmful. *Software Engineering Journal*, 11(2):104–110, 1996.

[92] Gudmund Grov. *Reasoning about Correctness Properties of a Coordination Programming Language*. PhD thesis, Heriot-Watt University, Edinburgh, March 2009.

[93] Gudmund Grov and Greg Michaelson. Hume box calculus: robust system development through software transformation. *Higher-Order and Symbolic Computation*, 23(2):191–226, 2010.

[94] Gudmund Grov, Robert Pointon, Greg Michaelson, and Andrew Ireland. On Hume Scheduling. Technical report, Heriot-Watt University, 2007.

[95] Florian Haftmann. From Higher-Order Logic to Haskell: there and back again. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 155–158, New York, NY, USA, 2010. ACM.

[96] Kevin Hammond, Greg Michaelson, and Robert Pointon. The Hume Report, Version 1.1. Technical report, School of Computer Science, University of St Andrews, 2007.

[97] Lin Han, Francisco Escolano, Edwin R. Hancock, and Richard C. Wilson. Graph characterizations from von Neumann entropy . *Pattern Recognition Letters*, 33(15):1958–1967, 2012.

[98] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[99] Ian Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, November 1989. Pub. Institution of Electrical Engineers for Institution of Electrical Engineers and British Computer Society.

[100] Eric C. R. Hehner. *A Practical Theory of Programming*. Department of Computer Science, University of Toronto, 2012-7-28 edition edition, 2012. First Edition published by Springer-Verlag Publishers, New York, 1993.

[101] T.A. Henzinger, C.M. Kirsch, E. Marques, and A. Sokolova. Distributed, Modular HTL. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 171 –180, 1-4 2009.

[102] Thomas Henzinger, Christoph Kirsch, and Slobodan Matic. Schedule-carrying code. In *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 241–256. Springer Berlin / Heidelberg, 2003.

[103] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: a time-triggered language for embedded programming. In *Proceedings of the IEEE*, pages 166–184. Springer-Verlag, 2001.

[104] Carl Hewitt. Actor model of computation: Scalable robust information systems. *arXiv.org (Cornell University Library) eprint*, (arXiv:1008.1459), 2014. Version 32.

[105] Carl Hewitt and Henry Baker. Laws for communicating parallel processes. In Bruce Gilchrist, editor, *International Federation for Information Processing. Congress (7th : 1977 : Toronto): Information processing 77 :proceedings of IFIP Congress 77, Toronto, August 8-12, 1977*, volume 7, pages 987–992. International Federation for Information Processing, Amsterdam; Oxford: North-Holland Publishing Company, 1977.

[106] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[107] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[108] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[109] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1998.

[110] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 262–272. Springer Berlin Heidelberg, 2003.

[111] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.

[112] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 410–423, New York, NY, USA, 1996. ACM.

[113] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.

[114] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. *SIGPLAN Not.*, 45(1):223–236, 2010.

[115] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 354–369. Springer Berlin / Heidelberg, 2009.

[116] Juvenal. *Satire 6*. Cambridge Greek and Latin Classics. Cambridge University Press, 2014.

[117] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[118] Paul H. J. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989.

[119] Stephen C. Kleene. Recursive predicates and quantifiers. In Martin Davis, editor, *The Undecidable: Basic Papers on Undecidable Propostions, Unsolvable Problems and Computable Functions*, pages 255–287. Dover, 2004. Reprinted from Transactions of the American Mathematical Society, 1943, Vol. 53, No. 1, pp. 41–73 in collection published by Raven Press Books, 1965; collection re-published in Dover edition.

[120] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 460–467, June 1992.

[121] Hermann Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[122] Robbert Krebbers and Freek Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In JamesH. Davenport, WilliamM. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 301–303. Springer Berlin Heidelberg, 2011.

[123] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[124] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6:254–280, April 1984.

[125] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.

[126] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[127] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and System*, 4(3):382–401, July 1982.

[128] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[129] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43:363–446, 2009. 10.1007/s10817-009-9155-4.

[130] M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Generating a Petri net from a CSP specification: A semantics-based method. *Advances in Engineering Software*, 50(0):110 – 130, 2012.

[131] Hans-Wolfgang Loidl and Steffen Jost. Improvements to a Resource Analysis for Hume. In Marko van Eekelen and Olha Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis*, volume 6324 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15331-0_2.

[132] M. Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, 1995. Version accessed is a revised electronic version of the book downloaded from `http://www.di.unito.it/~greatspn/GSPN-Wiley/` on 10 June 2010.

[133] Martin-Löf. *Intuitionistic Type Theory: Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980*. 1984.

[134] H. F. Martz, R. A. Waller, and E. T. Fickas. Bayesian reliability analysis of series systems of binomial subsystems and components. *Technometrics*, 30(2):143–154, 1988.

[135] David May. Occam. *SIGPLAN Not.*, 18(4):69–79, April 1983.

[136] David May. *The XMOS XS1 Architecture*. XMOS Limited, 2009. `http://www.xmos.com/download/public/The-XMOS-XS1-Architecture%28X7879A%29.pdf`. Accessed on 18 June 2014.

[137] T. F. Melham. *Higher Order Logic and Hardware Verification*. Number 31 in Cambridge Tracts ion Theoretical Computer Science. Cambridge University Press, 1993.

[138] L.F. Menabrea. Sketch of the Analytical Engine invented by Charles Babbage Esq. from the Bibliotèque Universelle de Génève, No. 82, October 1842. In Richard Taylor, editor, *Scientific memoirs, selected from the transactions of foreign academies of science and learned societies, and from foreign journals*, volume 3, chapter 29, pages 666–731. Richard and John E. Taylor, 1843. Translated and with notes by Augusta Ada King, Countess of Lovelace.

[139] R. Milner. A calculus of communicating systems. Technical Report ECS-LFCS-86-7, University of Edinburgh, August 1986. First published by Springer Verlag as Vol. 92 of *Lecture Notes in Computer Science*.

[140] Robin Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, January 1993.

[141] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, Cambridge, 2009.

[142] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[143] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[144] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[145] J Strother Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer Berlin Heidelberg, 2003.

[146] Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 265–280. Springer Berlin Heidelberg, 2011.

[147] Matthew Naylor and Colin Runciman. The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA. In *Implementation and Application of Functional Languages*, volume 5083/2008 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008.

[148] Isaac Newton. *The Principia*. University of California Press, 1999. Translated by I. Bernard Cohen and Anne Whitman, assisted by Julia Budenz. Contains Guide to Newton's Principia by I. Bernard Cohen.

[149] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle HOL: A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2011.

[150] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[151] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[152] Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In Marc Bezem and JanFriso Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer Berlin Heidelberg, 1993.

[153] Giuseppe Peano. The principals of arithmetic, presented by a new method. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 83–97. Harvard University Press, 1967.

[154] Roger Penrose. *The Road to Reality*. Vintage Books, 2005.

[155] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition: New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol. 1, 1966, pp. Suppl. 1, translated by C. F. Greene.

[156] Simon L. Peyton Jones, Chris Clack, John Salkild, and Mark Hardie. GRIP—A high-performance architecture for parallel graph reduction. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 98–112, London, UK, 1987. Springer-Verlag.

[157] Gordon Plotkin. A Powerdomain Construction. In *SIAM J. Comput.*, volume 5, pages 452–487. 1976.

[158] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 311977-nov.2 1977.

[159] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Dover, 2006. Orginally published by Almqvist & Wiksell, 1965.

[160] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In Laurent Kott, editor, *Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer Berlin Heidelberg, 1986.

[161] Claus Reinke. Haskell-coloured petri nets. In *Implementation of Functional Languages*, volume 1868/2000 of *Lecture Notes in Computer Science*, pages 165–180. Springer Berlin / Heidelberg, 2000.

[162] Didier Rémy and Jérôme Vouillon. Objective ml: An effective object-oriented extension to ml. *Theor. Pract. Object Syst.*, 4(1):27–50, January 1998.

[163] RFC5905. Network time protocol version 4: Protocol and algorithms specification, June 2010.

[164] RODIN. Rodin project homepage.

[165] Victor Rodych. Wittgenstein's inversion of gödel's theorem. *Erkenntnis*, 51(2-3):173–206, 1999.

[166] Carlo Rovelli. Relational quantum mechanics. *International Journal of Theoretical Physics*, 35(8):1637–1678, 1996.

[167] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, (CSL), Stanford Research Institute (SRI) International, September 2001.

[168] Steve Schneider. *Concurrent and Real-Time Systems: The CSP Approach.* John Wiley & Sons, Ltd, 2000.

[169] Eckehard Schnieder, Mourad Chouikha, Stefan Einer, and Michael Meyer zu Hörste. Basysnet – an integrated approach for automated control system development. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472/2003 of *Lecture Notes in Computer Science*, pages 352–362. Springer-Verlag Berlin Heidelberg, 2003.

[170] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3 and 4):379–423 and 623–656, July/October 1948.

[171] W. Sieg. Calculation by man and machine: Conceptual analysis. In Wilfried Sieg, Richard Sommer, and Carolyn Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, number 15 in Lecture Notes in Logic. A.K. Peters, Ltd for the Association for Symbolic Logic, 2002.

[172] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

[173] Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74464-1_16.

[174] Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. Unpublished. Accessed at `http://mattam.org/research/publications/drafts/univpoly.pdf` on 22 March 2014, 2014.

[175] Martin Strecker. Formal Verification of a Java Compiler in Isabelle. In Andrei Voronkov, editor, *Automated Deduction—CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Berlin Heidelberg, 2002.

[176] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[177] Alfred Tarski. *Logic, Semantics, Metamathematics: Papers from 1923-38*, chapter VIII: The concept of truth in formalized languages, pages 152–278. Hackett Publishing, 1983. Translated by J. H. Woodger.

[178] Frank Stephen Taylor. *Parallel functional programming by partitioning*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1996.

[179] Telcordia Technologies. Synchronous Optical Network (SONET) Transport Systems: Common Generic Criteria. Technical Report GR-253, Telcordia Technologies, 2009.

[180] C. Tofts. A temporal calculus of communicating systems. Technical Report ECS-LFCS-89-104, University of Edinburgh, December 1989.

[181] Chris M. N. Tofts. Timing concurrent processes. Technical Report LFCS report ECS-LFCS-89-103, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, December 1989.

[182] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing certified functional code from inductive specifications. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin Heidelberg, 2012.

[183] David Trachtenherz. Formal semantics of modular time refinement in AutoFocus. *Computer Science - Research and Development*, pages 1–20, 2011. 10.1007/s00450-011-0148-2.

[184] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12:469–510, 7 2002.

[185] A. Trybulec. The MIZAR-QC/6000 Logic Information Language. *ALLC Bulletin: Association for Literary and Linguistic Computing*, 6(2):136–140, 1978.

[186] A. M. Turing. On computable numbers, with an application to the *Entscheidungsproblem*. In *Proceedings of the London mathematical society*, volume 42, pages 230–265, 1936.

[187] A. M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, s2-45(1):161–228, 1939.

[188] D. A. Turner, J. Fairbairn, D. Park, P. Wadler, B. A. Wichmann, and M. H. Rogers. Functional programs as executable specifications [and discussion]. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522):pp. 363–388, 1984.

[189] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, number 34 in Annals of Mathematical Studies, pages 43–98. Princeton University Press, 1956.

[190] J. von Neumann. First draft of a report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

[191] John von Neumann. *Mathematical Foundations of Quantum Mechanics*. Princeton University Press, 1955. Translated by Robert T. Beyer.

[192] Ludwig Wittgenstein. *Remarks on the Foundations of Mathematics*. Basil Blackwell, 1978. Third edition. Translated by G. E. M. Anscombe.

[193] Hector Zenil, editor. *A Computable Universe*. World Scientific, 2013.

[194] Ernst Zermelo. Investigations in the foundations of set theory I. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 199–215. Harvard University Press, 1967.

Appendix A

Design of HBCL

A.1 Design space of HBCL

Full HBCL is intended to define the desired *behaviour* of computation systems as viewed from the outside, using computable functions for prototyping a specification. This serves two other purposes beyond prototyping. First, the prototype function can go on to form part of the existential proof that the causative relationship given by the specification is realizable by a Turing-equivalent computer, when we characterize that relation as a predicate in an embedding of Pre-HBCL in a host logic. Second, it can act as a starting point for realizing a practical implementation by stepwise refinement.

It is not especially helpful to think of HBCL as a language for hardware design or as a piece of software. Unlike most hardware designs, correctness is defined in relation to a (single) *external* clock, rather than an arbitrarily drifting internal quartz oscillator. Unlike a software design, it is a direct prescription of something that can be made to happen in the real world, not a set of instructions for refining the behaviour of a *particular* abstract or physical machine. The specification may be *realized* by stepwise refinement using any combination of hardware or software. As a corollary of this approach, we deny that anything described with HBCL has *agency*. Agents exist beyond the system boundary, and their behaviour, including any component of randomness, is a parameter observed by an HBCL specification and interpreted deterministically. This is a fundamental difference between HBCL and methods that use process calculi to frame high level specifications.

A.2 OID semantics

A.2.1 Why OID semantics and an ontological approach?

One of the design objectives of HBCL was not only to formalize the logical specification language, but also to consider how one might formalize its relationship to human intuition, and to do so with an extremely thin layer of ontological formalization: we do not wish to digress into the selection of formal ontologies, or worse, become embroiled in epistemological debates. Why this approach? Specification is ultimately a human activity. Computers may assist in specifying something, but specification is about minds communicating, ideally in the most structured way possible, their prescriptive views for how some part of the informational or physical worlds should evolve, and then being able to record what is agreed with convincing authority.

To cite a concrete example, consider a box with wheels, containing a motor, a brake, microprocessor, a start button, and some supporting systems. Suppose it is desired to specify that, if the button is pressed (by whatever agency) in some particular time interval, the box will move with some prescribed kinetic profile within prescribed tolerances. If one wishes to specify this system in natural language, one might write a lengthy report, containing some engineering designs and some pieces of code in a computer language. However, some key issues, such as how one can uniquely refer, within the logic, to instances of these boxes and their position in time and space, are likely to be specified in an *ad hoc* way, if at all. The specification might refer to serial numbers, some arbitrary time base, and some arbitrary spatial coordinate system. All these are barriers to interoperability, since different organizations will choose different ways to instantiate systems for keeping track of specifications and instances (of the wheeled boxes, for example). The interaction of different pieces of equipment detailed in different industrial reports becomes subject to informal and weakly specified semantics.

Clearly, for many applications, this degree of rigour is inappropriate and disproportionate. However, in the field of safety-critical and high integrity systems, absolute clarity of a specification's subject matter is essential, since system correctness is a property that only makes sense if defined relative to some original specification. Ambiguous informal semantic glue between heterogeneous systems is a source of error and potentially dangerous failures.

This argument motivates the need to provide a clear spatio-temporal and object reference system for an ontologically aware language such as HBCL. To be clear, HBCL is not an ontology itself. To reiterate, this is not the place to explore the philosophical subject matter involved, nor to formalize a language in an upper ontology, let alone to debate the epistemic intricacies of what the informational part of the model actually is. However, HBCL does attempt to illustrate how a computer specification language might interface to such ontologies, and through such, to other domains of system specification involving spatial issues.

A.2.2 Monotonicity principle of OID semantics

Once an OID is appended to an OID arc, we require that it persists for as long as the root does. If that root is the ITU/ISO OID root, to all intents and purposes, it exists in perpetuity. The semantics only allow a specification to be added to the OID tree if its references to other parts of the OID are consistent. Since parts of a specification may only ever be added, never taken away, problems with broken dependencies are impossible. The physical justification of this axiom is the arrow of time. Indeed, values at different times can be uniquely identified by appending a temporal OID component to memory evolution.

A.2.3 Sandboxing and the semantics of composition

The meta-instantiation of a version of HBCL onto a particular OID root should be a static parameter of the formalization, since, intuitively, an arbitrary prepended arc of natural numbers to identifiers should have no functional impact on anything that might be specified with the system.

This observation indicates a related one: namely, that prepending of identifier arcs may be a general method to instantiate repetitive structure. The appended part of the OID may be the same for every copy of a specification, but it gains its unique identity from what is prepended to it. This principle extends to sub-assemblies. Theorems about repeated structures of this kind can be parametrized by the prepended instance OID, and instantiated along with their subject matter. At the point where we have a root for the whole of HBCL, the specifications beneath it become the ‘type’ of reality, as it relates to the specification. Specifications with placeholder roots have a dependent existence; specifications transitively related to the ITU/ISO OID root have an independent existence. These semantics are dependent on the list structure of individual OIDs and the tree structure of the category of OIDs.

A.2.4 Static and dynamic OID semantics

Most of the OID semantics given here are static, in the sense that they declare objects that perdure, rather than in the sense of parsing source code. Their purpose is to specify the structural relationships between memories interposed between harmonic boxes and FIFOs. Dynamic OID semantics are a function of the environment at input interfaces, and the static specification of the harmonic boxes, FIFOs and memories. The dynamic semantics of HBCL given in chapter 4 and appendix C, being defined over OID types, furnish such a set of semantics. Should we have different versions of these semantics, we might give the semantics themselves OIDs in some meta-schema, though at present this would amount to over-engineering. Input data, along with the dynamic semantic rules, uniquely specify the (potentially infinite) counter-indexed state data appended to the memory instance OIDs. This infinite structure can be specified because the input environment is characterized by a coinductive type which acts as a parameter. The resulting infinite structure has precisely one inhabitant in the real world, giving a kind of typed reality. It is not intended that every value that has ever been defined in this way would

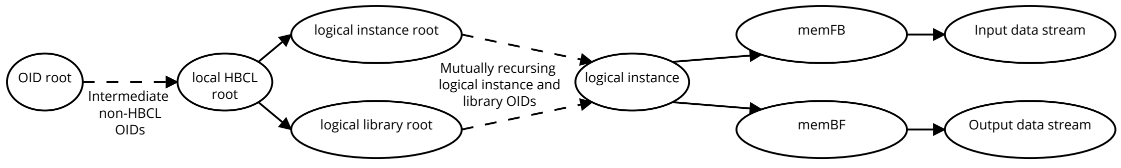


Figure A.1: OID layout for logical instances and libraries

be *recorded* anywhere, but it nevertheless has a reality within our formalism. Issues such as the creation and destruction of physical and logical objects are deliberately neglected at this stage, but can be dealt with fairly easily by adding times of validity, and hence temporal modality, to such specifications without violating the monotonicity principle of OIDs.

A.2.5 Qualitative heterogeneity in the OID string

Since OIDs are just strings of natural numbers, there is no facility for giving a type to different parts of the string, in the way that one can with, say, LDAP schemata, so we need a Gödelesque encoding. To deal with this, zero is treated as the universal escape number, and the number following this dictates how the next part of the string is to be treated. These arbitrary numbers are static parameters of the formalization: there is no reason to hard-code them. The list structure of OIDs means that many of the annoyances in parsing escape sequences in binary sources do not arise. Notwithstanding this, escape sequences always look ugly and unreadable. This ugliness can be largely hidden from view in the formalization by predicate subtyping of OIDs, and in HBCL source, by the use of textual aliases and automatic allocation of OID numbers according to static rules.

The four basic OID types are given by the type of terminating OID (after the last escape). These are the OID types for logical instances, logical libraries, and input (`memFB`) and output (`memBF`) memories. For the OID to be well-formed, it has to be escape-rooted or appended to a well-formed OID of a type permitted by the semantics. These static semantics of well-formedness constitute the definition of 'OID semantics', and correspond to the superficial textual static semantics of HBCL's coordination language. We show the layout of which type of OID component may be appended to a given arc in Figure A.1. The HBCL root may be instantiated on arbitrary arcs. The last element on any HBCL arc represents the data values for a particular memory, on an infinite set of time-indexed arcs according to the dynamic OID semantics.

Predicate subtyping of nested maps allows a compact representation of part of the OID tree in which ugly details such as escape sequences in the OIDs are implicit.

A.2.6 Rootless references

Within the semantics of HBCL, references to an OID may start with a logical instance or library OID, in which case, they are OID *fragments*, and identifiable as fragments because they start with an escape character. References to fragments starting with *instances* are implicitly prepended with the scope of the referencing instance's OID. References starting with *libraries* are taken to be prepended with the OID of the local scope or, if undefined, the first scope between the current scope and the root of the OID tree in which the library in question *is* defined. This is a subtlety of HBCL semantics, *not* OID semantics, since OIDs always have a transitive connection to the universal OID root.

A.3 Pre-HBCL: defining timed observables

HBCL is a language for realizing causative relations between timed inputs and outputs that can be implemented using computable functions. These inputs and outputs are defined as discrete streams of the most likely idealized instantaneous observations that could be made in the environment as timed by a global clock. There are no processes in this model, and no interprocess communications: it is simply a set of relationships between timed observable values. We call the inputs 'observations' and the outputs 'manifestations', and define each with an HBCL 'memory'. These memories are not abstractions of silicon, but rather perdurant abstractions of some point in space with which can be associated a persistent observable value for an externally defined time interval. A natural physical analogy is with the electrical potential of a pin.

These observables and manifestations of values constitute the signature of 'logical instances'. They form a thin temporal layer (on an arbitrary type system) that can be embedded in a proof assistant and reasoned about using predicates over inputs and outputs, and related within the proof assistant to some arbitrary axiomatization of an application-specific problem. This is Pre-HBCL: HBCL's handle on static OID semantics. Pre-HBCL says nothing about the relationships between these observables. This is left to the logic in which Pre-HBCL is embedded, concrete HBCL (see below), or some other coordination language that satisfies the instance signatures of Pre-HBCL.

Pre-HBCL also provides a conceptual framework for mapping logical instances to hardware. This is not part of the executable language but is rather an ontology layer that maps types and 'memories' to identifiable objects that may have spatial and temporal attributes. This might be referred to as the *pragmatics* of the language, a term that is common in linguistics but rare in computer science. We would like to extend our OID schema to refer to more general pieces of physical hardware to bring these pragmatics within the language, and to axiomatize enough basic physics to start to reason about this hardware spatially. A full development is beyond the scope of the present work.

A.3.1 Instance signatures

Pre-HBCL defines a number of structures for instantiating and organizing memories, without specifying the computationally causative relationships between them. These are:

- **Logical instance signatures** Signatures, or *fingerprints*, are sets of input and output memories of a logical instance, where a logical instance is a black box that houses some computable and provably terminating function. Such a function is any injective mapping from observed inputs over time to the values that should be manifested as outputs over time.

- **Hardware instance signatures** These have the same fingerprints as logical instances, but do not have causative logic associated with their inputs and outputs. Rather, they expect to have the input and output memories of logical instances *mapped* to them. This is to allow the model to be extended to give localizations to physical inputs and outputs and specify the physical layer (*cf.* layer one of the OSI model), so that the host logic in which HBCL is embedded can reason about the interaction of an HBCL program with its environment.

- **Configuration instances** Configuration instances are mappings of logical instances to hardware instances that are valid for some particular interval of time (which may stretch infinitely into the future).

All of these types of instances may be accommodated in library scopes that defer their instantiation: they can be seen in the examples given in section 3.5. An *instance reference* shows that the instance in question has an existence that is independent of the instance in which the reference occurs. However, given that the present development is concerned with a language for specifying computation only, we omit a rigorous development of hardware and configuration instance semantics.

A.4 Full HBCL: a coordination meta-language

The coordination language of full HBCL is an executable language for expressing realizations of specifications that satisfy Pre-HBCL signatures. It formalizes data flows between observables and manifestations as FIFOS, and formalizes computation as any computable function with guaranteed termination. The semantics of the coordination language express this idea by using the type system and strong termination implicit in the formalizing logic to abstract the formalization of *any* executable language with provable termination.[1] One of the purposes of the structure of the full coordination language is

[1]This includes languages that do not in the general case have provable termination, but whose inhabitants are expected always to be provided to HBCL along with a suitable termination proof, or the means of

to enable a systematic approach to strengthening a specification by transformation for fault tolerance. This is sketched in the final example given in this thesis (in chapter 6).

Full HBCL is therefore a language which satisfies the instance and manifestation fingerprint of instances of causation logic with the paraphernalia of FIFOs, boxes, nested instances, and other constructions that are explained in chapter 3. It would be possible for other coordination languages to specify the same behaviour using an appropriate binding to HBCL instance signatures or fingerprints (we use the terms interchangeably). The present concrete language is a functor on any number of expression languages, and we provide a simple example of such a language in this thesis.

Time within HBCL is axiomatically dense. In other words, it is representable as a rational number, and global time advances in discrete steps over the lowest common multiple frequency of all components in the system, although there is no lower limit on the size of these steps. All computations occurring at a point on this rational timeline are axiomatized to occur instantaneously.

Infinitely fast computation is unphsysical, so implementations must be witnesses of a given specification. This means that there must be an injective relationship between the evolution of an implementation through its state space and the evolution of the state space of the specification. Such implementations can be reflexive in HBCL or in some other hardware or software system.

A.4.1 Entropic sandwich

A reasonable question to ask of HBCL is how can it be as expressive as process calculi and other formalisms for modelling distributed systems: surely asynchrony is a spatial corollary inherent in any distributed system? The answer is that it does not occupy the same design space as high level distributed system modelling tools for dealing with the asynchrony of an *application*, neither is it designed as a hardware design language. It can be regarded as logical middleware that sits between these two design spaces, providing a substrate to high level abstractions that is better behaved than real hardware. Crucially, on account of the external clocking, HBCL is easily transformable for higher implementation reliability in a way that is completely transparent to the application developer. Hence, we stratify synchrony and asynchrony in what we call an 'entropic sandwich', illustrated in Figure A.2. The application programmer can be concerned with asynchrony between application objects in different clocked data streams. The HBCL implementer is concerned with preserving the total order (*given* application inputs) in spite of hardware that, owing to clocking uncertainties and physical irregularities, can only ever be approximately synchronous.

Non-determinism exists at the hardware level in the total ordering of distributed

deterministically constructing it from a suitable encoding of a proof object.

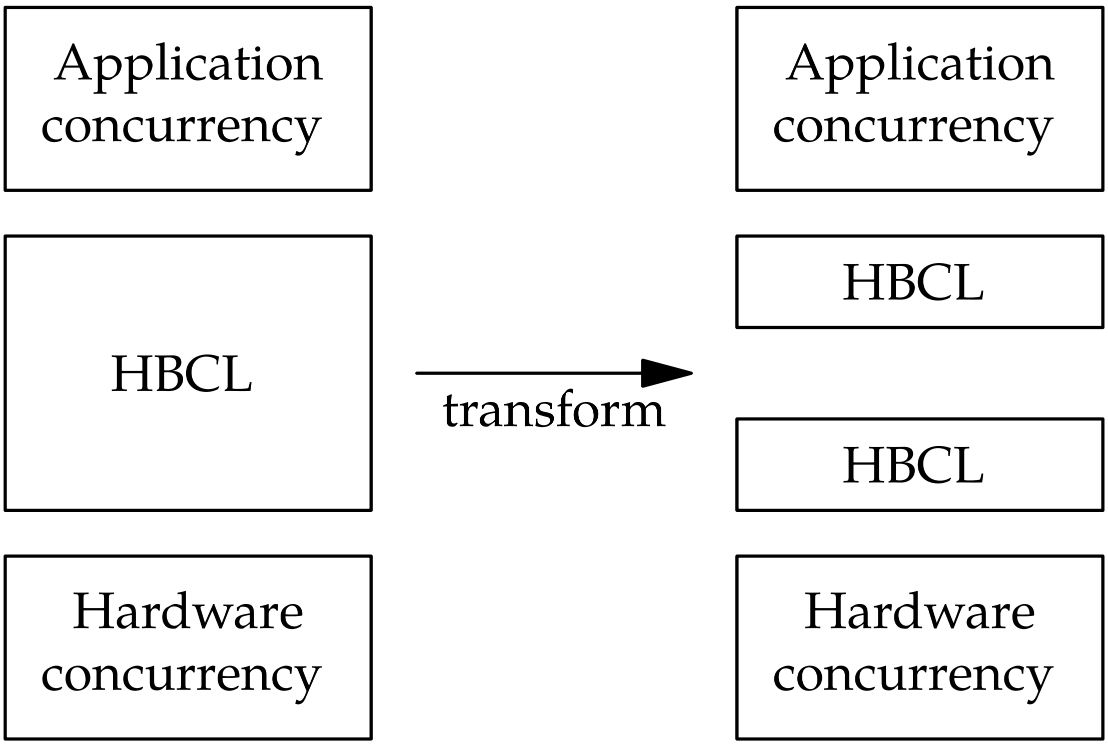


Figure A.2: Entropy sandwich

events. It also exists at the application level, but this non-determinism is *observed deterministically* by the semantics of HBCL. Correctness of HBCL executions is determined by a *total* order on its state space: entropy is an external *parameter*, and as such we view agency as a related concept to external entropy, in that the epistemic or human significance of either is not accessible to the semantics of HBCL.[2]

A.5 Full HBCL language structure

Figure A.3 shows the primitive structures of full HBCL, indicating how the full coordination infrastructure is built on top of type systems, timed and harmonic, and abstractions of untimed and timed box languages.

The type of interpreters is dependent on the final instantiated type of the coordination language. In order not to enshrine arbitrary choices (such as a type system) in the formalization, the whole language is set up in terms of *functors*. We do not include the spatial commitments of HBCL in this particular formalization.

The structure of Figure A.3 closely follows the module structure adopted in the Coq

[2]If we choose a suitably physical source of entropy, the significance of the information or noise therein is also opaque to human minds, and any idea of 'agency' becomes a metaphysical question that we deliberately avoid.

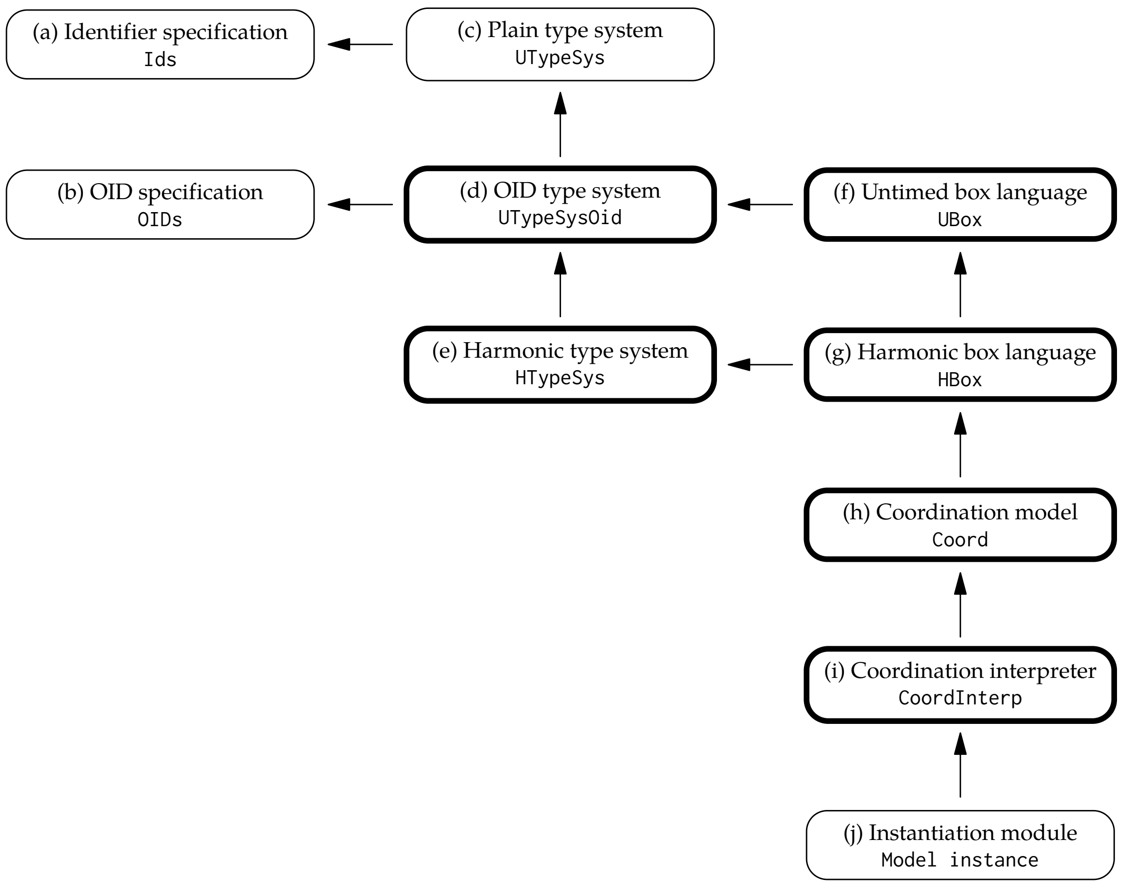


Figure A.3: Structure of full HBCL

formalization (see chapter 5), with each box representing a module. Boxes with a bold outline are module functors that require the modules on which they depend to be given as module arguments. Arrows indicate dependency relationships, and should be read transitively. This use of modules makes the final coordination functor *statically* parametric on arbitrary choices of type system and identifier specification. This is convenient for present purposes, but does have the limitation that it is impossible to quantify over multiple concurrent choices of type systems and identifiers. Generalizing this would introduce the need to provide equivalence classes and show morphisms that would obscure the main details of the coordination language that we are presenting. However, we do choose to make the language 'dynamically' dependent on untimed and harmonic box languages. This is because it *is* important to demonstrate HBCL as a meta-language and emphasize that the method of specifying the computable function inside a box is irrelevant to the temporal semantics of HBCL. This is done by using dependently typed records to contain the semantics of the language, with the result that the coordination language is formally quantified over any language that can be deeply embedded in the host logic. In further work, it may become desirable to remove module parametricity completely and use the object-oriented paradigm with dependently typed records throughout; this is discussed in section 6.9.

We now introduce the contents of the modules in Figure A.3 briefly in turn. The letters in the headings below refer to the labelled boxes in the figure.

A.5.1 Identifiers (a)

Everything in the development is dependent on the same set of identifiers. This makes it easy to convert named inputs and outputs on boxes into equivalent records within the embedded type system without writing tedious conversion and equivalence code. Given appropriate conversions, this dependency could be converted from a static parameter (baked into the coordination functor instantiation) to a dynamic one (as a dependently typed record).

A.5.2 OIDs (b)

Object identifiers are an axiomatization of ITU/ISO object identifiers, and are present so that the observation and manifestation semantics of the language can reference real objects in the world. OIDs are arcs of natural numbers, such as 1.2.3.4, where there is no restriction either on the maximum number in each component of the arc, or on the length of the arc. OIDs are administered hierarchically, in a similar way to the internet's domain name system (DNS). Unlike DNS, however, OIDs tend to be used for systems level designations. The numbers can be associated with text, and at present, in order to avoid the overhead of translating from text to numbers, we substitute text strings for numbers in our axiomatization of OIDs. A further advantage of OIDs is that an OID sub-structure, such as instances of HBCL logic, can be grafted onto any OID arc without changing the semantics. We simplify things further by introducing relative OIDs within HBCL programs as a syntactic and semantic convenience and as a method of removing semantic dependence on a root OID. A reference to a relative OID is automatically rooted at the closest scope at which the leftmost component of the OID reference matches the rightmost component of a library instance OID. The scope dependencies introduced by this mechanism form part of the *logical instance signature* of a *logical instance*. By adding the requirement that OIDs are allocated monotonically, that instances extend the OIDs of their type, and that temporal modality of a value is indexed by a further extension to an OID, we can postulate an extremely strong method of referring to system properties on spatial,[3] temporal and informational axes. The references that a particular instance makes to libraries outside its scope contribute to an implicit functor signature for that instance. The instance is only fully defined once these dependencies are discharged by being placed in a suitable context scope, forming an 'instance closure'.

The example type system that we provide for this version of HBCL is first order and consists of a Boolean base type and user-defined types that can be recursively defined as tuples or records.

A.5.3 The plain untimed type system (c)

The plain type system of HBCL is a deep embedding of an arbitrary intrinsic type system[4] on which the rest of the modules are parametrized. This enables the expression and coordination fragments to communicate with each other. The types are required to be be sized, and must use dependent typing (derived from predicates) to express the value domain of each type. In the same way as identifiers, this could be further generalized to a dynamic parameter with suitable conversion functions, equivalences and morphisms, but this is unnecessary for present purposes so we stay with a static module instantiation.

A.5.4 The OID type system functor (d)

The OID type system functor systematically lifts any deep embedding of a plain type system into an OID-qualified type system. It enables the programmer to enforce axiomatically disjoint domains between structurally identical types, according to *application level semantics*. These types are automatically qualified by the scope of the logical instances at which they are declared.

[3]On account of a physical object's position.

[4]An 'intrinsic' embedding of a type system is one in which each type of data in the embedded logic has a corresponding unique type in the embedded formalization. The term is popular in discussions of higher-order encodings of λ-calculi such as PHOAS [46], although we do not use it for that purpose here.

A.5.5 The harmonic type system functor (e)

The harmonic type system functor operates in the same way as the OID type system functor, except that it takes both plain and instantiated OID type systems as module parameters. It further refines untimed OID types by associating a sampling frequency with each one and assigning a further OID to the object thus described.

A.5.6 Untimed box language interface functor (f)

At this stage, we must distinguish between what we mean by an *untimed expression language* and an *untimed box language*. An untimed box language is provided with a set of input and output identifiers with untimed OID types. At this level of abstraction, it would be possible to embed an arbitrary coordination language or an expression language within the box language.[5] This language embedded in a box language may or may not be aware of OID semantics. If it is not, then explicit casting semantics must be provided in the embedding of the language to convert the types between OID and plain types (as annotated with cost functions). It is also possible to directly embed an expression language within the untimed box, and this is what is done in the toy expression language developed to demonstrate HBCL. Again, casting semantics are given in order to deal with the fact that the embedded language may not be aware of OID semantics (our toy expression language is not).[6]

The functor described here is thus a functor of type systems, *not* of box languages. The box languages themselves are formalized as a record containing its semantics, taking full advantage of using a higher order type system to specify HBCL. The type of this record is crystalized when the untimed box language interface functor is instantiated with suitable type systems.

A.5.6.1 Instantiaton of untimed box interface

The untimed box interface is instantiated with the toy expression language. It defines a number of built-in functions over primitive types, and the ability to define other functions through expressions that may include references to arguments and calls to any built-in or user-defined functions. The inclusion of built-in functions as a computational recursive base case has the advantage that we can use the host logic to implement them, meaning that we do not have to provide case analysis in the language to bootstrap some basic functionality. Providing case analysis would introduce considerably more complexity to the implementation, especially in making sure that only exhaustive

[5]Given that the coordination language would run for short bursts, we would need to add a mechanism to maintain a state variable between executions to permit this fully general case.

[6]A concept which could be extended to embed a box language with a completely different type system to the plain type system that is the common concurrency of an instantiation of HBCL.

match clauses were permitted. Given that this thesis is primarily about a coordination language, this additional complexity was not justified.

A.5.7 The harmonic box language interface functor (g)

This module is the glue between the coordination and untimed box languages.[7] It provides an input/output mechanism for the untimed box language and allows the coordination language to do meaningful computations with values, ensuring that semantics for down-casting from and up-casting to these timed types are fully specified and unambiguous. It enriches types with periodicity information and optionally defines the relative offset between when a value is notionally computed and the time to which it conceptually relates. It also fixes bounds on the minimum and maximum number of values that may be accepted by the harmonic box language at any one time, and how many it may produce. In the case of a type system and expression language with fixed-size types, such as the one used in this thesis, it enforces the frequencies of memory executions so that the number of values provided and produced in any one execution is fixed.

A.5.7.1 Instantiation of harmonic box language

This object is a simple timed binding for the toy untimed box language, changing a list of timed values for each input and output memory into a record of tuples of the correctly typed underlying values. It is at this point that the fact that the type system and the rest of HBCL share a common identifier representation becomes useful.

A.5.8 Concrete coordination language functor (h)

The coordination language assumes that all computation is reducible to a computable function of the host logic through the deep embedding of the harmonic box language. This is the sense in which full HBCL itself can be justified as a meta-language: it provides formal meta-semantics for these arbitrary harmonic box languages. This allows the coordination language to focus on specifying memories, FIFOs and their interconnection. It also provides the formal dynamic semantics of the model's concurrency in a form in which theorems may be proved. The coordination language is statically parametric on the formalization of the interfaces of the box languages, not on the languages themselves. This makes it possible to quantify simultaneously over arbitrary languages from within the host logic containing the formalization.

[7]In further work, the harmonic box language semantic object may be generalized to allow coordination languages that are aware of timed types to be axiomatized directly as harmonic box languages; in fact, nested HBCL instances can be regarded as a recursive case of this idea (see appendix A.5.8.4).

A.5.8.1 Periodicities

All frequencies in the model are expressed as rational numbers. This ensures that connected processes, memories and FIFOs fall in and out of phase with each other with constant periodicities. It is likely to be useful to users of HBCL to limit the frequencies used to the arithmetic product of the powers of a few low primes. By engineering an HBCL specification so that integer powers of low primes predominate as much as possible, these periodicities can be kept as short as possible. This should not be hard to arrange in practice.

A.5.8.2 Memories and FIFOs

There are four kinds of memory possible in the model, of which we implement two:

1. Memories that stand between the outputs of boxes and the inputs of FIFOs
2. Memories that stand between the outputs of FIFOs and the inputs of boxes
3. Memories that stand between boxes
4. Memories that stand between FIFOs

The formalism is in a sense a shared memory model, although the memories have less in common with physical memories, but are rather perdurant abstractions of points in space which have associated with them observable values. The memory is conceptually a list of data 'buckets' containing consecutive values on a timeline with discrete increments determined by the frequency of the timed type of the memory. Each memory also has a signed integer that gives the number of cycles of the memory's timed type to or from live. This determines the time that should exist between the first value to be readable after each memory execution and the value's past or future timestamp. There is no handshaking between the observed memory and the observer: coordination is achieved by timing alone. The values with these static relationships to the memory's time to or from live are the only values that need to be in the memory at the time it executes. However, because the potentially different frequencies of type, memory, FIFO and box may cause the number of values deposited and observed for each memory execution to differ, there are implicit minima and maxima on the lengths of the list. These minima and maxima ensure that old values are retained for long enough for them to be observed, and enough new values are available to prevent the observing process from becoming starved of data. The most elegant way of dealing with this problem is to include a data list for each FIFO of values that are 'in transit', to require all values that are available on any execution to be read immediately, and to add values in transit to the state space of the coordination language trace. However, to make things simpler for our implementation, we have collapsed this state space onto the memories themselves, and removed the frequencies and implicit transit capacities of FIFOs from the specification. This means there is not enough information to infer the lengths of the data lists in our memories. We therefore supply the missing information in the form of a minimum and a maximum length of memories as a required invariant, and FIFO semantics are partially subsumed into memory semantics in this way. FIFOs copy information directly from one memory to the other.

The frequency of the memory is not necessarily the same as the frequency of the timed type of the memory. As a consequence, if the frequency of the memory is lower, more than one new value and more than one old value will be made available and discarded on each memory tick respectively. If the frequency of the memory is higher, some cycles of the memory will not be observed to change, though they may have consumed values from their input buffer.

Under initial or fault conditions, memories may be empty. The handling of empty memories is delegated to the box binding for the particular harmonic box language. In the case of the toy expression language, output memories that fail to present the minimum number of output values are rejected and produce null-valued timed data. These nulls are a property of the timed data types, rather than the untimed OID types or plain types over which they are defined.

We do not implement memories that stand directly between boxes. However, if we did, they would have to have the same frequency as the boxes to which they were attached, and be expressed as zero-length FIFOs. This would be a neat means to prepend and append fan-out and voting logic to substantive processes. Similarly, memories that stood between FIFOs could be expressed as two memories either side of a null box. Zero-length FIFOs are not handled in the present implementation, since they would require a partial ordering on the execution of boxes in any given time slice, violating the global time-stepping of boxes, and requiring additional constraints checking for the absence of data flow cycles to prevent deadlock and livelock. This would add complexity, which would be distracting and unnecessary given that it would only become useful in the presence of automatic transformation of programs, and we have not yet implemented this. Where we introduce a voting and fan-out example in chapter 6, we do so with non-zero length FIFOs, and accept that the signatures of replicated and non-replicated programs will be temporally displaced by a few cycles to accommodate the extra communication.

A.5.8.3 Scheduling

The dynamic semantics of the coordination language employ a sparse time model on top of an underlying conception of dense time. Sparse time and dense time are both well known philosophies of time. Particular instances of the model have sparse time properties. That is, time elapses in increments demarcated by state transition events. Dense time is time thought of as a rational number. This is sufficient at the meta-model level to accommodate all specifiable harmonic frequencies.

The dynamic semantics entail four phases of execution, that proceed in an infinite round robin fashion. Only events whose time coincides precisely with the time of the pass through the round robin cycle are enabled on any given cycle. Each component of the four-fold cycle takes place at the same conceptual point in wall-clock time, but always in the same sequence. Alternatively, these events can be thought of as being separated by an infinitessimally small time slice. These phases are as follows:

- FIFOs execute.
- FIFO-box memories execute.
- Boxes excecute.
- Box-FIFO memories execute.

During the building of the static semantic object, events happening at the same point in dense time are consolidated into a single state transition, and are assigned to the correct one of these execution phases. The deadlock-free properties follow extremely easily from this model, since the state transitions must be reducible to Coq functions. For these to be expressible in Coq, they must be strongly normalizing. Therefore, the set of state transitions these dynamic semantics describe should be a coinductive list. We prefer not to embed this list in another temporal logic (such as the LTL[8] formalization in Coq due to Coupet-Grimal [57]), since this would add extra complexity which for the present objectives would not be useful: the concept of coinduction itself is adequately expressive.

A.5.8.4 Nesting

Where instances are nested, they acquire a frequency determined only by the signature of the nested instance, and the instance is only invoked with this frequency, as if it were an ordinary harmonic box. The nested case of HBCL is a recursive case of the idea that an arbitrary coordination language can be accommodated in a box language. Inside the box, several iterations of the four-fold stepping cycle may take place before the instance blocks on input or ouput operations. This finite set of cycles is realized as a fixpoint version of the cofixpoint that is required for executing the top level. The concatenation of each of these fixpoint slices produces the same trace as would have resulted had the

[8]Linear Temporal Logic

nested instance actually been at the top level, thus allowing us to recover the universal compositionality of the language.

A.5.9 Coordination language interpreter (i)

The coordination language interpreter supplies existential proof of the model's executability by giving an inhabitant in the type of interpreters.

A.5.10 Coordination language instantiation (j)

The coordination language instantiation module is merely the concrete version of the coordination module with the type system, identifier, OID and dependent module functors instantiated.

Appendix B

Semantic notation: a simple example

B.1 Notation: an illustrative example

We develop our structural operational semantics using the simple and familiar example of propositional calculus with a natural deduction [159] proof system, expressing the inference rules as semantic rules. It is not our purpose here to examine the logic of propositional calculus or natural deduction, so we have formalized the inference rules as presented in a primer text. We chose [18]. We will extend this to show how the same example is realized in Coq, according to our approach.

B.2 Abstract syntax of the propositional calculus

A slightly unusual syntax is adopted to describe the abstract syntax, involving named constructors. This choice is designed to ease translation into a meta-logical formalism when we come to express a deep embedding in a metalogical formalism or prove things about the language. It also functions as built-in documentation.

$$\begin{array}{lcl} wff & ::= & \texttt{WffStatement}\ \textit{asciiletter} \\ & | & \texttt{WffNot}\ wff \\ & | & \texttt{WffAnd}\ wff \quad wff \\ & | & \texttt{WffOr}\ wff \quad wff \\ & | & \texttt{WffIf}\ wff \quad wff \end{array} \tag{B.1}$$

Well-formed formulæ of the propositional calculus are built using the *wff* type.

$$\begin{array}{lcl} proof & ::= & \texttt{proofInd}\ wff \quad proof \\ & | & \texttt{proofBase} \end{array} \tag{B.2}$$

A syntactic *proof* is a list of well-formed formulæ. We give separate syntactic types for lists in this notation, rather than assume the existence of higher-order lists. These would presuppose a more complicated type system than the algebraic data types of most abstract syntax. The semantics of the natural deduction formalization qualify which syntactic '*proofs*' are real proofs of the logic.

B.3 Semantic domain of propositional calculus example

The main feature of the semantic domain of the propositional calculus example is its thirteen predicates axiomatizing natural deduction rules of inference. They show the typographical conventions described in section 4.4.1 being used. We also see that arguments are given to the meta-variables of predicates. This is because these arguments are bound to the dependent type of predicates under the constructors of well-formed formulæ. For example, the *modus tollens* rule has a dependent type that is bound to be the negation of the argument that was supplied to its meta-variable. The thirteen separate predicates are bound by mutual induction with an umbrella predicate, which occurs last in the table. It is the disjoint union of all the others. The type definitions use standard logical connectives.

Meta-variable or constant	**Type name**	**Type definition**
$\mathsf{ND\_ID}_{\mathsf{Prop}}\big(w : wff\big)$ (Natural deduction identity rule)	$\kappa_{\mathsf{ND\_ID}} \leftarrow w \leftarrow \big(\forall p : proof\big)$	$\mathsf{In}_{\mathsf{Prop}}\big(p, w\big)$
$\mathsf{ND\_ModusPonens}_{\mathsf{Prop}}\big(w : wff\big)$ (Modus ponens rule)	$\kappa_{\mathsf{ND\_ModusPonens}} \leftarrow w \leftarrow \big(\forall p : proof\big)$	$\exists w' : wff,\ \mathsf{ND}_{\mathsf{Prop}}\big(p, \mathtt{WffIf}(w', w)\big) \land \mathsf{ND}_{\mathsf{Prop}}\big(p, w'\big)$
$\mathsf{ND\_ModusTollens}_{\mathsf{Prop}}\big(w : wff\big)$ (Modus tollens rule)	$\kappa_{\mathsf{ND\_ModusTollens}} \leftarrow (\mathtt{WffNot}(w)) \leftarrow \big(\forall p : proof\big)$	$\exists w' : wff,\ \mathsf{ND}_{\mathsf{Prop}}\big(p, \mathtt{WffIf}(w, w')\big) \land \mathsf{ND}_{\mathsf{Prop}}\big(p, \mathtt{WffNot}(w')\big)$
$\mathsf{ND\_SimplA}_{\mathsf{Prop}}\big(w : wff\big)$ (Left-handed conjunction simplification rule)	$\kappa_{\mathsf{ND\_SimplA}} \leftarrow w \leftarrow \big(\forall p : proof\big)$	$\exists w' : wff,\ \mathsf{ND}_{\mathsf{Prop}}\big(p, \mathtt{WffAnd}(w, w')\big)$
$\mathsf{ND\_SimplB}_{\mathsf{Prop}}\big(w : wff\big)$ (Right-handed conjunction simplification rule)	$\kappa_{\mathsf{ND\_SimplB}} \leftarrow w \leftarrow \big(\forall p : proof\big)$	$\exists w' : wff,\ \mathsf{ND}_{\mathsf{Prop}}\big(p, \mathtt{WffAnd}(w', w)\big)$
$\mathsf{ND\_Conj}_{\mathsf{Prop}}\begin{pmatrix} w : wff \\ w' : wff \end{pmatrix}$ (Conjunction rule)	$\kappa_{\mathsf{ND\_Conj}} \leftarrow \mathtt{WffAnd}(w, w') \leftarrow \big(\forall p : proof\big)$	$\mathsf{ND}_{\mathsf{Prop}}\big(p, w\big) \land \mathsf{ND}_{\mathsf{Prop}}\big(p, w'\big)$

Meta-variable or constant	**Type name**	**Type definition**
$\mathsf{ND\_DisjA}_{\mathsf{Prop}}\begin{pmatrix} w : wff \\ w' : wff \end{pmatrix}$ (Left-handed disjunction rule)	$\kappa_{\mathsf{ND\_DisjA}} \leftarrow$ $\texttt{WffOr}(w, w') \leftarrow$ $\big(\forall p : proof\big)$	$\mathsf{ND}_{\mathsf{Prop}}\big(p, w\big)$
$\mathsf{ND\_DisjB}_{\mathsf{Prop}}\begin{pmatrix} w : wff \\ w' : wff \end{pmatrix}$ (Right-handed disjunction rule)	$\kappa_{\mathsf{ND\_DisjB}} \leftarrow$ $\texttt{WffOr}(w, w') \leftarrow$ $\big(\forall p : proof\big)$	$\mathsf{ND}_{\mathsf{Prop}}\big(p, w'\big)$
$\mathsf{ND\_ConjSyllogA}_{\mathsf{Prop}}\big(w : wff\big)$ (Left-handed conjunctive syllogism rule)	$\kappa_{\mathsf{ND\_ConjSyllogA}} \leftarrow$ $\texttt{WffNot}(w) \leftarrow$ $\big(\forall p : proof\big)$	$\exists w' : wff,$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffNot}(\texttt{WffAnd}(w, w'))\big)$ $\wedge\ \mathsf{ND}_{\mathsf{Prop}}\big(p, w'\big)$
$\mathsf{ND\_ConjSyllogB}_{\mathsf{Prop}}\big(w : wff\big)$ (Right-handed conjunctive syllogism rule)	$\kappa_{\mathsf{ND\_ConjSyllogB}} \leftarrow$ $\texttt{WffNot}(w) \leftarrow$ $\big(\forall p : proof\big)$	$\exists w' : wff,$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffNot}(\texttt{WffAnd}(w', w))\big)$ $\wedge\ \mathsf{ND}_{\mathsf{Prop}}\big(p, w'\big)$
$\mathsf{ND\_DisjSyllogA}_{\mathsf{Prop}}\big(w : wff\big)$ (Left-handed disjunctive syllogism rule)	$\kappa_{\mathsf{ND\_DisjSyllogA}} \leftarrow$ $w \leftarrow$ $\big(\forall p : proof\big)$	$\exists w' : wff,$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffOr}(w, w')\big)$ $\wedge\ \mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffNot}, (w')\big)$
$\mathsf{ND\_DisjSyllogB}_{\mathsf{Prop}}\big(w : wff\big)$ (Right-handed disjunctive syllogism rule)	$\kappa_{\mathsf{ND\_DisjSyllogB}} \leftarrow$ $w \leftarrow$ $\big(\forall p : proof\big)$	$\exists w' : wff,$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffOr}(w', w)\big)$ $\wedge\ \mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffNot}, (w')\big)$
$\mathsf{ND\_HypSyllog}_{\mathsf{Prop}}\begin{pmatrix} w : wff \\ w' : wff \end{pmatrix}$ (Hypothetical syllogism rule)	$\kappa_{\mathsf{ND\_HypSyllog}} \leftarrow$ $\texttt{WffIf}(w, w') \leftarrow$ $\big(\forall p : proof\big)$	$\exists w'' : wff,$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffIf}(w, w'')\big)$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffIf}(w'', w')\big)$
$\mathsf{ND\_Dilemma}_{\mathsf{Prop}}\big(w : wff\big)$ (Dilemma rule)	$\kappa_{\mathsf{ND\_Dilemma}} \leftarrow$ $w \leftarrow$ $\big(\forall p : proof\big)$	$\exists w' : wff,\ \exists w'' : wff,$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffOr}(w', w'')\big)$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffIf}(w', w)\big)$ $\mathsf{ND}_{\mathsf{Prop}}\big(p, \texttt{WffIf}(w'', w)\big)$

Meta-variable or constant	**Type name**	**Type definition**
$\mathsf{ND}_{\mathsf{Prop}}\big(w : wff\big)$ (Natural deduction disjoint product of inference rules)	$\kappa_{\mathsf{ND}} \leftarrow w \leftarrow \big(\forall p : proof\big)$	$\coprod \left\vert \begin{array}{l} \mathsf{ND\_ID}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_ModusPonens}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_ModusTollens}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_SimplA}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_SimplB}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_Conj}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_DisjA}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_DisjB}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_ConjSyllogA}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_ConjSyllogB}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_DisjSyllogA}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_DisjSyllogB}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_HypSyllog}_{\mathsf{Prop}}\big(p, w\big) \\ \mathsf{ND\_Dilemma}_{\mathsf{Prop}}\big(p, w\big) \end{array}\right.$

The table of predicates for valid arguments does not introduce new notation, but shows definitions which ensure that ‘*proof*s’ qualified by the valid argument predicate are correct by construction. We define a list concatenation function as a convenience, whose effect is as its name suggests. The empty predicate is defined to qualify an empty argument, using the ‘empty set’ symbol ∅.

Meta-variable or constant	**Type name**	**Type definition**
$\mathsf{ND\_ValArg\_empty}_{\mathsf{Prop}}$	$\kappa_{\mathsf{ND\_ValArg\_empty}} \leftarrow \mathtt{proofBase} \leftarrow \big(\forall p : proof\big)$	$\varnothing$
$\mathsf{ND\_ValArg\_cons}_{\mathsf{Prop}}\begin{pmatrix} w : wff \\ p' : proofs \end{pmatrix}$	$\kappa_{\mathsf{ND\_ValArg\_cons}} \leftarrow \mathtt{proofInd}\big(w, p'\big) \leftarrow \big(\forall p : proof\big)$	$\mathsf{ND}\big(\mathsf{WffConcat}\big(p, p'\big), w\big)$ $\mathsf{ND}_{\mathsf{Prop}}(w)$ $\mathsf{ND\_ValidArgument}_{\mathsf{Prop}}\big(p, p'\big)$
$\mathsf{ND\_ValidArgument}_{\mathsf{Prop}}\big(p' : proofs\big)$	$\kappa_{\mathsf{ND\_ValidArgument}} \leftarrow p' \leftarrow \big(\forall p : proof\big)$	$\coprod \left\vert \begin{array}{l} \mathsf{ND\_ValArg\_empty}_{\mathsf{Prop}}\big(p, p'\big) \\ \mathsf{ND\_ValArg\_cons}_{\mathsf{Prop}}\big(p, p'\big) \end{array}\right.$

Having defined some predicates, we now show how dependent types, restricted to be dependent in the arguments of parametrized σ-types, are defined.

Meta-variable or constant	**Type name**	**Type definition**
$W_{\top}$	$\mathscr{W}_{\top} \leftarrow \big(\forall p : proof\big)$	$\left\{ \begin{array}{l} w \in wff : \\ \mathsf{ND}_{\mathsf{Prop}}\big(p, w\big) \end{array} \right\}$
$\mathtt{W}_{\bot}$	$\mathscr{W}_{\bot}$	$\varnothing$
$\mathbf{W}_{\top\bot}$	$\mathfrak{W}_{\top\bot}$	$\mathscr{W}_{\top} \bigsqcup \mathscr{W}_{\bot}$

In the first row of the table for the strong definition of a well-formed formula, we see that it has acquired a dependence in the type of syntactic proofs. The definition is of a σ-type, parametrized in the environment of premises. These are the premises that form the enviroment in which the formula is contingently true (given the truth and mutual consistency of the premises) if a strong well-formed formula W can be constructed. The second row gives the object that denotes an inconsistent formula, and the final row defines the disjoint union of consistent and inconsistent formulæ. These objects are needed so that the operational semantics can pass around a definite object denoting a failed attempt to check a proof script.

Meta-variable or constant	**Type name**	**Type definition**
$P_{\top}$, $\mathtt{P}_{\top\varnothing}$	$\mathscr{P}_{\top} \leftarrow \big(\forall p : proof\big)$	$\left\{ \begin{array}{l} p' \in wff : \\ \mathsf{ND\_ValidArgument}_{\mathsf{Prop}}\big(p, p'\big) \end{array} \right\}$
$\mathtt{P}_{\bot}$	$\mathscr{P}_{\bot}$	$\varnothing$
$\mathbf{P}_{\top\bot}$	$\mathfrak{P}_{\top\bot}$	$\mathscr{P}_{\top} \bigsqcup \mathscr{P}_{\bot}$

The definition of a valid argument from its allied predicate follows the same pattern as the definition of a well-formed formula above. A constant denoting the consistent empty argument is also present in this definition, subscripted by the empty set sign.

B.4 Structural operational semantics: notation

We now introduce the type of operational structural semantic notation that we will be using. In the case of our example proof-checker, the static semantics are the same as the dynamic semantics: the static semantics of a valid proof are the dynamic semantics of a proof-checker. For programs that do something other than check or compute with static properties, this is not true, and so HBCL has separate static and dynamic semantics. Nevertheless, static and dynamic semantics have the same basic form, and so our example serves to illustrate the constructs we will come across in the HBCL dynamic semantics.

In the semantic rules that follow, we refer to the terms on the left of the turnstile as the ‘environment’, to the terms between the turnstile and the arrow as the ‘implicant’, and to the terms on the right of the arrow as the ‘implicand’.

The first four rules show the possible matches on a proof script.

$$\frac{\left\{ \vdash \left\{ \ e_{proof},\ p_{proof} \ \right\} \Rightarrow \left\{ \mathrm{P}_{\bot} \right\} \right\}}{\left\{ \vdash \left\{ \ e_{proof},\ \texttt{proofInd}\left(w_{wff},\ p_{proof} \right) \ \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\ \mathrm{P}_{\bot} \ \right) \right\} \right\}} \tag{B.3}$$

Rule B.3 shows the match that results in an empty and inconsistent object being returned in the implicand of the conclusion. All the preceding elements of the proof script are invoked by the premise, and if the result is inconsistent, then this particular invocation of the rule can only be inconsistent too: there is no point in investigating whether the well-formed formula that is matched in the implicant of the conclusion is valid. We observe at this stage that the enviroment of premises is *after* the turnstile of the semantic rule. This is different to the usual operational semantic style, but since we will be matching directly on these environment objects, we reserve the environment of semantic rules for variables that only constitute arguments to local *predicates*, and which are not computationally relevant to concrete (non-predicate) types.

$$\frac{\begin{array}{c} \left\{ \vdash \left\{ \ e_{proof},\ p_{proof} \ \right\} \Rightarrow \left\{ \frac{\mathrm{P}_{\top}}{e_{proof}}\left(\ \sigma\left(\begin{array}{l} p'_{proof} \\ \mathsf{ND\_ValidArgument}_{\mathsf{Prop}}\left(\ e_{proof},\ p'_{proof} \ \right) \end{array} \right) \right) \right\} \right\} \\ \left\{ \vdash \left\{ \ \mathsf{WffConcat}\left(\ e_{proof},\ p'_{proof} \ \right),\ w_{wff} \ \right\} \Rightarrow \left\{ \mathrm{W}_{\bot} \right\} \right\} \end{array}}{\left\{ \vdash \left\{ \ e_{proof},\ \texttt{proofInd}\left(w_{wff},\ p_{proof} \right) \ \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\ \mathrm{P}_{\bot} \ \right) \right\} \right\}} \tag{B.4}$$

Rule B.4 shows the situation where the preceeding proof matched in the conclusion's implicant, e_{proof}, yields a consistent proof, but where the local well-formed formula, matched as w_{wff} cannot be proven given the available premises. In the implicand of the first premise, we see that the underlying σ-type has been matched, to produce the syntactic proof object which we now know is really a valid proof. This new proof fragment e_{proof}', is used in the second premise, concatenated with the common (logical) premise e_{proof}, to invoke a match on one of the rules for proving a formula. However, the inconsistent formula is found in the implicand, which means the implicand of the conclusion must be that the whole proof script is not proven. The rule therefore returns the inconsistent proof object, wrapped in the disjoint union constructor for consistent and inconsistent proof objects.

$$\frac{\begin{array}{c}\left\{ \vdash \left\{ e_{proof},\, p_{proof} \right\} \Rightarrow \left\{ \frac{\mathrm{P}_{\top}}{e_{proof}} \left(\sigma \left(\begin{array}{l} p'_{proof} \\ \mathsf{ND\_ValidArgument}_{\mathsf{Prop}}\left(e_{proof},\, p'_{proof} \right) \end{array} \right) \right) \right\} \right\} \\ \left\{ \vdash \left\{ \mathsf{WffConcat}\left(e_{proof},\, p'_{proof} \right),\, w_{wff} \right\} \Rightarrow \left\{ \frac{\mathrm{W}_{\top}}{\mathsf{WffConcat}\left(e_{proof},\, p'_{proof} \right)} \right\} \right\}\end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \texttt{proofInd}\left(w_{wff},\, p_{proof} \right) \right\} \\ \Rightarrow \left\{ \mathfrak{P}_{\top\bot} \left(\frac{\mathscr{P}_{\top}}{e_{proof}} \left(\sigma \left(\begin{array}{l} \texttt{proofInd}\left(w_{wff},\, p'_{proof} \right) \\ \mathsf{ND\_ValidArgument}_{\mathsf{Prop}} \left(\begin{array}{l} e_{proof} \\ \mathsf{WffConcat}\left(e_{proof},\, p'_{proof} \right) \end{array} \right) \end{array} \right) \right) \right) \right\} \end{array} \right\}} \tag{B.5}$$

The form of rule B.5 is similar to that of rule B.4, except that in the second premise, there is now a successful match on a proven formula in the environment. It is dependent in the concatenated proof script that was supplied in the implicant of this premise. This new and predicate qualified object is used in the implicand of the conclusion to construct a new strong proof object, elevating the syntactic proof (a list of formulæ) to the type of correct proofs. Both the concrete terms (under the proofInd constructor) and new predicate are shown. We do not show the operational rules for constructing the predicate. This is an obligation that is discharged in the environment of a full-fledged proof assistant when we show the equivalent of this rule in appendix B.8.

$$\frac{}{\left\{ \vdash \left\{ e_{proof},\, \texttt{proofBase()} \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot} \left(\,,\, \frac{\mathrm{P}_{\top\varnothing}}{e_{proof}} \right) \right\} \right\}} \tag{B.6}$$

Rule B.6 shows the trivial base case in which the empty correct proof object is constructed from the empty syntactic proof object.

We now show each operational rule for building a proof based on one of the natural deduction predicates in the semantic domain. We do not meet any new features of the notation here, so the point of interest is the comparison of the structure with that of the allied predicate in the semantic domain, and with the equivalent Coq function in appendix B.8.

$$\frac{\begin{array}{c}\left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{\mathrm{W}_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \texttt{WffIf}\left(w'_{wff},\, w_{wff} \right) \right\} \Rightarrow \left\{ \frac{\mathrm{W}'_{\top}}{e_{proof}} \right\} \right\}\end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, w_{wff} \right\} \\ \Rightarrow \left\{ \mathfrak{W}_{\top\bot} \left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} w_{wff} \\ \mathsf{ND\_ModusPonens}\left(e_{proof},\, w_{wff} \right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.7}$$

Rule B.7 implements the *modus ponens* rule of inference. The conclusion implicand matches on the environment of already accepted propositions and the syntactic formula object, and shows two premises corresponding to the logical premises of the rule. A new variable w' is shown in the environment of the premises: the existence of such a variable is a prerequisite for the application of the rule. We now enumerate the remainder of the rules, which follow the same structural pattern.

$$\frac{\begin{array}{c} \left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, \mathtt{WffNot}\left(w'_{wff}\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \mathtt{WffIf}\left(w_{wff},\, w'_{wff}\right) \right\} \Rightarrow \left\{ \frac{W'_{\top}}{e_{proof}} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \mathtt{WffNot}\left(w_{wff}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{WB}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \mathtt{WffNot}\left(w_{wff}\right) \\ \mathsf{ND\_ModusTollens}\left(e_{proof},\, \mathtt{WffNot}\left(w_{wff}\right) \right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.8}$$

$$\frac{\left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \mathtt{WffAnd}\left(w_{wff},\, w'_{wff}\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, w_{wff} \right\} \\ \Rightarrow \left\{ \mathfrak{WB}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} w_{wff} \\ \mathsf{ND\_SimplA}\left(w_{wff} \right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.9}$$

$$\frac{\left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \mathtt{WffAnd}\left(w'_{wff},\, w_{wff}\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, w_{wff} \right\} \\ \Rightarrow \left\{ \mathfrak{WB}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} w_{wff} \\ \mathsf{ND\_SimplB}\left(w_{wff} \right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.10}$$

$$\frac{\begin{array}{c} \left\{ w_{wff} \;\vdash \left\{ e_{proof},\, w_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{W'_{\top}}{e_{proof}} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \mathtt{WffAnd}\left(w_{wff},\, w'_{wff}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{WB}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \mathtt{WffAnd}\left(w_{wff},\, w'_{wff}\right) \\ \mathsf{ND\_Conj}\left(w_{wff} \right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.11}$$

$$\frac{\left\{ w_{wff} \;\vdash \left\{ e_{proof},\, w_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \mathtt{WffOr}\left(w_{wff},\, w'_{wff}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{WB}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \mathtt{WffOr}\left(w_{wff},\, w'_{wff}\right) \\ \mathsf{ND\_DisjA}\left(w_{wff} \right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.12}$$

$$\frac{\left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \texttt{WffOr}\left(w_{wff},\, w'_{wff}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{W}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \texttt{WffOr}\left(w_{wff},\, w'_{wff}\right) \\ \textsf{ND\_DisjB}\left(\, w_{wff} \,\right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.13}$$

$$\frac{\begin{array}{l} \left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \texttt{WffNot}\left(\texttt{WffAnd}\left(w_{wff},\, w'_{wff}\right)\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \texttt{WffNot}\left(w_{wff}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{W}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \texttt{WffNot}\left(w_{wff}\right) \\ \textsf{ND\_ConjSyllogA}\left(\, e_{proof},\, \texttt{WffNot}\left(w_{wff}\right) \,\right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.14}$$

$$\frac{\begin{array}{l} \left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \texttt{WffNot}\left(\texttt{WffAnd}\left(w'_{wff},\, w_{wff}\right)\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, \texttt{WffNot}\left(w_{wff}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{W}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \texttt{WffNot}\left(w_{wff}\right) \\ \textsf{ND\_ConjSyllogB}\left(\, e_{proof},\, \texttt{WffNot}\left(w_{wff}\right) \,\right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.15}$$

$$\frac{\begin{array}{l} \left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \texttt{WffOr}\left(w_{wff},\, w'_{wff}\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, w_{wff} \right\} \\ \Rightarrow \left\{ \mathfrak{W}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \texttt{WffNot}\left(w_{wff}\right) \\ \textsf{ND\_DisjSyllogA}\left(\, e_{proof},\, \texttt{WffNot}\left(w_{wff}\right) \,\right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.16}$$

$$\frac{\begin{array}{l} \left\{ w_{wff},\, w'_{wff} \;\vdash \left\{ e_{proof},\, \texttt{WffOr}\left(w'_{wff},\, w_{wff}\right) \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \\ \left\{ w'_{wff} \;\vdash \left\{ e_{proof},\, w'_{wff} \right\} \Rightarrow \left\{ \frac{W_{\top}}{e_{proof}} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} \vdash \left\{ e_{proof},\, w_{wff} \right\} \\ \Rightarrow \left\{ \mathfrak{W}_{\top\bot}\left(\frac{\mathscr{W}_{\top}}{e_{proof}} \left(\begin{array}{l} \texttt{WffNot}\left(w_{wff}\right) \\ \textsf{ND\_DisjSyllogB}\left(\, e_{proof},\, \texttt{WffNot}\left(w_{wff}\right) \,\right) \end{array} \right) \right) \right\} \end{array} \right\}} \tag{B.17}$$

$$\frac{\begin{array}{l}\left\{\ w_{wff},\ w''_{wff}\ \vdash \left\{\ e_{proof},\ \texttt{WffIf}\left(w_{wff},\ w''_{wff}\right)\ \right\} \Rightarrow \left\{\frac{W_{\top}}{e_{proof}}\right\}\right\} \\ \left\{\ w''_{wff},\ w'_{wff}\ \vdash \left\{\ e_{proof},\ \texttt{WffIf}\left(w''_{wff},\ w'_{wff}\right)\ \right\} \Rightarrow \left\{\frac{W'_{\top}}{e_{proof}}\right\}\right\}\end{array}}{\left\{\begin{array}{l}\vdash \left\{\ e_{proof},\ \texttt{WffIf}\left(w_{wff},\ w'_{wff}\right)\ \right\} \\ \Rightarrow \left\{\mathfrak{W}_{\top\bot}\left(\ \frac{\mathscr{W}_{\top}}{e_{proof}}\left(\begin{array}{l} w_{wff} \\ \mathsf{ND\_HypSyllog}\left(\ e_{proof},\ w_{wff}\ \right)\end{array}\right)\right)\right\}\end{array}\right\}} \tag{B.18}$$

$$\frac{\begin{array}{l}\left\{\ w'_{wff},\ w''_{wff}\ \vdash \left\{\ e_{proof},\ \texttt{WffOr}\left(w'_{wff},\ w''_{wff}\right)\ \right\} \Rightarrow \left\{\frac{W_{\top}}{e_{proof}}\right\}\right\} \\ \left\{\ w'_{wff},\ w_{wff}\ \vdash \left\{\ e_{proof},\ \texttt{WffIf}\left(w'_{wff},\ w_{wff}\right)\ \right\} \Rightarrow \left\{\frac{W'_{\top}}{e_{proof}}\right\}\right\} \\ \left\{\ w''_{wff},\ w_{wff}\ \vdash \left\{\ e_{proof},\ \texttt{WffIf}\left(w''_{wff},\ w_{wff}\right)\ \right\} \Rightarrow \left\{\frac{W''_{\top}}{e_{proof}}\right\}\right\}\end{array}}{\left\{\begin{array}{l}\vdash \left\{\ e_{proof},\ w_{wff}\ \right\} \\ \Rightarrow \left\{\mathfrak{W}_{\top\bot}\left(\ \frac{\mathscr{W}_{\top}}{e_{proof}}\left(\begin{array}{l} w_{wff} \\ \mathsf{ND\_Dilemma}\left(\ e_{proof},\ w_{wff}\ \right)\end{array}\right)\right)\right\}\end{array}\right\}} \tag{B.19}$$

B.5 The embedding of semantics in Coq

We now carry the propositional calculus example forward to an embedding of the same semantics in Coq, in order to demonstrate, in the context of a simple example, how we have approached the translation of HBCL into a formal logic.

Since Coq uses a general-purpose logic, it can be employed to reason about semantics in any style. We use two styles: a predicative, type-theoretical style to define a strongly typed semantic domain, and an operational semantic style, in the form of Coq functions. The latter provides concrete procedures to find inhabitants of the former.

In the listings that follow, we have used the 'coqdoc' tool to format listings for presentation. 'coqdoc' is part of the Coq distribution. The tool adds syntax highlighting and some notational niceties such as replacing 'ASCII' representations of quantifiers with their mathematical counterparts. It also provides facilities to hide uninteresting parts of code files, such as proof scripts. The software produces many different formats and types of documentation, but we use the LaTeX output.

B.6 Propositional calculus abstract syntax in Coq

The abstract syntax of our propositional logic example is composed of an inductive type for well-formed formulæ, in combination with the `ascii` and `list` types from the Coq Standard Library.

Listing B.1: The propositional calculus abstract syntax

```
Definition Statement := ascii.
Inductive Wff : Set :=
| Wff_St : Statement → Wff
| Wff_Not : Wff → Wff
| Wff_And : Wff → Wff → Wff
| Wff_Or : Wff → Wff → Wff
| Wff_If : Wff → Wff → Wff.
```

We define `Statement` to be the Coq type `ascii`. A well-formed formula `Wff` is, to take the constructors as they appear, either a statement, a negation, a conjunction, a disjunction or an implication. The concrete type of syntactic proofs is not shown, as it is just an instance of Coq's polymorphic list type, `list Wff`.

B.7 Propositional calculus semantic domain in Coq

The semantic domain is composed of a number of predicates over the concrete syntactic types. We illustrate how these are turned into parametrized σ-types to form a strong semantic domain which, by construction, cannot contain meaningless subsets of the concrete objects.

Listing B.2: The propositional calculus semantic domain

```
Inductive NatDeduct(env : list Wff) : Wff → Prop :=
| ND_ID (A : Wff) : In A env → NatDeduct env A
| ND_ModusPonens(B : Wff) :
  (∃ A, NatDeduct env (Wff_If A B) ∧ NatDeduct env A) → NatDeduct env B
| ND_ModusTollens(A : Wff) :
  (∃ B, NatDeduct env (Wff_If A B) ∧ NatDeduct env (Wff_Not B)) →
  NatDeduct env (Wff_Not A)
| ND_SimplA(A : Wff) :
  (∃ B, NatDeduct env (Wff_And A B)) → NatDeduct env A
| ND_SimplB(B : Wff) :
  (∃ A, NatDeduct env (Wff_And A B)) → NatDeduct env B
| ND_Conj(A B : Wff) : NatDeduct env A ∧ NatDeduct env B →
  NatDeduct env (Wff_And A B)
| ND_DisjA(A B : Wff) : NatDeduct env A → NatDeduct env (Wff_Or A B)
| ND_DisjB(A B : Wff) : NatDeduct env B → NatDeduct env (Wff_Or A B)
| ND_ConjSyllogA(A : Wff) :
  (∃ B, NatDeduct env (Wff_Not (Wff_And A B)) ∧ NatDeduct env B) →
  NatDeduct env (Wff_Not A)
| ND_ConjSyllogB(B : Wff) :
  (∃ A, NatDeduct env (Wff_Not (Wff_And A B)) ∧ NatDeduct env A) →
  NatDeduct env (Wff_Not B)
| ND_DisjSyllogA(A : Wff) :
  (∃ B, NatDeduct env (Wff_Or A B) ∧ NatDeduct env (Wff_Not B)) →
  NatDeduct env A
| ND_DisjSyllogB(B : Wff) :
  (∃ A, NatDeduct env (Wff_Or A B) ∧ NatDeduct env (Wff_Not A)) →
  NatDeduct env B
| ND_HypSyllog(A C : Wff) :
  (∃ B, NatDeduct env (Wff_If A B) ∧ NatDeduct env (Wff_If B C)) →
  NatDeduct env (Wff_If A C)
| ND_Dilemma(C : Wff) :
  (∃ A, ∃ B, NatDeduct env (Wff_Or A B) ∧
     NatDeduct env (Wff_If A C) ∧ NatDeduct env (Wff_If B C)) →
  NatDeduct env C.
Inductive ND_ValidArgument(premises : list Wff) : list Wff → Prop :=
| ND_ValArg_empty : ND_ValidArgument premises nil
| ND_ValArg_cons(wff : Wff)(env : list Wff) :
  ND_ValidArgument premises env →
  NatDeduct (premises ++ env) wff → ND_ValidArgument premises (wff :: env).
```

The definition of the inductive type NatDeduct shows a predicate over well formed formulæ. It can be seen in the first line of the definition that it is declared in the Sort of propositions, Prop, whereas the concrete Wff type was declared in the concrete Set Sort. NatDeduct has two dependent arguments. The first, env, is a list of well-formed formulæ. The consistency of NatDeduct is dependent on the truth and mutual consistency of the formulæ in this list. This is a named parameter, which in Coq has the effect that nothing in a constructor can contribute to the named type parameter. It is intuitively correct that this should be a named type, because premises are by definition prior to the specification of any well-formed formula that can be constructed in this particular instance of NatDeduct. The type parameters are Curried like any other parameters, so once env is supplied, the new definition is tied to this set of premises. The second dependent parameter of NatDeduct is a well-formed formula that is supplied by each individual constructor. The form of this constructor varies depending on the inference rule that is being relied upon to produce constructive proof of the formula's validity in this environment.

This can be illustrated by the first constructor, ND_ID, which is an identity constructor. Its parameters are a concrete Wff, A, and a proof that A is contained in env. In is a predicate that we imported from Coq's List Module from the Standard Library. This allows us to use lemmas concerning Coq lists that are supplied by the Standard Library. The result of this constructor, NatDeduct env A, shows the instantiation of NatDeduct with its obligatory env parameter, and A as the concrete type over which this predicate can be constructed, which in this case is A itself, since this is an identity inference rule.

The rest of the constructors behave similarly, following the predicates that we defined in appendix B.3. The *modus ponens* constructor shows an example where Coq's polymorphic existential inductive predicate, shown by the '∃' symbol, is used to require that there is some A that is true in the environment, which together with a proof that A implies B, allows us to construct a predicate certifying that B is justified. In the *modus tolles* constructor, we can see an example of an inference rule that constrains the Wff over which the NatDeduct predicate is defined to be a synthesis of the constructor parameters, in this case providing a proof of the negation of the A, giving Wff_Not A. The rest of the constructor definitions of NatDeduct do not introduce any new Coq concepts.

The predicate ND_ValidArgument provides a way of specifying lists of inferences where every inference is justified by the previous instances added to the list. It has a parameter, called premises. ND_ValidArgument proves a valid argument in the logic, given the premises. It is itself a syllogism that does not comment upon the truth or mutual consistency of the premises. The value of the second type parameter is determined under the individual constructors of ND_ValidArgument. Each element of ND_ValidArgument is built up in tandem with the underlying list, ensuring that the underlying list is correct by construction. The ND_ValArg_empty constructor is the recursive base case, axiomatizing the consistency of the empty argument. ND_ValArg_cons mirrors the constructor of the underlying list, taking a well-formed fomula wff, the preceding proof script, env, proof that this environment is itself a ND_ValidArgument, and a proof that the wff is justified in the environment of the premises and env. The concatenation of these lists is shown by the ++ notation. The :: notation in the construction of the resulting ND_ValidArgument shows that this proof is a proof of the old list env with wff added to the head of the list.

Listing B.3: The propositional calculus consistency assertions

```
Inductive ND_PremisesConsistent : list Wff → Prop :=
| ND_PConsist_intro_nil : ND_PremisesConsistent nil
| ND_PConsist_intro_cons(wff : Wff)(pc : list Wff) :
   ¬ (ND_ValidArgument nil ((Wff_Not wff) :: pc)) →
   ND_PremisesConsistent pc → ND_PremisesConsistent (wff :: pc).
Definition ND_IsTrue(premises : sig (ND_PremisesConsistent))(wff : Wff) :
   Prop :=
   In wff (`premises) ∨
   ∃ script : sig (ND_ValidArgument (`premises)), NatDeduct (`script) wff.
Theorem ND_consistent(premises : sig (ND_PremisesConsistent)) :
   ∀ wff, ¬ (ND_IsTrue premises wff ∧ ND_IsTrue premises (Wff_Not wff)).
```

For completeness, Listing B.3 shows a predicate that constrains premises to be consistent, by building up the list of premises from an empty list (denoted by the nil constructor). ND_PConsist_intro_cons shows that a new formula can be added only if there is no valid argument from which can be deduced its negation. To negate a proposition in the intuitionistic logic of Coq is equivalent to saying that it does not exist, which is the same as saying it cannot be constructed. The definition ND_IsTrue and theorem ND_consistent, which we admit without proof, formalizes the claim that the logic is consistent. The sig type of Coq is the formalization of a σ-type, turning a predicate into a union of the underlying concrete value over which the predicate is declared and an instance of the predicate that shows that the concrete instance is 'true' according to the predicate. The backtick notation in Coq is used to denote the concrete part of a sig type, and we see that the predicate is stripped from the argument of ND_IsTrue, so that the predicates referenced in the type definition receive an argument of the correct type. The σ-type we use for the strong (*i.e.* correct by construction) definition of the type of proofs in the logic is formed from the ND_Valid_Argument, and we see that the predicate has two arguments. However, the description of a σ-type that we just gave is a monadic predicate, or one that has only one argument. The extra argument, the 'premises' of the ND_Valid_Argument definition, is bound as a parameter to the σ-type.

B.8 Structural operational semantics of the propositional calculus example in Coq

The functions given here realize a proof-checker. We use Coq's `option` type to wrap the object produced by this function. This corresponds to the union object in fraktur font in the semantic domain of appendix B.3. The `Some` constructor of the `option` type wraps a verified proof script, which has been enriched with a predicate that certifies this. The `None` constructor, which takes no arguments, is returned when the script being processed does not meet the semantic specification, corresponding to the branches in our semantic rules where the 'inconsistent object' is returned. There is a strong correspondence between the semantic rules and the realizing functions, but the executability of the Coq functions necessitates that they deal with details of searching lists for terms, rendering them less succinct.

We have proven that our proof-cheking function only produces inhabitants of the predicate-qualified semantic domain. We have not proved that it always produces such an inhabitant when one exists for the supplied un-predicated proof script. Neither have we proven consistency or completeness of the logical system we have formalized. These properties could be shown with more work, but the business of this appendix is to introduce our method, rather than to prove things about propositional calculus. Accordingly, we have omitted these proofs.

In searching the cumulative proof script, we need a decision procedure to check the equality of terms. This is given in Listing B.4. The equality we need here is simple Leibniz equality, in which two structures are equal if they have the same structure in Coq. Often, in our formalization of HBCL, we need to use equivalence relations instead. This arises from our use of map structures, where many underlying structures can denote the same map.

Listing B.4: The propositional calculus formula equality decision function

```
Fixpoint Wff_dec(a b : Wff) { struct a } : {a = b} + {a ≠ b}.
refine (
  match a, b with
    | Wff_St s, Wff_St s′ ⇒
      match ascii_dec s s′ with
        | left _ ⇒ _
        | right _ ⇒ _
      end
    | Wff_Not p, Wff_Not q ⇒ _
    | Wff_And p q, Wff_And r s ⇒ _
    | Wff_Or p q, Wff_Or r s ⇒ _
    | Wff_If p q, Wff_If r s ⇒ _
    | _, _ ⇒ _
  end
).
Defined.
```

The function Wff_dec returns the disjoint union of a proof that the parameters a and b are equal or a proof that they are not equal. The Coq notation + in this context is shorthand for the polymorphic disjoint union type in Coq's Standard Library. The definition of the fixpoint is given within the refine tactic. This enables us to use wildcards which we can fill in later by using proof tactics. We have written out the high level structure explicitly, showing matches on 'a' and 'b'. We are only interested in the matches where we have the same constructor for each of 'a' and 'b', since all other cases cannot be equal and we can immediately use the discriminate tactic to dismiss them. Where we have the same constructor, we either, in the case of a statement, call the decision function from the Coq's ascii Module, or we recursively call Wff_dec. Although this function has a concrete return type, it is a return type that wraps a proof term, and so it is easy and straightforward to fill in the details of the function using proof tactics, which is what we do. We do not show our calls to proof tactics in filling in the wildcards.

Listing B.5: The formula resolution section introduction

```
Section resFuncS.
  Variable P : Wff → Prop.
  Variable specFunc : ∀ wff : Wff, option (P wff).
  Let PList(lOrig lFilter : _) := ∀ w, In w lFilter → P w ∧
    NatDeduct lOrig w.
```

The Section keyword in Coq allows us to open an environment where we are able to declare variables or hypotheses wihout giving values for them or explaining where they came from. This does not lead to inconsistency, because on closing a section, Coq adds an extra argument to any definitions that depend on these variables or hypotheses, which corresponds to the unknown in question. The definition therefore becomes contingent upon supplying the missing information. This paradigm is useful because it allows us to factor out common arguments from a collection of definitions, and makes code clearer. In the section resFuncS, which we open in Listing B.5, we declare two such variables: a proposition about well-formed formulæ, P, whose definition we defer, and a function specFunc, which either produces a proof that a given formula satisfies P, or None. The Let keyword introduces a definition that will be expanded inline when the section is closed. PList is a predicate of two proof scripts lOrig and lFilter, which states that any formula in lFilter satisfies P and is a valid inference given lOrig.

Listing B.6: The ambivalent formula resolution function

```
  Definition specAnyWff(arg : list Wff) :
    option (sig2 P (fun x ⇒ In x arg)).
  refine (
    let fix specAnyWffInner(arg' : list Wff)(arg'Incl : incl arg' arg) :
      option (sig2 P (fun x ⇒ In x arg)) :=
      match arg' as arg' return _ = arg' → _ with
        | wff :: arg'' ⇒ fun J : arg' = wff :: arg'' ⇒
```

```
            match specFunc wff with
              | Some p ⇒ Some (exist2 _ _ wff p _)
              | None ⇒ specAnyWffInner arg'' _
            end
        | nil ⇒ fun _ ⇒ None
      end (eq_refl _)
      in specAnyWffInner arg (incl_refl _)
  ).
Defined.
```

The purpose of the ambivalent formula resolution function of Listing B.6 is to retrieve from the proof script thus far *any* term (hence 'ambivalent') that satisfies `P`. The return type is a σ-type that is loaded with two predicates (`sig2`), meaning that the underlying concrete type satisfies both predicates parametrizing the σ-type. The two predicates in question here are the `P` and proof that the returned formula occurred in the list `arg`.

The function body of `specAnyWff` is given within the `refine` tactic, enabling us to give concrete terms, but leave proof terms, which are not computationally relevant, to be dealt with by later proof tactics. `specAnyWff` is implemented by means of an inner fixpoint, which we declare with '`let fix`'. `specAnyWffInner` is called at the bottom of the outer function definition, where we see '`in specAnyWffInner arg (incl_refl _)`'. `specAnyWffInner` has two arguments: a list of formulæ, and a proof that this list is included in the original list of the outer function, `arg`. When we call `specAnyWffInner` at the bottom of Listing B.6, the argument we give *is* `arg`, so we use the lemma from the Standard Library that a list is included in itself, `incl_refl`, to provide the proof term. The wildcard to `inclRefl` is `arg`, but it can inferred by Coq, so we use the shorter wildcard. This particular wildcard does not therefore correspond to a missing proof that we have to supply later.

Inside `specAnyWffInner`, we see a new `match` notation, which we use pervasively. First, we need to explain why we are returning a function type here from a match of some plain data. The reason is that we require a proof term inside the match that "`arg'`" is the same as "`wff :: arg''`". Without such a term, we would not be able to provide proof that the sublist was included in the original list, or that the formula `wff` occurred in the original list, which we need to know in order to satisfy `P`. However, without more, there is no way to prove this obvious-looking fact. The standard paradigm for dealing with this in Coq involves, rather than returning a concrete proof of these things, returning an implication proof that *generates* an unqualified proof from the necessary equality, which we can supply outside the scope of the match with the `eq_refl` constructor, which produces equality proofs of its argument. We can see `eq_refl` appearing in Listing B.6 after the end of the match, with the type of equality inferred by Coq under the wildcard. Type inference is not complete in dependently typed λ-calculi, so without the "`as arg' return _= arg' -> _`" after the `match` keyword, the code could not have successfully type-checked. Finally, we can obtain a reference to the equality we need by writing the match logic under an anonymous function that binds the form of the equality proof we require to the term `J`. We need to give the type of `J` explicitly; otherwise, it would default to "`arg' = arg'`", which would not help us.

Inside the anonymous function, we make a call to `specFunc`, which is a section variable. If it returns `None`, then `specAnyWffInner` is called again, with a wildcard for the list inclusion proof term for "`arg''`", which we fill in later in the Coq proof script. However, if we find a value under `Some`, then we can construct a member of `sig2 P (fun x => In x arg)` using its constructor `exist2`. The proof of `P` we obtain directly from the match, while the list inclusion proof we fill in under a wildcard later in the Coq proof script, where we use `J` to show the necessary proposition.

Listing B.7: The proof filtration function

```
Definition filterWff(arg : list Wff) : sig (PList arg).
refine (
  let fix filterWffInner(arg′ : list Wff)(arg′Incl : incl arg′ arg) :
    sig (PList arg) :=
    match arg′ as arg′ return _ = arg′ → sig (PList arg) with
      | wff :: arg″ ⇒ fun J : arg′ = wff :: arg″ ⇒
        let filterRec := filterWffInner arg″ _
          in
          match specFunc wff with
            | Some p ⇒ exist _ (wff :: (‘filterRec)) _
            | None ⇒ exist _ (‘filterRec ) _
          end
      | nil ⇒ fun _ ⇒ (exist _ nil _)
    end (eq_refl _)
    in filterWffInner arg (incl_refl _)
).
Defined.
End resFuncS.
```

The proof filtration function of Listing B.7 fulfills a similar function to `specAnyWff`, except that rather than return an arbitrary formula satisfying the filtration criteria, it returns every formula in the list that satisfies the criterion `P`. The structure is the same, except that under the match of `specFunc wff`, the inner fixpoint is called recursively regardless of whether `wff` matched the criterion `P`. If it did (the first branch), it is added to the list obtained with the recursive call; if it did not (the second branch), the recursive call is made straight away. In both cases, the result is a σ-type, so the results are placed under the `exist` constructor, and a wildcard acts as a placeholder for the proof, which we supply later in the script, but do not show here for the sake of brevity. After this function, we see that the section we opened to define these resolution functions is closed with `End resFuncS`.

Listing B.8: The ID rule prover

```
Section tryMatchS.
  Definition try_ID(arg : list Wff)(wff : Wff) : option (NatDeduct arg wff).
  refine (
    let fix memWffInner(arg' : list Wff)(arg'Incl : incl arg' arg) :
      option (NatDeduct arg wff) :=
      match arg' as arg' return _ = arg' → option (NatDeduct arg wff) with
        | nil ⇒ fun _ ⇒ None
        | wff' :: arg'' ⇒ fun J : arg' = wff' :: arg'' ⇒
          match Wff_dec wff' wff with
            | left _ ⇒ Some _
            | right _ ⇒ memWffInner arg'' _
          end
      end (eq_refl _)
      in memWffInner arg (incl_refl _)
  ).
Defined.
```

We now see a series of functions, each of which attempts to prove a proposition using one of the inference rules. `arg` stands for 'logical argument', rather than the argument of a function, and is synonymous with e_{proof} in the semantic rules of appendix B.4. `try_ID` attempts to verify the formula by seeing if it already exists in the argument `arg`. The return type `option (NatDeduct arg wff)` allows either a suitable proof of `NatDeduct arg wff` to be returned under the `Some` constructor, or `None`, if `wff` cannot be proved using this rule, because it did not appear in `arg`. Inside the inner fixpoint, the equality decision function `Wff_dec` is called for each member of the environment. If it returns a proof of equality, `NatDeduct arg wff` can be shown immediately; otherwise, the inner fixpoint is called recursively.

Listing B.9: The *modus ponens* rule prover

```
Let findHasImplicand(A B : Wff) : option (hasImplicand A B).
refine (
  match B with
    | Wff_If a b ⇒
      match Wff_dec A b with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.

Definition try_ModusPonens(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg wff).
refine (
  let implicandMatches :=
    filterWff (hasImplicand wff) (findHasImplicand wff) arg
    in
    let fix tryEachImpl(impls : list Wff)
      (implsIncl : incl impls ('implicandMatches)) :
      option (NatDeduct arg wff) :=
      match impls as impls return _ = impls → option (NatDeduct arg wff) with
        | wff' :: impls' ⇒ fun J : impls = wff' :: impls' ⇒
          match wff' as wff' return _ = wff' → _ with
            | Wff_If x y ⇒ fun J0 : wff' = Wff_If x y ⇒
              match try_ID arg x with
                | Some prf ⇒ Some (ND_ModusPonens arg wff _)
                | None ⇒ tryEachImpl impls' _
              end
            | _ ⇒ fun _ ⇒ !
```

```
                end (eq_refl _)
              | nil ⇒ fun _ ⇒ None
            end (eq_refl _)
            in tryEachImpl (‘implicandMatches) (incl_refl _)
).

Defined.
```

The implementation of *modus ponens* given here follows the operational semantic rule given in rule B.7. In Listing B.9, we see that a helper function `findHasImplicand` is introduced. The idea here is that we need to find an implication from any formula (we do not mind which) to `wff`, as long as the implicant also appears in the environment `arg`. The arbitrary choice of implicant forms a witness allowing us to construct the existential predicate in the definition of `ND_ModusPonens`. We therefore need first to filter the formulæ in the environment to obtain all those implications that have `wff` as an implicand. `findHasImplicand` allows us to do this. It has the same type as the `specFunc` variable of the `resFuncS` section, so we can supply this function as a parameter to the list filtration function `filterWff`. In `try_ModusPonens`, we see that `filterWff` is used in this way to obtain a list of candidate implications. The rest of the function tries to find an implicant for one of them, so that from the combination of both we can deduce `wff`. The first level match in the inner fixpoint, `tryEachImpl`, unpacks the head of the list of candidate implications. The next level match matches on the `Wff` constructors, to obtain an implicant we can search for. Only one of the constructors is given, `Wff_If`. The rest of the constructors are matched by a wildcard, where the result is “`!`”. The “`!`” is Coq notation for a type which allows us to build an inhabitant of any type from the False proposition, using the *ex falso quodlibet* principle (anything follows from the False proposition). We can discharge the proof of Falsity here because we can prove that this branch of match can never be reached. We can prove this from the predicate over `implicandMatches`, from which we can deduce that it only contains well-formed formulæ that are implications.

Having found a candidate implicant `x`, `try_ID` is invoked to try to prove it. If successful (a proof is returned under `Some`), the `ND_ModusPonens` constructor can be used to construct a proof and return it under the `Some` constructor. If unsuccessful (`try_ID` returns `None`) then we continue by recursively calling the inner fixpoint. If we reach the end of the list (`nil`) without proving `wff`, we know it cannot be proved in this environment using the *modus ponens* inference rule, so we return `None`.

Listing B.10: The *modus tollens* rule prover

```
Definition try_ModusTollens(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg (Wff_Not wff)).
refine (
  let implicantMatches :=
    filterWff (hasImplicant wff)
    (findHasImplicant wff) arg
    in
    let fix tryEachImpl(impls : list Wff)
      (implsIncl : incl impls (‘implicantMatches)) :
      option (NatDeduct arg (Wff_Not wff)) :=
      match impls as impls return _ = impls →
        option (NatDeduct arg (Wff_Not wff)) with
        | wff′ :: impls′ ⇒ fun J : impls = wff′ :: impls′ ⇒
          match wff′ as wff′ return _ = wff′ → _ with
            | Wff_If x y ⇒ fun J0 : wff′ = Wff_If x y ⇒
              match try_ID arg (Wff_Not y) with
                | Some prf ⇒ Some (ND_ModusTollens arg wff _)
                | None ⇒ tryEachImpl impls′ _
              end
            | _ ⇒ fun _ ⇒ !
          end (eq_refl _)
        | nil ⇒ fun _ ⇒ None
      end (eq_refl _)
      in tryEachImpl (‘implicantMatches) (incl_refl _)
).

Defined.
```

The *modus tollens* rule of Listing B.10 has an almost identical structure. We note, however, that the return type ‘`option (NatDeduct arg (Wff_Not(wff))`’ now contains the *negation* of the argument `wff`. This accords with what we expect from the *modus tollens* rule, both from the definition in the semantic domain, and from the operational semantic rule we gave in rule B.8. The remaining functions for trying particular inference rules do not introduce any new Coq syntax or features. We give them for completeness below.

Listing B.11: The left-hand simplification rule prover

```
Let hasAndA(A B : Wff) := ∃ C, Wff_And A C = B.
Let findHasAndA(A B : Wff) : option (hasAndA A B).
refine (
  match B with
    | Wff_And a b ⇒
      match Wff_dec A a with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.
Definition try_SimplA(arg : list Wff)(wff : Wff) : option (NatDeduct arg wff).
refine (
  let anyAndA := specAnyWff (hasAndA wff) (findHasAndA wff) arg
    in
    match anyAndA as anyAndA return _ = anyAndA → _ with
      | Some (exist2 wff′ p q) ⇒
        fun J : anyAndA = Some (exist2 _ _ wff′ p q) ⇒
          Some (ND_SimplA arg wff _)
      | None ⇒ fun _ ⇒ None
    end (eq_refl _)
).
Defined.
```

Listing B.11 implements rule B.9. A new filtration function for `filterWff` is provided, `findHasAndA`, which filters the existing formulæ to find those that have `wff` as their left-hand argument. This time, we only need to find one formula in the environment that satisfies this requirement, so we pass the filtration function to `specAnyWff`. When we match the result of this call under `Some`, we further deconstruct the σ-type with the `exist2` constructor, matching the two proof terms it yields as '`p`' and '`q`'. These proof terms are needed so that we can fill in the wildcard in `ND_SimplA arg wff _)` using proof tactics later in the Coq script.

This method of predicate extraction is used extensively. In simple cases, Coq's `Program` tactic can automatically deal with some of the repetitive aspects of producing these sorts of definitions. We avoid the `Program` tactic here because it can obscure what is happening. The method becomes slightly more complicated when the inductive propositional predicates have multiple constructors, in which case the irrelevant ones are ruled out by proving falsehood in the context of their matches. Again, Coq has proof tactics that make this easier. Although when specifying concrete functions, we must in general only match on concrete types, where we are trying to generate a pure propositional term, for instance, when constructing the propositional part of a new σ-type, we can match on propositional types. Since we have proof-irrelevance of propositional functions from the Curry-Howard isomorphism, the form of the proof term that the proof tactics substitute into propositional wildcards in the refine tactic is of no concern, as long as the type-checker accepts it.

Listing B.12: The right-hand simplification rule prover

```
Let hasAndB(A B : Wff) := ∃ C, Wff_And C A = B.
Let findHasAndB(A B : Wff) : option (hasAndB A B).
refine (
  match B with
    | Wff_And a b ⇒
      match Wff_dec A b with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.
Definition try_SimplB(arg : list Wff)(wff : Wff) : option (NatDeduct arg wff).
refine (
  let anyAndB := specAnyWff (hasAndB wff) (findHasAndB wff) arg
    in
    match anyAndB as anyAndB return _ = anyAndB → _ with
      | Some (exist2 wff' p q) ⇒
        fun J : anyAndB = Some (exist2 _ _ wff' p q) ⇒
          Some (ND_SimplB arg wff _)
      | None ⇒ fun _ ⇒ None
    end (eq_refl _)
).
Defined.
```

Listing B.12 implements rule B.10. `try_SimplB` is constructed in exactly the same way as try_SimplA, except that the filtration occurs on the right-hand member of the conjunction rather than the left.

Listing B.13: The conjunction rule prover

```
Definition try_Conj(arg : list Wff)(wff wff' : Wff) :
  option (NatDeduct arg (Wff_And wff wff')) :=
  match try_ID arg wff with
    | Some prf ⇒
      match try_ID arg wff' with
        | Some prf' ⇒ Some (ND_Conj arg wff wff' (conj prf prf'))
        | None ⇒ None
      end
    | None ⇒ None
  end.
```

Listing B.13 implements rule B.11. It calls try_ID twice, once with wff and once with wff'. If it finds both, the proof is constructed directly from the two proofs using the built-in conj constructor of Coq which constructs Coq's axiomatization of conjunction that we used in the definition of ND_Conj. If the function does not find one of the formulæ it is looking for, then it returns None.

Listing B.14: The left-hand disjunction rule prover

```
Definition try_DisjA(arg : list Wff)(wff wff' : Wff) :
  option (NatDeduct arg (Wff_Or wff wff')) :=
  match try_ID arg wff with
    | Some prf ⇒ Some (ND_DisjA arg wff wff' prf)
    | None ⇒ None
  end.
```

Listing B.14 implements rule B.12. The left-hand disjunction function constructs the 'or' formula from wff and wff' and uses try_ID to find this in the environment. If it does so successfully, then the inference proof of the first argument wff is constructed directly from the matched proof term and the definition of ND_DisjA. Otherwise, None is returned.

Listing B.15: The right-hand disjunction rule prover

```
Definition try_DisjB(arg : list Wff)(wff wff': Wff) :
  option (NatDeduct arg (Wff_Or wff wff')) :=
  match try_ID arg wff' with
    | Some prf ⇒ Some (ND_DisjB arg wff wff' prf)
    | None ⇒ None
  end.
```

Listing B.15 implements rule B.13. The right-hand disjunction function works in exactly the same way as the left, except this time ND_DisjA is used to prove the second argument wff' from the disjunction.

Listing B.16: The left-hand conjunctive syllogism rule prover

```
Let hasNandA(A B : Wff) := ∃ C, (Wff_Not (Wff_And A C)) = B.
Let findHasNandA(A B : Wff) : option (hasNandA A B).
refine (
  match B with
    | (Wff_Not (Wff_And a b)) ⇒
      match Wff_dec A a with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.
Definition try_ConjSyllogA(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg (Wff_Not wff)).
refine (
  let NandAMatches :=
    filterWff (hasNandA wff)
    (findHasNandA wff) arg
    in
    let fix tryEachNand(nands : list Wff)
      (nandsIncl : incl nands ('NandAMatches)) :
      option (NatDeduct arg (Wff_Not wff)) :=
      match nands as nands return _ = nands →
        option (NatDeduct arg (Wff_Not wff)) with
        | wff' :: nands' ⇒ fun J : nands = wff' :: nands' ⇒
          match wff' as wff' return _ = wff' → _ with
            | Wff_Not (Wff_And x y) ⇒ fun J0 : wff' = Wff_Not (Wff_And x y) ⇒
              match try_ID arg y with
                | Some prf ⇒ Some (ND_ConjSyllogA arg wff _)
                | None ⇒ tryEachNand nands' _
              end
            | _ ⇒ fun _ ⇒ !
          end (eq_refl _)
        | nil ⇒ fun _ ⇒ None
      end (eq_refl _)
      in tryEachNand ('NandAMatches) (incl_refl _)
).


Defined.
```

Listing B.16 implements rule B.14. The Coq function left-hand conjunctive syllogism, `try_ConjSyllogA`, works in a similar way to the *modus tollens* function, in that it is trying to draw an inference negating the argument of the rule from two formulæ in the environment. It first filters the environment using a new filter function, `findHasNandA`, to locate suitable negations of conjunctions that can be used to draw the correct inference. Like the *modus tollens* rule, it then systematically searches the list for a positive statement from the right-hand side of the negated conjunction. If it finds one, it can construct a proof with `ND_ConjSyllogA`. Otherwise, the function returns `None`.

Listing B.17: The right-hand conjunctive syllogism rule prover

```
Let hasNandB(A B : Wff) := ∃ C, (Wff_Not (Wff_And C A)) = B.
Let findHasNandB(A B : Wff) : option (hasNandB A B).
```

```
refine (
  match B with
    | (Wff_Not (Wff_And a b)) ⇒
      match Wff_dec A b with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.
Definition try_ConjSyllogB(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg (Wff_Not wff)).
refine (
  let NandBMatches :=
    filterWff (hasNandB wff)
    (findHasNandB wff) arg
    in
    let fix tryEachNand(nands : list Wff)
      (nandsIncl : incl nands ('NandBMatches)) :
      option (NatDeduct arg (Wff_Not wff)) :=
      match nands as nands return _ = nands →
        option (NatDeduct arg (Wff_Not wff)) with
        | wff' :: nands' ⇒ fun J : nands = wff' :: nands' ⇒
          match wff' as wff' return _ = wff' → _ with
            | Wff_Not (Wff_And x y) ⇒ fun J0 : wff' = Wff_Not (Wff_And x y) ⇒
              match try_ID arg x with
                | Some prf ⇒ Some (ND_ConjSyllogB arg wff _)
                | None ⇒ tryEachNand nands' _
              end
            | _ ⇒ fun _ ⇒ !
          end (eq_refl _)
        | nil ⇒ fun _ ⇒ None
      end (eq_refl _)
      in tryEachNand ('NandBMatches) (incl_refl _)
).


Defined.
```

Listing B.17 implements rule B.15. The right-hand rule operates in exactly the same way as the left-hand rule, but now the positive statement in the environment must match that on the left-hand side of the negated conjunction, in order to show the negation of the formula on the right-hand side.

Listing B.18: The left-hand disjunctive syllogism rule prover

```
Let hasOrA(A B : Wff) := ∃ C, Wff_Or A C = B.
Let findHasOrA(A B : Wff) : option (hasOrA A B).
refine (
  match B with
    | Wff_Or a b ⇒
      match Wff_dec A a with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.
Definition try_DisjSyllogA(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg wff).
refine (
  let OrAMatches := filterWff (hasOrA wff) (findHasOrA wff) arg
    in
    let fix tryEachOr(ors : list Wff)
      (orsIncl : incl ors (‘OrAMatches)) :
      option (NatDeduct arg wff) :=
      match ors as ors return _ = ors →
        option (NatDeduct arg wff) with
        | wff′ :: ors′ ⇒ fun J : ors = wff′ :: ors′ ⇒
          match wff′ as wff′ return _ = wff′ → _ with
            | Wff_Or x y ⇒ fun J0 : wff′ = Wff_Or x y ⇒
              match try_ID arg (Wff_Not y) with
                | Some prf ⇒ Some (ND_DisjSyllogA arg wff _)
                | None ⇒ tryEachOr ors′ _
              end
            | _ ⇒ fun _ ⇒ !
          end (eq_refl _)
        | nil ⇒ fun _ ⇒ None
      end (eq_refl _)
      in tryEachOr (‘OrAMatches) (incl_refl _)
).

Defined.
```

Listing B.18 implements rule B.16. The structure of the function for the construction of an inference from a disjunctive syllogism is similar to that of the *modus ponens* rule. We are trying to show `wff`, given a disjunction containing (in the case of the left-handed rule) `wff` on the left-hand side, and a negation somewhere in the environment of whatever is on the right-hand side. Again, we need a new filtration function for this purpose, `findHasOrA`.

Listing B.19: The right-hand disjunctive syllogism rule prover

```
Let hasOrB(A B : Wff) := ∃ C, Wff_Or C A = B.
Let findHasOrB(A B : Wff) : option (hasOrB A B).
refine (
  match B with
    | Wff_Or a b ⇒
      match Wff_dec A b with
        | left prf ⇒ Some _
        | right _ ⇒ None
      end
    | _ ⇒ None
  end
).
Defined.
Definition try_DisjSyllogB(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg wff).
refine (
  let OrBMatches := filterWff (hasOrB wff) (findHasOrB wff) arg
    in
    let fix tryEachOr(ors : list Wff)
      (orsIncl : incl ors (‘OrBMatches)) :
      option (NatDeduct arg wff) :=
      match ors as ors return _ = ors →
        option (NatDeduct arg wff) with
        | wff′ :: ors′ ⇒ fun J : ors = wff′ :: ors′ ⇒
          match wff′ as wff′ return _ = wff′ → _ with
```

```
                | Wff_Or x y ⇒ fun J0 : wff' = Wff_Or x y ⇒
                   match try_ID arg (Wff_Not x) with
                      | Some prf ⇒ Some (ND_DisjSyllogB arg wff _)
                      | None ⇒ tryEachOr ors' _
                   end
                | _ ⇒ fun _ ⇒ !
             end (eq_refl _)
          | nil ⇒ fun _ ⇒ None
       end (eq_refl _)
       in tryEachOr ('OrBMatches) (incl_refl _)
).

Defined.
```

Listing B.19 implements rule B.17. The right-handed disjunction prover has exactly the same structure as the left-handed one, except that we are now trying to prove the formula on the right-hand side of a disjunction.

Listing B.20: The hypothetical syllogism rule prover

```
Definition try_HypSyllog(arg : list Wff)(wff wff' : Wff) :
   option (NatDeduct arg (Wff_If wff wff')).
refine (
   let implicantMatches :=
      filterWff (hasImplicant wff) (findHasImplicant wff) arg
      in let implicandMatches :=
         filterWff (hasImplicand wff') (findHasImplicand wff') arg
         in
         let fix tryEachImpl(impls : list Wff)
            (implsIncl : incl impls ('implicandMatches)) :
            option (NatDeduct arg (Wff_If wff wff')) :=
            match impls as impls return _ = impls → _ with
               | wff'' :: impls' ⇒ fun J : impls = wff'' :: impls' ⇒
                  match wff'' as wff'' return _ = wff'' → _ with
                     | Wff_If x y ⇒ fun J0 : wff'' = Wff_If x y ⇒
                        match specAnyWff (hasImplicand x) (findHasImplicand x)
                           ('implicantMatches) with
                           | Some implA ⇒
                              Some (ND_HypSyllog arg wff wff' _)
                           | None ⇒ tryEachImpl impls' _
                        end
                     | _ ⇒ fun _ ⇒ !
                  end (eq_refl _)
               | nil ⇒ fun _ ⇒ None
            end (eq_refl _)
            in tryEachImpl ('implicandMatches) (incl_refl _)
).

Defined.
```

Listing B.20 implements rule B.18. The function for constructing proofs of hypothetical syllogisms, `try_HypSyllog`, attempts to show that `wff` implies `wff'` by finding a chain of implications that interpolates some third formula. `try_HypSyllog` does this by first using a filter function to construct a list of inferences that have `wff` as their implicant, and then using another filter function to construct a list of inferences that have `wff'` as their implicand. The function then attempts to find a pair from these two lists in which the implicand of the implication from the first list matches the implicant of the implication from the second list. Going systematically through the list of implications that imply `wff'`, it uses the `specAnyWff` filter to search the list of implications with `wff` as their implicant to find a matching implicand. If one is found, a proof can be constructed to satisfy `ND_HypSyllog` in the environment. The proof was the most complex of all of these examples, as it relied on predicates that had to be stripped from the results of three separate filtration operations.

Listing B.21: The dilemma rule prover

```
Let isOr(A : Wff) := ∃ B, ∃ C, Wff_Or B C = A.
Let findIsOr(A : Wff) : option (isOr A).
refine (
  match A with
    | Wff_Or _ _ ⇒ Some _
    | _ ⇒ None
  end
).
Defined.
Definition try_Dilemma(arg : list Wff)(wff : Wff) :
  option (NatDeduct arg wff).
refine (
  let isOrMatches := filterWff isOr findIsOr arg
    in
    let fix tryEachOr(ors : list Wff)
      (orsIncl : incl ors (‘isOrMatches)) :
      option (NatDeduct arg wff) :=
      match ors as ors return _ = ors → _ with
        | wff' :: ors' ⇒ fun J : ors = wff' :: ors' ⇒
          match wff' as wff' return _ = wff' → _ with
            | Wff_Or x y ⇒ fun J0 : wff' = Wff_Or x y ⇒
              match try_ID arg (Wff_If x wff) with
                | Some prf ⇒
                  match try_ID arg (Wff_If y wff) with
                    | Some prf' ⇒ Some (ND_Dilemma arg wff _)
                    | None ⇒ tryEachOr ors' _
                  end
                | None ⇒ tryEachOr ors' _
              end
            | _ ⇒ fun _ ⇒ !
          end (eq_refl _)
        | nil ⇒ fun _ ⇒ None
      end (eq_refl _)
      in tryEachOr (‘isOrMatches) (incl_refl _)
).

Defined.
End tryMatchS.
```

Listing B.21 implements rule B.19. The filtration function `findIsOr` filters all of the disjunctions in the environment, and then checks systematically for implications that enable us to conclude that `wff` is proven in the environment. `try_ID` is used twice, first to look for an implication of the left-hand side of the disjunction to `wff`, and then from the right-hand side of the disjunction to `wff`. If both can be found for a particular disjunction, then a proof can immediately be constructed using the `ND_Dilemma` constructor. If the list of disjunctions is exhausted without finding a suitable pair of implications, then None is returned.

We now introduce some extra functions that do not have a direct analogue in the structural operational semantics. These functions group the inference rules by the relationship of their argument to the formula that they prove.

Listing B.22: The ID rule searcher

```
Definition tryIDSearch(arg : list Wff)(A : Wff) : option (NatDeduct arg A) :=
  let fix trySearchInner
    (tryList :
      list (∀ (arg : list Wff)(wff : Wff), option (NatDeduct arg wff))) :
    option (NatDeduct arg A) :=
    match tryList with
      | func :: tryList′ ⇒
        match func arg A with
          | Some wff′ ⇒ Some wff′
          | None ⇒ trySearchInner tryList′
        end
      | nil ⇒ None
    end
    in trySearchInner (try_ID :: try_ModusPonens :: try_SimplA ::
      try_SimplB :: try_DisjSyllogA :: try_DisjSyllogB :: try_Dilemma :: nil).
```

The function of Listing B.22 tries all of the rules that try to establish their argument. The list of functions at the bottom of tryIDSearch is processed sequentially by trySearchInner. This higher-order construction is used to avoid excessive use of nested matches. When each function is tried with the call to func arg A, a match on a Some constructor causes an immediately successful return, while a match on None causes a recursive call with the unsuccessful function removed from the list. If the end (nil) of the list is reached without any successful matches, the function returns None.

Listing B.23: The Not rule searcher

```
Definition tryNotSearch(arg : list Wff)(A : Wff) :
  option (NatDeduct arg (Wff_Not A)) :=
  let fix trySearchInner
    (tryList : list (∀ (arg : list Wff) (wff : Wff),
      option (NatDeduct arg (Wff_Not wff)))) :
    option (NatDeduct arg (Wff_Not A)) :=
    match tryList with
      | func :: tryList′ ⇒
        match func arg A with
          | Some wff′ ⇒ Some wff′
          | None ⇒ trySearchInner tryList′
        end
      | nil ⇒ None
    end
    in trySearchInner ((fun a w ⇒ try_ID a (Wff_Not w)) :: try_ModusTollens ::
      try_ConjSyllogA :: try_ConjSyllogB :: nil).
```

The function tryNotSearch of Listing B.23 follows exactly the same structure as tryIDSearch, except that it now calls on functions that try to prove the negation of their argument. Again, these functions appear in a list at the bottom of the function in the call to the inner fixpoint. The reason we cannot use exactly the same function for each of these inference-searching functions is that the number of arguments and return types are connected and vary. A higher order construction that would be able to cope with this would look clumsy, having to condense variable numbers of arguments into a single argument, and give many more type parameters.

Listing B.24: The And rule searcher

```
Definition tryAndSearch(arg : list Wff)(A B : Wff) :
  option (NatDeduct arg (Wff_And A B)) :=
  let fix trySearchInner
    (tryList : list (∀ (arg : list Wff) (wff wff' : Wff),
      option (NatDeduct arg (Wff_And wff wff')))) :
    option (NatDeduct arg (Wff_And A B)) :=
    match tryList with
      | func :: tryList' ⇒
        match func arg A B with
          | Some wff' ⇒ Some wff'
          | None ⇒ trySearchInner tryList'
        end
      | nil ⇒ None
    end
    in trySearchInner (try_Conj :: nil).
```

`tryAndSearch` of Listing B.24 follows the same pattern as the other search functions, even though there is only one prover function in the list this time: `try_Conj`. We keep the same function structure purely for consistency. This function differs from those before in that it has two arguments, one for either side of the conjunction.

Listing B.25: The Or rule searcher

```
Definition tryOrSearch(arg : list Wff)(A B : Wff) :
  option (NatDeduct arg (Wff_Or A B)) :=
  let fix trySearchInner
    (tryList : list (∀ (arg : list Wff) (wff wff' : Wff),
      option (NatDeduct arg (Wff_Or wff wff')))) :
    option (NatDeduct arg (Wff_Or A B)) :=
    match tryList with
      | func :: tryList' ⇒
        match func arg A B with
          | Some wff' ⇒ Some wff'
          | None ⇒ trySearchInner tryList'
        end
      | nil ⇒ None
    end
    in trySearchInner (try_DisjA :: try_DisjB :: nil).
```

The function `tryOrSearch` of Listing B.25 follows the same pattern as `tryAndSearch`, except that it now looks for implications of disjunctions, which can be either left-handed (`try_DisjA`) or right-handed (`try_DisjB`). Again, there are two arguments to this function, one for either side of the disjunction.

Listing B.26: The If rule searcher

```
Definition tryIfSearch(arg : list Wff)(A B : Wff) :
  option (NatDeduct arg (Wff_If A B)) :=
  let fix trySearchInner
     (tryList : list (∀ (arg : list Wff) (wff wff' : Wff),
        option (NatDeduct arg (Wff_If wff wff')))) :
     option (NatDeduct arg (Wff_If A B)) :=
     match tryList with
        | func :: tryList' ⇒
           match func arg A B with
              | Some wff' ⇒ Some wff'
              | None ⇒ trySearchInner tryList'
           end
        | nil ⇒ None
     end
     in trySearchInner (try_HypSyllog :: nil).
```

Finally, `tryIfSearch` of Listing B.26 searches for inferences that build an implication out of its two arguments. There is only one inference rule that produces conclusions of this type: it is `try_HypSyllog`.

Listing B.27: The top rule searcher

```
Definition searchArg
  (arg : list Wff)(wff : Wff) :
  option (NatDeduct arg wff) :=
  match (tryIDSearch arg wff) with
     | Some prf ⇒ Some prf
     | None ⇒
        match wff with
           | Wff_St _ ⇒ None
           | Wff_Not A ⇒ tryNotSearch arg A
           | Wff_And A B ⇒ tryAndSearch arg A B
           | Wff_Or A B ⇒ tryOrSearch arg A B
           | Wff_If A B ⇒ tryIfSearch arg A B
        end
  end.
```

The top-level inference search function, `searchArg` of Listing B.27, first tries the identity function, `tryID_Search`. This can provide a direct proof, if the formula to be proved is a formula that already appears verbatim in the argument, or if it can be shown by one of the inference rules that proves its argument. If this does not succeed, then one of the functions that tries to make an inference of a formula depending on the arguments and return type is called, depending on what is found when the candidate formula `wff` is deconstructed. If we are trying to prove a statement, which corresponds to a letter, then it cannot be proven here, since if it was capable of being proved in this environment, it would have to be proved by an inference tried by `tryIDSearch`. Each of the other matches calls the worker function with the name and arguments corresponding to the relevant match.

Listing B.28: The proof checker function

```
Fixpoint checkProof(premises : list Wff)(script : list Wff) :
  option (sig (ND_ValidArgument premises)) :=
  match script with
    | wff :: script' ⇒
      let prevScriptOpt := checkProof premises script' in
        match prevScriptOpt with
          | None ⇒ None
          | Some prevScript ⇒
            match searchArg (premises ++ (‘prevScript)) wff with
              | None ⇒ None
              | Some ndPred ⇒
                Some (exist _ _ (ND_ValArg_cons premises wff (‘prevScript)
                  (proj2_sig prevScript) ndPred))
            end
        end
    | nil ⇒ Some (exist _ _ (ND_ValArg_empty premises))
  end.
```

The function `checkProof` of Listing B.28 implements the top-level operational semantic rules of rule B.3, rule B.4, rule B.5 and rule B.6. Rule B.6, the recursive base case, is implemented in the last match of the function, where an empty valid argument built with `ND_ValArg_empty` is constructed for an empty candidate argument. The other branches of the rule correspond to the options that arise when `checkProof` is recursively invoked. The situation in rule B.3, where a recursive attempt to build a valid argument has failed with `None`, immediately short-circuits to a return of `None`, as there is no way that we can make an invalid argument valid by adding formulæ to it. The result of the match on calling `searchArg` on the local candidate formula `wff` accounts for the remaining two branches of the top-level operational semantic rule. The ++ operator of Coq denotes list concatenation. The case where `searchArg` returns `None` corresponds to rule B.4, where no proof that `wff` was proven in the environment could be found. Finally, where `searchArg` produces a proof under the `Some` constructor, an iteration of the inductive valid argument constructor can proceed, as at this point in the function, we are in a context where we can build the necessary proof terms for `ND_ValArg_cons`. This corresponds to rule B.5.

Appendix C

Further Harmonic Box Coordination Language Syntax and Semantics

C.1 Concrete syntax

C.1.1 Primitive tokens

C.1.1.1 Lexing

$$\langle\text{romanletter}\rangle ::= \{\ \texttt{a}\ ,\ \ldots\ ,\ \texttt{z}\ \} \mid \{\ \texttt{A}\ ,\ \ldots\ ,\ \texttt{Z}\ \} \tag{C.1}$$

$$\langle\text{arabicnumeral}\rangle ::= \{\ \texttt{0}\ ,\ \ldots\ ,\ \texttt{9}\ \} \tag{C.2}$$

$$\langle\text{id}\rangle ::= \{\ \texttt{\_}\ ,\ \langle\text{romanletter}\rangle\ \}\{\ \langle\text{romanletter}\rangle\ ,\ \langle\text{arabicnumeral}\rangle\ ,\ \texttt{'}\ \}* \tag{C.3}$$

$$\langle\text{boolconst}\rangle ::= \texttt{true} \mid \texttt{false} \tag{C.4}$$

C.1.1.2 Constants

$$\langle\text{natconst}\rangle ::= \{\ \langle\text{arabicnumeral}\rangle\ \}+ \tag{C.5}$$

$$\langle\text{intconst}\rangle ::= \langle\text{natconst}\rangle \mid -\ \langle\text{intconst}\rangle \quad \text{precedence 1 right associativity} \tag{C.6}$$

C.1.2 Concrete syntax common to coordination and expression languages

$$\begin{array}{lcl} \langle\text{type}\rangle & ::= & \langle\text{basetype}\rangle \\ & | & \langle\text{tupletype}\rangle \\ & | & \langle\text{recordtype}\rangle \\ & | & \langle\text{typid}\rangle \end{array} \tag{C.7}$$

$$\langle\text{basetype}\rangle \quad ::= \quad \texttt{bool} \tag{C.8}$$

$$\langle\text{tupletype}\rangle \quad ::= \quad (\ \langle\text{type}\rangle_1\ ,\ \ldots\ ,\ \langle\text{type}\rangle_n\ ) \quad n \geq 1 \tag{C.9}$$

$$\langle\text{vardeclprim}\rangle \quad ::= \quad \langle\text{varid}\rangle\ :\ \langle\text{type}\rangle \tag{C.10}$$

$$\langle\text{recordtype}\rangle \quad ::= \quad \{\ \langle\text{type}\rangle_1\ ;\ \ldots\ ;\ \langle\text{vardeclprim}\rangle_n\ \} \quad n \geq 1 \tag{C.11}$$

$$\langle\text{typid}\rangle \quad ::= \quad \langle\text{id}\rangle \tag{C.12}$$

$$\langle\text{varid}\rangle \quad ::= \quad \langle\text{id}\rangle \tag{C.13}$$

$$\langle\text{typedef}\rangle \quad ::= \quad \texttt{type}\ \langle\text{typeid}\rangle\ \langle\text{type}\rangle \tag{C.14}$$

$$\langle\text{utypedef}\rangle \quad ::= \quad \texttt{oidtype}\ \langle\text{typeid}\rangle\ \langle\text{typeid}\rangle \tag{C.15}$$

$$\langle\text{freq}\rangle \quad ::= \quad [\ \langle\text{natconst}\rangle\ /\ \langle\text{natconst}\rangle\ ] \tag{C.16}$$

$$\langle\text{htypedef}\rangle \quad ::= \quad \texttt{htype}\ \langle\text{typeid}\rangle\ :\ \langle\text{typeid}\rangle\ \langle\text{freq}\rangle \tag{C.17}$$

C.1.3 Concrete syntax for coordination language

$$\begin{array}{lcll} \langle\text{linst}\rangle & ::= & \texttt{linst}\ \{\ \langle\text{id}\rangle\ \langle\text{linstdecl}\rangle_1\ \texttt{;}\ \ldots\ \texttt{;}\ \langle\text{linstdecl}\rangle_n\ \}\quad n \geq 1 & \\ & \mid & \texttt{linst}\ \langle\text{id}\rangle\ \texttt{:}\ \texttt{linstref} & \end{array} \tag{C.18}$$

$$\langle\text{linstref}\rangle\ ::=\ \langle\text{id}\rangle_1\ \texttt{.}\ \ldots\ \texttt{.}\ \langle\text{id}\rangle_n\ \texttt{.}\ \langle\text{id}\rangle\quad n \geq 1 \tag{C.19}$$

$$\langle\text{utyperef}\rangle\ ::=\ \langle\text{id}\rangle_1\ \texttt{.}\ \ldots\ \texttt{.}\ \langle\text{id}\rangle_n\ \texttt{.}\ \langle\text{typeid}\rangle\quad n \geq 1 \tag{C.20}$$

$$\langle\text{htyperef}\rangle\ ::=\ \langle\text{id}\rangle_1\ \texttt{.}\ \ldots\ \texttt{.}\ \langle\text{id}\rangle_n\ \texttt{.}\ \langle\text{typeid}\rangle\quad n \geq 1 \tag{C.21}$$

$$\langle\text{llib}\rangle\ ::=\ \texttt{linst}\ \{\ \langle\text{id}\rangle\ \langle\text{libdecl}\rangle_1\ \texttt{;}\ \ldots\ \texttt{;}\ \langle\text{libdecl}\rangle_n\ \}\quad n \geq 1 \tag{C.22}$$

$$\begin{array}{lcl} \langle\text{typeany}\rangle & ::= & \langle\text{typedef}\rangle \\ & \mid & \langle\text{utypedef}\rangle \\ & \mid & \langle\text{htypedef}\rangle \end{array} \tag{C.23}$$

$$\begin{array}{lcll} \langle\text{linstdecl}\rangle & ::= & \langle\text{linst}\rangle & \\ & \mid & \langle\text{llib}\rangle & \\ & \mid & \texttt{hbox}\ \langle\text{boxid}\rangle\ \langle\text{memfbids}\rangle\ \texttt{:}\ \langle\text{membfids}\rangle\ -\!>\ \langle\text{freq}\rangle\ \{\ \langle\text{uprogram}\rangle\ \} & \\ & \mid & \texttt{mem(bf)}\ \langle\text{membfid}\rangle\ \texttt{:}\ \langle\text{htyperef}\rangle\ \langle\text{ttfl}\rangle & \\ & \mid & \texttt{mem(fb)}\ \langle\text{memfbid}\rangle\ \texttt{:}\ \langle\text{htyperef}\rangle\ \langle\text{ttfl}\rangle & \\ & \mid & \langle\text{typeany}\rangle & \\ & \mid & \texttt{observe}\ \{\ \langle\text{memfbidref}\rangle_1\ \texttt{;}\ \ldots\ \texttt{;}\ \langle\text{memfbidref}\rangle_n\ \} & n \geq 1 \\ & \mid & \texttt{manifest}\ \{\ \langle\text{membfidref}\rangle_1\ \texttt{;}\ \ldots\ \texttt{;}\ \langle\text{membfidref}\rangle_n\ \} & n \geq 1 \\ & \mid & \texttt{fifo}\ \langle\text{membfidref}\rangle\ \texttt{to}\ \langle\text{memfbidref}\rangle & \end{array} \tag{C.24}$$

$$\begin{array}{lcl} \langle\text{libdecl}\rangle & ::= & \langle\text{linst}\rangle \\ & \mid & \langle\text{llib}\rangle \end{array} \tag{C.25}$$

$$\begin{array}{lcl} \langle\text{ttfl}\rangle & ::= & \texttt{ttl}\ (\ \langle\text{natconst}\rangle\ ) \\ & \mid & \texttt{tfl}\ (\ \langle\text{natconst}\rangle\ ) \end{array} \tag{C.26}$$

$$\langle\text{membfid}\rangle\ ::=\ \langle\text{id}\rangle \tag{C.27}$$

$$\langle\text{membfidref}\rangle \quad ::= \quad \langle\text{linstref}\rangle \ . \ \langle\text{membfid}\rangle \tag{C.28}$$

$$\langle\text{memfbid}\rangle \quad ::= \quad \langle\text{id}\rangle \tag{C.29}$$

$$\langle\text{memfbidref}\rangle \quad ::= \quad \langle\text{linstref}\rangle \ . \ \langle\text{memfbid}\rangle \tag{C.30}$$

$$\langle\text{boxid}\rangle \quad ::= \quad \langle\text{id}\rangle \tag{C.31}$$

C.1.4 Concrete syntax for expression language

$$\langle\text{uprogram}\rangle \quad ::= \quad \langle\text{udecl}\rangle_1 \ ; \ \ldots \ ; \ \langle\text{udecl}\rangle_n \quad n \geq 1 \tag{C.32}$$

$$\begin{array}{lcl} \langle\text{udecl}\rangle & ::= & \langle\text{vardecl}\rangle \\ & | & \langle\text{vardef}\rangle \end{array} \tag{C.33}$$

$$\begin{array}{lcl} \langle\text{expr}\rangle & ::= & \langle\text{patt}\rangle \\ & | & \langle\text{constr}\rangle \\ & | & \langle\text{varid}\rangle \ (\ \langle\text{expr}\rangle_1 \ , \ \ldots \ , \ \langle\text{expr}\rangle_n \ ) \quad n \geq 1 \end{array} \tag{C.34}$$

$$\langle\text{exprassoc}\rangle \quad ::= \quad \langle\text{varid}\rangle \ = \ \langle\text{expr}\rangle \tag{C.35}$$

$$\begin{array}{lcll} \langle\text{constr}\rangle & ::= & \langle\text{boolconst}\rangle & \\ & | & (\ \langle\text{expr}\rangle_1 \ , \ \ldots \ , \ \langle\text{expr}\rangle_n \ ) & n \geq 1 \\ & | & \{ \ \langle\text{exprassoc}\rangle_1 \ ; \ \ldots \ ; \ \langle\text{exprassoc}\rangle_n \ \} & n \geq 1 \end{array} \tag{C.36}$$

$$\langle\text{fundeclprim}\rangle \quad ::= \quad \langle\text{varid}\rangle \ : \ \langle\text{type}\rangle \ -> \ \langle\text{type}\rangle \tag{C.37}$$

$$\begin{array}{lcl} \langle\text{vardecl}\rangle & ::= & \langle\text{vardeclprim}\rangle \\ & | & \langle\text{fundeclprim}\rangle \end{array} \tag{C.38}$$

$$\begin{array}{lcl} \langle\text{vardef}\rangle & ::= & \langle\text{vardeclprim}\rangle \ ::= \ \langle\text{expr}\rangle \\ & | & \langle\text{fundeclprim}\rangle \ ::= \ \langle\text{expr}\rangle \end{array} \tag{C.39}$$

$$\langle\text{patt}\rangle \quad ::= \quad \langle\text{varid}\rangle \;.\; \langle\text{datresolve}\rangle_1 \;.\; \ldots \;.\; \langle\text{datresolve}\rangle_n \;\}\quad n \geq 1 \tag{C.40}$$

$$\begin{array}{lcl} \langle\text{datresolve}\rangle & ::= & \langle\text{varid}\rangle \\ & | & \langle\text{natconst}\rangle \end{array} \tag{C.41}$$

C.2 Static semantics as a *static semantic object*

These rules are couched so as to correspond to total pattern matching functions on abstract syntax trees representing conclusions.

The semantics are given in an evaluation style.

C.2.1 Fragment common to expression and coordination languages

The following rules are those of the static interface between the coordination and expression languages on which our present untimed expression language depends.

Matches on the *type* syntactic object:

$$\frac{}{\left\{ U_\top \;\vdash \left\{ \texttt{typeBasetype}\left(b_{basetype}\right) \right\} \Rightarrow \left\{ \mathfrak{T}_{\top\perp}\left(\frac{\mathscr{T}_\top\left(b_{basetype} \right)}{U_\top} \right) \right\} \right\}} \tag{C.42}$$

$$\frac{\left\{ U_\top \;\vdash \left\{ t_{types} \right\} \Rightarrow \left\{ \mathrm{L}_\perp \right\} \right\}}{\left\{ U_\top \;\vdash \left\{ \texttt{typeTupletype}\left(t_{types}\right) \right\} \Rightarrow \left\{ \mathfrak{T}_{\top\perp}\left(\mathrm{T}_\perp \right) \right\} \right\}} \tag{C.43}$$

$$\frac{\left\{ U_\top \;\vdash \left\{ t_{types} \right\} \Rightarrow \left\{ \frac{L_\top}{U_\top} \right\} \right\}}{\left\{ U_\top \;\vdash \left\{ \texttt{typeTupletype}\left(t_{types}\right) \right\} \Rightarrow \left\{ \mathscr{T}_\top\left(\frac{L_\top}{U_\top} \right) \right\} \right\}} \tag{C.44}$$

$$\frac{\left\{ U_\top \;\vdash \left\{ a_{assoctypes} \right\} \Rightarrow \left\{ \mathrm{A}_\perp \right\} \right\}}{\left\{ U_\top \;\vdash \left\{ \texttt{typeRecordtype}\left(a_{assoctypes}\right) \right\} \Rightarrow \left\{ \mathfrak{T}_{\top\perp}\left(\mathrm{T}_\perp \right) \right\} \right\}} \tag{C.45}$$

$$\frac{\left\{ U_\top \;\vdash \left\{ a_{assoctypes} \right\} \Rightarrow \left\{ \frac{A_\top}{U_\top} \right\} \right\}}{\left\{ U_\top \;\vdash \left\{ \texttt{typeRecordtype}\left(t_{types}\right) \right\} \Rightarrow \left\{ \mathscr{T}_\top\left(\frac{L_\top}{U_\top} \right) \right\} \right\}} \tag{C.46}$$

$$\frac{U_\top\left(t_{typeid} \right) = \mathrm{T}_\perp}{\left\{ U_\top \;\vdash \left\{ \texttt{typeTypeid}\left(t_{typeid}\right) \right\} \Rightarrow \left\{ \mathfrak{T}_{\top\perp}\left(\mathrm{T}_\perp \right) \right\} \right\}} \tag{C.47}$$

$$\frac{U_\top\left(t_{typeid} \right) = \frac{T_\top}{U_\top}}{\left\{ U_\top \;\vdash \left\{ \texttt{typeTypeid}\left(t_{typeid}\right) \right\} \Rightarrow \left\{ \mathfrak{T}_{\top\perp}\left(\frac{T_\top}{U_\top} \right) \right\} \right\}} \tag{C.48}$$

Matches on the *types* syntactic object:

$$\frac{\left\{ U_{\top} \vdash \left\{ t_{types} \right\} \Rightarrow \left\{ \mathrm{L}_{\bot} \right\} \right\}}{\left\{ U_{\top} \vdash \left\{ \texttt{typesInd}\left(t_{types},\ t'_{type}\right) \right\} \Rightarrow \left\{ \mathfrak{L}_{\top\bot}\left(\mathrm{L}_{\bot} \right) \right\} \right\}} \tag{C.49}$$

$$\frac{\left\{ U_{\top} \vdash \left\{ t_{types} \right\} \Rightarrow \left\{ \frac{L_{\top}}{U_{\top}} \right\} \right\} \quad \left\{ U_{\top} \vdash \left\{ t'_{type} \right\} \Rightarrow \left\{ \mathrm{T}_{\bot} \right\} \right\}}{\left\{ U_{\top} \vdash \left\{ \texttt{typesInd}\left(t_{types},\ t'_{type}\right) \right\} \Rightarrow \left\{ \mathfrak{L}_{\top\bot}\left(\mathrm{L}_{\bot} \right) \right\} \right\}} \tag{C.50}$$

$$\frac{\left\{ U_{\top} \vdash \left\{ t_{types} \right\} \Rightarrow \left\{ \frac{L_{\top}}{U_{\top}} \right\} \right\} \quad \left\{ U_{\top} \vdash \left\{ t'_{type} \right\} \Rightarrow \left\{ \frac{T_{\top}}{U_{\top}} \right\} \right\}}{\left\{ U_{\top} \vdash \left\{ \texttt{typesInd}\left(t_{types},\ t'_{type}\right) \right\} \Rightarrow \left\{ \mathfrak{L}_{\top\bot}\left(\mathscr{L}_{\top}\left(\frac{L_{\top}}{U_{\top}},\ \frac{T_{\top}}{U_{\top}} \right) \right) \right\} \right\}} \tag{C.51}$$

$$\frac{}{\left\{ U_{\top} \vdash \left\{ \texttt{typesBase()} \right\} \Rightarrow \left\{ \mathfrak{L}_{\top\bot}\left(\mathscr{L}_{\top}\left(\frac{T_{\top}}{U_{\top}} \right) \right) \right\} \right\}} \tag{C.52}$$

Matches on the *assoctypes* syntactic object:

$$\frac{\left\{ U_{\top} \vdash \left\{ a_{assoctypes} \right\} \Rightarrow \left\{ \mathrm{R}_{\mathrm{D}\bot} \right\} \right\}}{\left\{ U_{\top} \vdash \left\{ \texttt{assoctypesInd}\left(a_{assoctypes},\ v_{vardeclprim}\right) \right\} \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\mathrm{R}_{\mathrm{D}\bot} \right) \right\} \right\}} \tag{C.53}$$

$$\frac{\left\{ U_{\top} \vdash \left\{ a_{assoctypes} \right\} \Rightarrow \left\{ \frac{R_{\mathrm{D}\top}}{U_{\top}} \right\} \right\} \quad \left\{ U_{\top},\ \frac{R_{\mathrm{D}\top}}{U_{\top}} \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ \mathrm{R}_{\mathrm{D}\bot} \right\} \right\}}{\left\{ U_{\top} \vdash \left\{ \texttt{assoctypesInd}\left(a_{assoctypes},\ v_{vardeclprim}\right) \right\} \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\mathrm{R}_{\mathrm{D}\bot} \right) \right\} \right\}} \tag{C.54}$$

$$\frac{\left\{ U_{\top} \vdash \left\{ a_{assoctypes} \right\} \Rightarrow \left\{ \frac{R_{\mathrm{D}\top}}{U_{\top}} \right\} \right\} \quad \left\{ U_{\top},\ \frac{R_{\mathrm{D}\top}}{U_{\top}} \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ \frac{R'_{\mathrm{D}\top}}{U_{\top}} \right\} \right\}}{\left\{ U_{\top} \vdash \left\{ \texttt{assoctypesInd}\left(a_{assoctypes},\ v_{vardeclprim}\right) \right\} \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\frac{R'_{\mathrm{D}\top}}{U_{\top}} \right) \right\} \right\}} \tag{C.55}$$

$$\frac{}{\left\{ U_{\top} \vdash \left\{ \texttt{assoctypesBase()} \right\} \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\frac{\mathscr{R}_{\mathrm{D}\top\varnothing}}{U_{\top}} \right) \right\} \right\}} \tag{C.56}$$

Matches on the *vardeclprim* syntactic object:

$$\frac{\frac{R_{\mathrm{D}\top}}{U_{\top}}\left(v_{varid} \right) \neq \mathrm{T}_{\bot}}{\left\{ U_{\top},\ \frac{R_{\mathrm{D}\top}}{U_{\top}} \vdash \left\{ \texttt{vardeclprim}\left(v_{varid},\ t_{type}\right) \right\} \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\mathrm{R}_{\mathrm{D}\bot} \right) \right\} \right\}} \tag{C.57}$$

$$\frac{\left\{ U_{\top} \vdash \left\{ t_{type} \right\} \Rightarrow \left\{ \mathrm{T}_{\bot} \right\} \right\}}{\left\{ U_{\top},\ \frac{R_{\mathrm{D}\top}}{U_{\top}} \vdash \left\{ \texttt{vardeclprim}\left(v_{varid},\ t_{type}\right) \right\} \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\mathrm{R}_{\mathrm{D}\bot} \right) \right\} \right\}} \tag{C.58}$$

$$\frac{\frac{R_{\mathrm{D}\top}}{U_{\top}}\left(v_{varid} \right) = \mathrm{T}_{\bot} \quad \left\{ U_{\top} \vdash \left\{ t_{type} \right\} \Rightarrow \left\{ T_{\top} \right\} \right\}}{\left\{ \begin{array}{l} U_{\top},\ \frac{R_{\mathrm{D}\top}}{U_{\top}} \vdash \left\{ \texttt{vardeclprim}\left(v_{varid},\ t_{type}\right) \right\} \\ \Rightarrow \left\{ \Re_{\mathrm{D}\top\bot}\left(\begin{array}{l} \frac{R_{\mathrm{D}\top}}{U_{\top}} \oplus \left(v_{varid} \rightarrow \frac{T_{\top}}{U_{\top}} \right), \\ \frac{R_{\mathrm{D}\top}}{U_{\top}}\left(v_{varid} \right) = \mathrm{T}_{\bot} \end{array} \right) \right\} \end{array} \right\}} \tag{C.59}$$

C.2.2 Untimed expression fragment

Matches on the *uprogram* syntactic object:

$$\frac{\left\{ E_{\top\top}, \frac{\mathrm{P}_{\top\varnothing}}{E_{\top\top}} \vdash \left\{ d_{udecls} \right\} \Rightarrow \left\{\mathrm{P}_{\bot}\right\}\right\}}{\left\{ E_{\top\top}, \frac{\mathrm{P}_{\top\varnothing}}{E_{\top\top}} \vdash \left\{ \mathtt{uProgDecls}(d_{udecls}) \right\} \Rightarrow \left\{\mathfrak{P}_{\top\mathsf{def}\bot}\left(\mathrm{P}_{\bot} \right)\right\}\right\}} \tag{C.60}$$

$$\frac{\left\{ E_{\top\top}, \frac{\mathrm{P}_{\top\varnothing}}{E_{\top\top}} \vdash \left\{ d_{udecls} \right\} \Rightarrow \left\{P'_{\top}\right\}\right\} \quad \mathsf{decldef}\left(P'_{\top} \right) = \bot}{\left\{ E_{\top\top}, \frac{\mathrm{P}_{\top\varnothing}}{E_{\top\top}} \vdash \left\{ \mathtt{uProgDecls}(d_{udecls}) \right\} \Rightarrow \left\{\mathfrak{P}_{\top\mathsf{def}\bot}\left(\mathrm{P}_{\bot} \right)\right\}\right\}} \tag{C.61}$$

$$\frac{\left\{ E_{\top\top}, \frac{\mathrm{P}_{\top\varnothing}}{E_{\top\top}} \vdash \left\{ d_{udecls} \right\} \Rightarrow \left\{\frac{P'_{\top}}{E_{\top\top}}\right\}\right\} \quad \mathsf{decldef}\left(P'_{\top} \right) = \top}{\left\{ \begin{array}{l} E_{\top\top}, \frac{\mathrm{P}_{\top\varnothing}}{E_{\top\top}} \vdash \left\{ \mathtt{uProgDecls}(d_{udecls}) \right\} \\ \Rightarrow \left\{\mathfrak{P}_{\top\mathsf{def}\bot}\left(\mathscr{P}_{\top\mathsf{def}}\left(\begin{array}{l} \frac{P'_{\top}}{E_{\top\top}}, \\ \mathsf{decldef}_{\mathsf{Prop}}\left(\frac{P'_{\top}}{E_{\top\top}} \right) \end{array} \right)\right)\right\} \end{array} \right\}} \tag{C.62}$$

Matches on the *udecls* syntactic object:

$$\frac{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ d_{udecls} \right\} \Rightarrow \left\{\mathrm{P}_{\bot}\right\}\right\}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \mathtt{udeclsInd}\left(d_{udecls}, d'_{udecl}\right) \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right)\right\}\right\}} \tag{C.63}$$

$$\frac{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ d_{udecls} \right\} \Rightarrow \left\{\frac{P'_{\top}}{E_{\top\top}}\right\}\right\} \quad \left\{ E_{\top\top}, \frac{P'_{\top}}{E_{\top\top}} \vdash \left\{ d'_{udecl} \right\} \Rightarrow \left\{\mathrm{P}_{\bot}\right\}\right\}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \mathtt{udeclsInd}\left(d_{udecls}, d'_{udecl}\right) \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right)\right\}\right\}} \tag{C.64}$$

$$\frac{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ d_{udecls} \right\} \Rightarrow \left\{\frac{P'_{\top}}{E_{\top\top}}\right\}\right\} \quad \left\{ E_{\top\top}, \frac{P'_{\top}}{E_{\top\top}} \vdash \left\{ d'_{udecl} \right\} \Rightarrow \left\{\frac{P''_{\top}}{E_{\top\top}}\right\}\right\}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \mathtt{udeclsInd}\left(d_{udecls}, d'_{udecl}\right) \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\frac{P''_{\top}}{E_{\top\top}} \right)\right\}\right\}} \tag{C.65}$$

$$\frac{}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \mathtt{udeclsBase}() \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\frac{P_{\top}}{E_{\top\top}} \right)\right\}\right\}} \tag{C.66}$$

Matches on the *udecl* syntactic object:

$$\frac{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ d_{vardecl} \right\} \Rightarrow \left\{\mathrm{P}_{\bot}\right\}\right\}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \mathtt{declVardecl}(d_{vardecl}) \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right)\right\}\right\}} \tag{C.67}$$

$$\frac{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ d_{vardecl}\ \right\} \Rightarrow \left\{\frac{P'_\top}{E_{\top\top}}\right\}\right\}}{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ \texttt{declVardecl}(d_{vardecl})\ \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\ P'_\top\ \right)\right\}\right\}} \tag{C.68}$$

$$\frac{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ d_{vardef}\ \right\} \Rightarrow \left\{\mathrm{P}_\bot\right\}\right\}}{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ \texttt{declVardef}\left(d_{vardef}\right)\ \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\ \mathrm{P}_\bot\ \right)\right\}\right\}} \tag{C.69}$$

$$\frac{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ d_{vardef}\ \right\} \Rightarrow \left\{\frac{P'_\top}{E_{\top\top}}\right\}\right\}}{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ \texttt{declVardef}\left(d_{vardef}\right)\ \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\ P'_\top\ \right)\right\}\right\}} \tag{C.70}$$

Matches on the *vardecl* syntactic object:

$$\frac{\left\{ E_{\top\top}.U_{\top\top},\ \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\ \vdash \left\{\ v_{vardeclprim}\ \right\} \Rightarrow \left\{\mathrm{R}_\bot\right\}\right\}}{\left\{ E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ \texttt{vardeclVar}\left(v_{vardeclprim}\right)\ \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\ \mathrm{P}_\bot\ \right)\right\}\right\}} \tag{C.71}$$

$$\frac{\begin{array}{l}\left\{ E_{\top\top}.U_{\top\top},\ \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\ \vdash \left\{\ v_{vardeclprim}\ \right\} \Rightarrow \left\{\frac{R'_\top}{E_{\top\top}.U_{\top\top}}\right\}\right\}\\ \frac{R'_\top}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\end{array}}{\left\{\begin{array}{l} E_{\top\top},\ \frac{P_\top}{E_{\top\top}}\ \vdash \left\{\ \texttt{vardeclVar}\left(v_{vardeclprim}\right)\ \right\}\\ \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\begin{array}{c} P_\top +_W \frac{R'_\top}{E_{\top\top}.U_{\top\top}},\\ \frac{R'_\top}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\end{array}\right)\right\}\end{array}\right\}} \tag{C.72}$$

$$\frac{\left\{ E_{\top\top}.U_{\top\top},\ \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\ \vdash \left\{\ v_{fundeclprim}\ \right\} \Rightarrow \left\{\mathrm{R}_\bot\right\}\right\}}{\left\{ E_{\top\top},\ \frac{E_\top}{P_\top,E_{\top\top}}\ \vdash \left\{\ \texttt{vardeclFun}\left(v_{fundeclprim}\right)\ \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\ \mathrm{P}_\bot\ \right)\right\}\right\}} \tag{C.73}$$

$$\frac{\begin{array}{l}\left\{ E_{\top\top}.U_{\top\top},\ \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\ \vdash \left\{\ v_{fundeclprim}\ \right\} \Rightarrow \left\{\frac{R'_\top}{E_{\top\top}.U_{\top\top}}\right\}\right\}\\ \frac{R'_\top}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\end{array}}{\left\{ E_{\top\top},\ \frac{E_\top}{P_\top,E_{\top\top}}\ \vdash \left\{\ \texttt{vardeclFun}\left(v_{fundeclprim}\right)\ \right\} \Rightarrow \left\{\mathfrak{P}_{\top\bot}\left(\begin{array}{c} P_\top +_W \frac{R'_\top}{E_{\top\top}.U_{\top\top}},\\ \frac{R'_\top}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}\end{array}\right)\right\}\right\}} \tag{C.74}$$

Matches on the *vardeclprim* syntactic object:

$$\frac{\frac{R_\top}{U_\top}\left(\ v_{varid}\ \right) \not\models \mathrm{Y}_\bot \quad \frac{R_\top}{U_\top}\left(\ v_{varid}\ \right) \not\models \frac{\mathscr{Y}_\top}{U_\top}\left(\ t_{type}\ \right)}{\left\{ U_\top,\ \frac{R_\top}{U_\top}\ \vdash \left\{\ \texttt{vardeclprim}\left(v_{varid},\ t_{type}\right)\ \right\} \Rightarrow \left\{\mathfrak{R}_{\top\bot}\left(\ \mathrm{R}_\bot\ \right)\right\}\right\}} \tag{C.75}$$

$$\frac{\left\{ U_{\top} \vdash \left\{ t_{type} \right\} \Rightarrow \left\{ \mathrm{T}_{\bot} \right\} \right\}}{\left\{ U_{\top}, \frac{R_{\top}}{U_{\top}} \vdash \left\{ \mathtt{vardeclprim}\left(v_{varid}, t_{type} \right) \right\} \Rightarrow \left\{ \Re_{\top\bot}\left(\mathrm{R}_{\bot} \right) \right\} \right\}} \tag{C.76}$$

$$\frac{\frac{R_{\top}}{U_{\top}}\left(v_{varid} \right) = \frac{\mathscr{Y}_{\top}}{U_{\top}}\left(t_{type} \right)}{\left\{ U_{\top}, \frac{R_{\top}}{U_{\top}} \vdash \left\{ \mathtt{vardeclprim}\left(v_{varid}, t_{type} \right) \right\} \Rightarrow \left\{ \Re_{\top\bot}\left(\frac{R_{\top}}{U_{\top}} \right) \right\} \right\}} \tag{C.77}$$

$$\frac{\frac{R_{\top}}{U_{\top}}\left(v_{varid} \right) = \mathrm{Y}_{\bot} \quad \left\{ U_{\top} \vdash \left\{ t_{type} \right\} \Rightarrow \left\{ \frac{T_{\top}}{U_{\top}} \right\} \right\}}{\left\{ \begin{array}{l} U_{\top}, \frac{R_{\top}}{U_{\top}} \vdash \left\{ \mathtt{vardeclprim}\left(v_{varid}, t_{type} \right) \right\} \\ \Rightarrow \left\{ \Re_{\top\bot}\left(\begin{array}{l} \frac{R_{\top}}{U_{\top}} \oplus \left(v_{varid} \rightarrow \frac{\mathscr{Y}_{\top}}{U_{\top}}\left(\frac{T_{\top}}{U_{\top}} \right) \right), \\ \frac{R_{\top}}{U_{\top}}\left(v_{varid} \right) = \mathrm{Y}_{\bot} \end{array} \right) \right\} \end{array} \right\}} \tag{C.78}$$

$$\frac{}{\left\{ \vdash \left\{ \mathtt{vardeclprim}\left(v_{varid}, t_{type} \right) \right\} \Rightarrow \left\{ v_{varid} \right\} \right\}} \tag{C.79}$$

Matches on the *vardef* syntactic object:

$$\frac{\left\{ E_{\mathrm{T}\top}.U_{\mathrm{T}\top}, \frac{P_{\mathrm{T}\top}.R_{\mathrm{T}\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ \mathrm{R}_{\bot} \right\} \right\}}{\left\{ E_{\mathrm{T}\top}, \frac{P_{\top}}{E_{\mathrm{T}\top}} \vdash \left\{ \mathtt{vardefVar}\left(v_{vardeclprim}, e_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right) \right\} \right\}} \tag{C.80}$$

$$\frac{\begin{array}{l} \left\{ E_{\mathrm{T}\top}.U_{\mathrm{T}\top}, \frac{P_{\mathrm{T}\top}.R_{\mathrm{T}\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ \frac{R'_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \right\} \right\} \\ \left\{ \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ v_{varid} \right\} \right\} \\ \left\{ E_{\mathrm{T}\top}.U_{\mathrm{T}\top}, \frac{\mathscr{Y}_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}}\left(\frac{R'_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}}\left(v_{varid} \right) \right), \frac{R'_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \mathrm{K}_{\bot} \right\} \right\} \end{array}}{\left\{ E_{\mathrm{T}\top}, \frac{P_{\top}}{E_{\mathrm{T}\top}} \vdash \left\{ \mathtt{vardefVar}\left(v_{vardeclprim}, e_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right) \right\} \right\}} \tag{C.81}$$

$$\frac{\begin{array}{l} \left\{ E_{\mathrm{T}\top}.U_{\mathrm{T}\top}, \frac{P_{\mathrm{T}\top}.R_{\mathrm{T}\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ \frac{R'_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \right\} \right\} \\ \left\{ \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ v_{varid} \right\} \right\} \\ \left\{ E_{\mathrm{T}\top}.U_{\mathrm{T}\top}, \frac{\mathscr{Y}_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}}\left(\frac{R'_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}}\left(v_{varid} \right) \right), \frac{R'_{\top}}{E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \frac{K_{\top}}{R'_{\top}, Y_{\top}, E_{\mathrm{T}\top}.U_{\mathrm{T}\top}} \right\} \right\} \\ \frac{P_{\mathrm{T}\top}.W_{\mathrm{T}\top}}{P_{\mathrm{T}\top}.R_{\mathrm{T}\top}, E_{\mathrm{T}\top}.U_{\mathrm{T}\top}}\left(v_{varid} \right) \not\models \mathrm{Z}_{\bot} \end{array}}{\left\{ \begin{array}{l} E_{\mathrm{T}\top}, \frac{P_{\top}}{E_{\mathrm{T}\top}} \vdash \left\{ \mathtt{vardefVar}\left(v_{vardeclprim}, e_{expr} \right) \right\} \\ \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right) \right\} \end{array} \right\}} \tag{C.82}$$

$$\frac{\begin{array}{l}\left\{ E_{\top\top}.U_{\top\top}, \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}} \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \left\{ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \right\} \right\} \\ \left\{ \vdash \left\{ v_{vardeclprim} \right\} \Rightarrow \{ v_{varid} \} \right\} \\ \left\{ E_{\top\top}.U_{\top\top}, \frac{\mathscr{Y}_{\top}}{E_{\top\top}.U_{\top\top}}\left(\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}}\left(v_{varid} \right) \right), \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \frac{K_{\top}}{R'_{\top}, Y_{\top}, E_{\top\top}.U_{\top\top}} \right\} \right\} \\ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}} \\ \frac{P_{\top\top}.W_{\top\top}}{P_{\top\top}.R_{\top\top}, E_{\top\top}.U_{\top\top}}\left(v_{varid} \right) = \mathrm{Z}_{\bot}\end{array}}{\left\{ \begin{array}{l} E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \texttt{vardefVar}\left(v_{vardeclprim}, e_{expr} \right) \right\} \\ \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\begin{array}{l} P_{\top} +_{W} \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \mathscr{Z}_{\top}\left(\frac{K_{\top}}{R'_{\top}, Y_{\top}, E_{\top\top}.U_{\top\top}} \right), , \\ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}}, \frac{P_{\top\top}.W_{\top\top}}{P_{\top\top}.R_{\top\top}, E_{\top\top}.U_{\top\top}}\left(v_{varid} \right) = \mathrm{Z}_{\bot} \end{array} \right) \right\} \end{array} \right\}} \quad \text{(C.83)}$$

$$\frac{\left\{ E_{\top\top}.U_{\top\top}, \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}} \vdash \left\{ f_{fundeclprim} \right\} \Rightarrow \left\{ \mathrm{R}_{\bot} \right\} \right\}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \texttt{vardefFun}\left(v_{fundeclprim}, e_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right) \right\} \right\}} \quad \text{(C.84)}$$

$$\frac{\begin{array}{l}\left\{ E_{\top\top}.U_{\top\top}, \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}} \vdash \left\{ f_{fundeclprim} \right\} \Rightarrow \left\{ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \right\} \right\} \\ \left\{ \vdash \left\{ v_{fundeclprim} \right\} \Rightarrow \{ v_{varid} \} \right\} \\ \left\{ \begin{array}{l} E_{\top\top}.U_{\top\top}, \frac{\mathscr{Y}_{\top}}{E_{\top\top}.U_{\top\top}}\left(\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}}\left(v_{varid} \right) \right), \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \left(v_{varid} \rightarrow \frac{Y_{\top}}{E_{\top\top}.U_{\top\top}} \right) \\ \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \mathrm{K}_{\bot} \right\} \end{array} \right\}\end{array}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \texttt{vardefFun}\left(v_{vardeclprim}, e_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right) \right\} \right\}} \quad \text{(C.85)}$$

$$\frac{\begin{array}{l}\left\{ E_{\top\top}.U_{\top\top}, \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}} \vdash \left\{ f_{fundeclprim} \right\} \Rightarrow \left\{ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \right\} \right\} \\ \left\{ \vdash \left\{ v_{fundeclprim} \right\} \Rightarrow \{ v_{varid} \} \right\} \\ \left\{ \begin{array}{l} E_{\top\top}.U_{\top\top}, \frac{\mathscr{Y}_{\top}}{E_{\top\top}.U_{\top\top}}\left(\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}}\left(v_{varid} \right) \right), \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \left(v_{varid} \rightarrow \frac{Y_{\top}}{E_{\top\top}.U_{\top\top}} \right) \\ \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \frac{K_{\top}}{\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \left(v_{varid} \rightarrow \frac{Y_{\top}}{E_{\top\top}.U_{\top\top}} \right), Y_{\top}, E_{\top\top}.U_{\top\top}} \right\} \end{array} \right\} \\ \frac{P_{\top\top}.W_{\top\top}}{P_{\top\top}.R_{\top\top}, E_{\top\top}.U_{\top\top}}\left(v_{varid} \right) \not\models \mathrm{Z}_{\bot}\end{array}}{\left\{ E_{\top\top}, \frac{P_{\top}}{E_{\top\top}} \vdash \left\{ \texttt{vardefFun}\left(v_{vardeclprim}, e_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{P}_{\top\bot}\left(\mathrm{P}_{\bot} \right) \right\} \right\}} \quad \text{(C.86)}$$

$$\frac{\begin{array}{l}\left\{ E_{\top\top}.U_{\top\top},\ \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}} \ \vdash \left\{\ f_{fundeclprim}\ \right\} \Rightarrow \left\{ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \right\} \right\} \\ \left\{ \vdash \left\{\ v_{fundeclprim}\ \right\} \Rightarrow \{v_{varid}\} \right\} \\ \left\{ \begin{array}{l} E_{\top\top}.U_{\top\top},\ \frac{\mathscr{Y}_{\top}}{E_{\top\top}.U_{\top\top}}\left(\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}}\left(\ v_{varid}\ \right) \right),\ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \left(\ v_{varid} \rightarrow \frac{Y_{\top}}{E_{\top\top}.U_{\top\top}}\ \right) \\ \vdash \left\{\ e_{expr}\ \right\} \Rightarrow \left\{ \frac{K_{\top}}{\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \left(\ v_{varid} \rightarrow \frac{Y_{\top}}{E_{\top\top}.U_{\top\top}}\ \right), Y_{\top}, E_{\top\top}.U_{\top\top}} \right\} \end{array} \right\} \\ \frac{P_{\top\top}.W_{\top\top}}{P_{\top\top}.R_{\top\top}, E_{\top\top}.U_{\top\top}}\left(\ v_{varid}\ \right) = \mathrm{Z}_{\perp} \end{array}}{\left\{ \begin{array}{l} E_{\top\top},\ \frac{P_{\top}}{E_{\top\top}} \ \vdash \left\{\ \mathtt{vardefFun}\left(v_{vardeclprim},\ e_{expr}\right)\ \right\} \\ \Rightarrow \left\{ \mathfrak{P}_{\top\perp}\left(\begin{array}{l} P_{\top} +_{W} \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \mathscr{Z}_{\top}\left(v_{varid},\ \frac{K_{\top}}{\frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} + \left(\ v_{varid} \rightarrow \frac{Y_{\top}}{E_{\top\top}.U_{\top\top}}\ \right), Y_{\top}, E_{\top\top}.U_{\top\top}} \right),, \\ \frac{R'_{\top}}{E_{\top\top}.U_{\top\top}} \geq \frac{P_{\top\top}.R_{\top\top}}{E_{\top\top}.U_{\top\top}},\ \frac{P_{\top\top}.W_{\top\top}}{P_{\top\top}.R_{\top\top}, E_{\top\top}.U_{\top\top}}\left(\ v_{varid}\ \right) = \mathrm{Z}_{\perp} \end{array} \right) \right\} \end{array} \right\}} \tag{C.87}$$

Matches on the *fundeclprim* syntactic object:

$$\frac{\frac{R_{\top}}{U_{\top}}\left(\ v_{varid}\ \right) \not\models \mathrm{Y}_{\perp} \quad \frac{R_{\top}}{U_{\top}}\left(\ v_{varid}\ \right) \not\models \frac{\mathscr{Y}_{\top}}{U_{\top}}\left(\ t_{type},\ t'_{type}\ \right)}{\left\{ U_{\top},\ \frac{R_{\top}}{U_{\top}} \ \vdash \left\{\ \mathtt{fundeclprim}\left(v_{varid},\ t_{type},\ t'_{type}\right)\ \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\perp}\left(\ \mathrm{R}_{\perp}\ \right) \right\} \right\}} \tag{C.88}$$

$$\frac{\left\{\ U_{\top} \ \vdash \left\{\ t_{type}\ \right\} \Rightarrow \left\{ \mathrm{T}_{\perp} \right\} \right\}}{\left\{ U_{\top},\ \frac{R_{\top}}{U_{\top}} \ \vdash \left\{\ \mathtt{fundeclprim}\left(v_{varid},\ t_{type},\ t'_{type}\right)\ \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\perp}\left(\ \mathrm{R}_{\perp}\ \right) \right\} \right\}} \tag{C.89}$$

$$\frac{\left\{\ U_{\top} \ \vdash \left\{\ t'_{type}\ \right\} \Rightarrow \left\{ \mathrm{T}_{\perp} \right\} \right\}}{\left\{ U_{\top},\ \frac{R_{\top}}{U_{\top}} \ \vdash \left\{\ \mathtt{fundeclprim}\left(v_{varid},\ t_{type},\ t'_{type}\right)\ \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\perp}\left(\ \mathrm{R}_{\perp}\ \right) \right\} \right\}} \tag{C.90}$$

$$\frac{\frac{R_{\top}}{U_{\top}}\left(\ v_{varid}\ \right) = \frac{\mathscr{Y}_{\top}}{U_{\top}}\left(\ t_{type},\ t'_{type}\ \right)}{\left\{ U_{\top},\ \frac{R_{\top}}{U_{\top}} \ \vdash \left\{\ \mathtt{fundeclprim}\left(v_{varid},\ t_{type},\ t'_{type}\right)\ \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\perp}\left(\ \frac{R_{\top}}{U_{\top}}\ \right) \right\} \right\}} \tag{C.91}$$

$$\frac{\frac{R_{\top}}{U_{\top}}\left(\ v_{varid}\ \right) = \mathrm{Y}_{\perp} \quad \left\{\ U_{\top} \ \vdash \left\{\ t_{type}\ \right\} \Rightarrow \left\{ \frac{T_{\top}}{U_{\top}} \right\} \right\} \quad \left\{\ U_{\top} \ \vdash \left\{\ t'_{type}\ \right\} \Rightarrow \left\{ \frac{T'_{\top}}{U_{\top}} \right\} \right\}}{\left\{ \begin{array}{l} U_{\top},\ \frac{R_{\top}}{U_{\top}} \ \vdash \left\{\ \mathtt{fundeclprim}\left(v_{varid},\ t_{type}\right)\ \right\} \\ \Rightarrow \left\{ \mathfrak{R}_{\top\perp}\left(\begin{array}{l} \frac{R_{\top}}{U_{\top}} \oplus \left(\ v_{varid} \rightarrow \frac{\mathscr{Y}_{\top}}{U_{\top}}\left(\frac{T_{\top}}{U_{\top}},\ \frac{T'_{\top}}{U_{\top}} \right) \right), \\ \frac{R_{\top}}{U_{\top}}\left(\ v_{varid}\ \right) = \mathrm{Y}_{\perp} \end{array} \right) \right\} \end{array} \right\}} \tag{C.92}$$

$$\frac{}{\left\{ \vdash \left\{\ \mathtt{fundeclprim}\left(v_{varid},\ t_{type},\ t'_{type}\right)\ \right\} \Rightarrow \{v_{varid}\} \right\}} \tag{C.93}$$

Matches on the *expr* syntactic object:

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ p_{patt} \right\} \Rightarrow \left\{ \mathrm{Q}_\bot \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprPatt}\left(p_{patt}\right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\mathrm{K}_\bot \right) \right\} \right\}} \tag{C.94}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ p_{patt} \right\} \Rightarrow \left\{ \frac{\mathrm{Q}_\top}{R_\top, Y_\top, U_\top} \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprPatt}\left(p_{patt}\right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\frac{\mathrm{Q}_\top}{R_\top, Y_\top, U_\top} \right) \right\} \right\}} \tag{C.95}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ c_{constr} \right\} \Rightarrow \left\{ \mathrm{C}_\bot \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprConstr}(c_{constr}) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\mathrm{K}_\bot \right) \right\} \right\}} \tag{C.96}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ c_{constr} \right\} \Rightarrow \left\{ \frac{C_\top}{R_\top, Y_\top, U_\top} \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprConstr}(c_{constr}) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\frac{C_\top}{R_\top, Y_\top, U_\top} \right) \right\} \right\}} \tag{C.97}$$

$$\frac{\frac{R_\top}{U_\top}\left(v_{varid} \right) = Y_\bot}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprFunapp}\left(v_{varid}, e_{expr}\right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\mathrm{K}_\bot \right) \right\} \right\}} \tag{C.98}$$

$$\frac{\frac{R_\top}{U_\top}\left(v_{varid} \right) \not\models Y_\bot \quad \left\{ U_\top, \frac{R_\top}{U_\top} \vdash \left\{ v_{varid} \right\} \Rightarrow \left\{ \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top} \right) \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprFunapp}\left(v_{varid}, e_{expr}\right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\mathrm{K}_\bot \right) \right\} \right\}} \tag{C.99}$$

$$\frac{\begin{array}{l} \frac{R_\top}{U_\top}\left(v_{varid} \right) \not\models Y_\bot \\ \left\{ U_\top, \frac{R_\top}{U_\top} \vdash \left\{ v_{varid} \right\} \Rightarrow \left\{ \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top}, \frac{T_\top^{\prime}}{U_\top} \right) \right\} \right\} \\ \frac{Y_\top}{U_\top} \not\models \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top^{\prime}}{U_\top} \right) \end{array}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprFunapp}\left(v_{varid}, e_{expr}\right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\mathrm{K}_\bot \right) \right\} \right\}} \tag{C.100}$$

$$\frac{\begin{array}{l} \frac{R_\top}{U_\top}\left(v_{varid} \right) \not\models Y_\bot \\ \left\{ U_\top, \frac{R_\top}{U_\top} \vdash \left\{ v_{varid} \right\} \Rightarrow \left\{ \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top}, \frac{T_\top^{\prime}}{U_\top} \right) \right\} \right\} \\ \frac{Y_\top}{U_\top} = \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top^{\prime}}{U_\top} \right) \\ \left\{ U_\top, \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top} \right), \frac{R_\top}{U_\top} \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \mathrm{K}_\bot \right\} \right\} \end{array}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprFunapp}\left(v_{varid}, e_{expr}\right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\top\bot}\left(\mathrm{K}_\bot \right) \right\} \right\}} \tag{C.101}$$

$$\frac{\begin{array}{l}\frac{R_\top}{U_\top}\left(v_{varid} \right)\not\models Y_\bot \\ \left\{ U_\top, \frac{R_\top}{U_\top} \vdash \left\{ v_{varid} \right\} \Rightarrow \left\{ \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top}, \frac{T_\top'}{U_\top} \right)\right\}\right\} \\ \frac{Y_\top}{U_\top} = \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top'}{U_\top} \right) \\ \left\{ U_\top, \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top} \right), \frac{R_\top}{U_\top} \vdash \left\{ e_{expr} \right\} \Rightarrow \left\{ \frac{K_\top}{R_\top, \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top} \right), U_\top} \right\}\right\}\end{array}}{\left\{\begin{array}{l} U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprFunapp}\left(v_{varid}, e_{expr}\right) \right\} \\ \Rightarrow \left\{ \mathfrak{K}_{\top\bot}\left(\mathscr{K}_\top\left(v_{varid}, U_\top, \frac{T_\top}{U_\top}, \frac{K_\top}{R_\top, \frac{\mathscr{Y}_\top}{U_\top}\left(\frac{T_\top}{U_\top} \right), U_\top} \right)\right)\right\}\end{array}\right\}} \tag{C.102}$$

Matches on the *constr* syntactic object:

$$\frac{}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{constrBase}(b_{boolconst}) \right\} \Rightarrow \left\{ \mathfrak{C}_{\top\bot}\left(\frac{\mathscr{C}_\top}{R_\top, Y_\top, U_\top}\left(b_{boolconst} \right)\right)\right\}\right\}} \tag{C.103}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprtup} \right\} \Rightarrow \left\{ \mathrm{K}_{\mathbf{tup}\bot} \right\}\right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{constrTup}\left(e_{exprtup}\right) \right\} \Rightarrow \left\{ \mathfrak{C}_{\top\bot}\left(\mathrm{C}_\bot \right)\right\}\right\}} \tag{C.104}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprtup} \right\} \Rightarrow \left\{ \frac{K_{\mathbf{tup}\top}}{R_\top, Y_\top, U_\top} \right\}\right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{constrTup}\left(e_{exprtup}\right) \right\} \Rightarrow \left\{ \mathfrak{C}_{\top\bot}\left(\frac{\mathscr{C}_\top}{R_\top, Y_\top, U_\top}\left(\frac{K_{\mathbf{tup}\top}}{R_\top, Y_\top, U_\top} \right)\right)\right\}\right\}} \tag{C.105}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprrec} \right\} \Rightarrow \left\{ \mathrm{K}_{\mathbf{rec}\bot} \right\}\right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{constrRec}\left(e_{exprrec}\right) \right\} \Rightarrow \left\{ \mathfrak{C}_{\top\bot}\left(\mathrm{C}_\bot \right)\right\}\right\}} \tag{C.106}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprrec} \right\} \Rightarrow \left\{ \frac{K_{\mathbf{rec}\top}}{R_\top, Y_\top, U_\top} \right\}\right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{constrRec}\left(e_{exprrec}\right) \right\} \Rightarrow \left\{ \mathfrak{C}_{\top\bot}\left(\frac{\mathscr{C}_\top}{R_\top, Y_\top, U_\top}\left(\frac{K_{\mathbf{rec}\top}}{R_\top, Y_\top, U_\top} \right)\right)\right\}\right\}} \tag{C.107}$$

Matches on the *exprtup* syntactic object:

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprtup} \right\} \Rightarrow \left\{ \mathrm{K}_{\mathbf{tup}\bot} \right\}\right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprtupInd}\left(e_{exprtup}, e_{expr}'\right) \right\} \Rightarrow \left\{ \mathfrak{K}_{\mathbf{tup}\top\bot}\left(\mathrm{K}_{\mathbf{tup}\bot} \right)\right\}\right\}} \tag{C.108}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprtup} \right\} \Rightarrow \left\{ \frac{K_{\mathrm{tup}\top}}{R_\top, Y_\top, U_\top} \right\} \right\} \quad \left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e'_{expr} \right\} \Rightarrow \left\{ \mathrm{K}_\perp \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprtupInd}\left(e_{exprtup}, e'_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\mathrm{tup}\top\perp}\left(\mathrm{K}_{\mathrm{tup}\perp} \right) \right\} \right\}} \tag{C.109}$$

$$\frac{\begin{array}{c} \left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprtup} \right\} \Rightarrow \left\{ \frac{K_{\mathrm{tup}\top}}{R_\top, Y_\top, U_\top} \right\} \right\} \\ \left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e'_{expr} \right\} \Rightarrow \left\{ \frac{K'_\top}{R_\top, Y_\top, U_\top} \right\} \right\} \end{array}}{\left\{ \begin{array}{l} U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprtupInd}\left(e_{exprtup}, e'_{expr} \right) \right\} \\ \Rightarrow \left\{ \mathfrak{R}_{\mathrm{tup}\top\perp}\left(\frac{K_{\mathrm{tup}\top}}{R_\top, Y_\top, U_\top}, \frac{K'_\top}{R_\top, Y_\top, U_\top} \right) \right\} \end{array} \right\}} \tag{C.110}$$

$$\frac{}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprtupBase()} \right\} \Rightarrow \left\{ \mathfrak{R}_{\mathrm{tup}\top\perp}\left(\mathscr{K}_{\mathrm{tup}\top}() \right) \right\} \right\}} \tag{C.111}$$

Matches on the *exprrec* syntactic object:

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprrec} \right\} \Rightarrow \left\{ \mathrm{K}_{\mathrm{rec}\perp} \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprrecInd}\left(e_{exprrec}, v_{varid}, e'_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\mathrm{rec}\top\perp}\left(\mathrm{K}_{\mathrm{rec}\perp} \right) \right\} \right\}} \tag{C.112}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprrec} \right\} \Rightarrow \left\{ \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top} \right\} \right\} \quad \left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e'_{expr} \right\} \Rightarrow \left\{ \mathrm{K}_\perp \right\} \right\}}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprrecInd}\left(e_{exprrec}, v_{varid}, e'_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\mathrm{rec}\top\perp}\left(\mathrm{K}_{\mathrm{rec}\perp} \right) \right\} \right\}} \tag{C.113}$$

$$\frac{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprrec} \right\} \Rightarrow \left\{ \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top} \right\} \right\} \quad \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top}\left(v_{varid} \right) \neq \mathrm{K}_\perp}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprrecInd}\left(e_{exprrec}, v_{varid}, e'_{expr} \right) \right\} \Rightarrow \left\{ \mathfrak{R}_{\mathrm{rec}\top\perp}\left(\mathrm{K}_{\mathrm{rec}\perp} \right) \right\} \right\}} \tag{C.114}$$

$$\frac{\begin{array}{c} \left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e_{exprrec} \right\} \Rightarrow \left\{ \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top} \right\} \right\} \\ \left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ e'_{expr} \right\} \Rightarrow \left\{ \frac{K'_\top}{R_\top, Y_\top, U_\top} \right\} \right\} \\ \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top}\left(v_{varid} \right) = \mathrm{K}_\perp \end{array}}{\left\{ \begin{array}{l} U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprrecInd}\left(e_{exprrec}, v_{varid}, e'_{expr} \right) \right\} \\ \Rightarrow \left\{ \mathfrak{R}_{\mathrm{rec}\top\perp}\left(\begin{array}{l} \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top} \oplus \left(v_{varid} \rightarrow \frac{K'_\top}{R_\top, Y_\top, U_\top} \right), \\ \frac{K_{\mathrm{rec}\top}}{R_\top, Y_\top, U_\top}\left(v_{varid} \right) = \mathrm{K}_\perp \end{array} \right) \right\} \end{array} \right\}} \tag{C.115}$$

$$\frac{}{\left\{ U_\top, \frac{Y_\top}{U_\top}, \frac{R_\top}{U_\top} \vdash \left\{ \texttt{exprrecBase()} \right\} \Rightarrow \left\{ \mathfrak{R}_{\mathrm{rec}\top\perp}\left(\mathscr{K}_{\mathrm{rec}\top}() \right) \right\} \right\}} \tag{C.116}$$

Matches on the *patt* syntactic object:

$$\frac{\frac{R_\top}{U_\top}\left(\ v_{varid}\ \right) = \mathrm{Y}_\bot}{\left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{R_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{pattIndex}(v_{varid},\ d_{datresolvelist})\ \right\} \Rightarrow \left\{\mathfrak{Q}_{\top\bot}\left(\ \mathrm{Q}_\bot\ \right)\right\}\right\}} \tag{C.117}$$

$$\frac{\frac{R_\top}{U_\top}\left(\ v_{varid}\ \right) = \frac{\mathscr{Y}_\top}{U_\top}\left(\ \frac{T_\top}{U_\top}\ \right) \quad \left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ d_{datresolvelist}\ \right\} \Rightarrow \left\{\mathrm{J}_\bot\right\}\right\}}{\left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{R_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{pattIndex}(v_{varid},\ d_{datresolvelist})\ \right\} \Rightarrow \left\{\mathfrak{Q}_{\top\bot}\left(\ \mathrm{Q}_\bot\ \right)\right\}\right\}} \tag{C.118}$$

$$\frac{\frac{R_\top}{U_\top}\left(\ v_{varid}\ \right) = \frac{\mathscr{Y}_\top}{U_\top}\left(\ \frac{T_\top}{U_\top}\ \right) \quad \left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ d_{datresolvelist}\ \right\} \Rightarrow \left\{\frac{J_\top}{Y_\top,U_\top}\right\}\right\}}{\left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{R_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{pattIndex}(v_{varid},\ d_{datresolvelist})\ \right\} \Rightarrow \left\{\mathfrak{Q}_{\top\bot}\left(\ \frac{J_\top}{Y_\top,U_\top}\ \right)\right\}\right\}} \tag{C.119}$$

Matches on the *datresolvelist* syntactic object:

$$\frac{\left\{\ U_\top,\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ d_{datresolve}\ \right\} \Rightarrow \left\{\mathrm{T}_\bot\right\}\right\}}{\left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{datResolveInd}\left(d_{datresolve},\ d'_{datresolvelist}\right)\ \right\} \Rightarrow \left\{\mathfrak{J}_{\top\bot}\left(\ \mathrm{J}_\bot\ \right)\right\}\right\}} \tag{C.120}$$

$$\frac{\left\{\ U_\top,\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ d_{datresolve}\ \right\} \Rightarrow \left\{\frac{T'_\top}{U_\top}\right\}\right\} \quad \left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T'_\top}{U_\top}\ \ \vdash \left\{\ d'_{datresolvelist}\ \right\} \Rightarrow \left\{\mathrm{J}_\bot\right\}\right\}}{\left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{datResolveInd}\left(d_{datresolve},\ d'_{datresolvelist}\right)\ \right\} \Rightarrow \left\{\mathfrak{J}_{\top\bot}\left(\ \mathrm{J}_\bot\ \right)\right\}\right\}} \tag{C.121}$$

$$\frac{\left\{\ U_\top,\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ d_{datresolve}\ \right\} \Rightarrow \left\{\frac{T'_\top}{U_\top}\right\}\right\} \quad \left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T'_\top}{U_\top}\ \ \vdash \left\{\ d'_{datresolvelist}\ \right\} \Rightarrow \left\{\frac{J_\top}{Y_\top,U_\top}\right\}\right\}}{\left\{\begin{array}{l} U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{datResolveInd}\left(d_{datresolve},\ d'_{datresolvelist}\right)\ \right\} \\ \Rightarrow \left\{\mathfrak{J}_{\top\bot}\left(\ \frac{\mathscr{J}_\top}{Y_\top,U_\top}\left(\ \frac{T'_\top}{U_\top},\ \frac{J_\top}{Y_\top,U_\top}\ \right)\ \right)\right\}\end{array}\right\}} \tag{C.122}$$

$$\frac{\frac{Y_\top}{U_\top} = \frac{\mathscr{Y}_\top}{U_\top}\left(\ \frac{T_\top}{U_\top}\ \right)}{\left\{\ U_\top,\ \frac{Y_\top}{U_\top},\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ \mathtt{datResolveBase}()\ \right\} \Rightarrow \left\{\mathfrak{J}_{\top\bot}\left(\ \frac{\mathscr{J}_\top}{Y_\top,U_\top}()\ \right)\right\}\right\}} \tag{C.123}$$

Matches on the *datresolve* syntactic object:

$$\frac{\left\{\ U_\top\ \ \vdash \left\{\ \mathsf{tupTypeDestruct}\left(\ \frac{T_\top}{U_\top},\ i_{intconst}\ \right)\ \right\} \Rightarrow \left\{\mathrm{T}_\bot\right\}\right\}}{\left\{\ U_\top,\ \frac{T_\top}{U_\top}\ \ \vdash \left\{\ i_{intconst}\ \right\} \Rightarrow \left\{\mathfrak{T}_{\top\bot}\left(\ \mathrm{T}_\bot\ \right)\right\}\right\}} \tag{C.124}$$

$$\frac{\left\{\ U_{\top}\ \vdash \left\{\ \mathsf{tupTypeDestruct}\left(\ \frac{T_{\top}}{U_{\top}},\ i_{intconst}\ \right)\ \right\} \Rightarrow \left\{\frac{T_{\top}'}{U_{\top}}\right\}\right\}}{\left\{\ U_{\top},\ \frac{T_{\top}}{U_{\top}}\ \vdash \left\{\ i_{intconst}\ \right\} \Rightarrow \left\{\mathfrak{T}_{\top\bot}\left(\ \frac{T_{\top}'}{U_{\top}}\ \right)\right\}\right\}} \tag{C.125}$$

$$\frac{\left\{\ U_{\top}\ \vdash \left\{\ \mathsf{recTypeDestruct}\left(\ \frac{T_{\top}}{U_{\top}},\ v_{varid}\ \right)\ \right\} \Rightarrow \left\{\mathrm{T}_{\bot}\right\}\right\}}{\left\{\ U_{\top},\ \frac{T_{\top}}{U_{\top}}\ \vdash \left\{\ v_{varid}\ \right\} \Rightarrow \left\{\mathfrak{T}_{\top\bot}\left(\ \mathrm{T}_{\bot}\ \right)\right\}\right\}} \tag{C.126}$$

$$\frac{\left\{\ U_{\top}\ \vdash \left\{\ \mathsf{recTypeDestruct}\left(\ \frac{T_{\top}}{U_{\top}},\ v_{varid}\ \right)\ \right\} \Rightarrow \left\{\frac{T_{\top}'}{U_{\top}}\right\}\right\}}{\left\{\ U_{\top},\ \frac{T_{\top}}{U_{\top}}\ \vdash \left\{\ v_{varid}\ \right\} \Rightarrow \left\{\mathfrak{T}_{\top\bot}\left(\ \frac{T_{\top}'}{U_{\top}}\ \right)\right\}\right\}} \tag{C.127}$$

C.3 Coordination dynamic semantics

We now give operational semantics of run-time HBCL in an evaluation style, enriched with the type system discussed in section 4.4. We give all rules down to the point where their premises can be explained in simple English. Our general scheme is to introduce the rule, then give the rule, and follow it with a more detailed explanation.

C.3.1 Program super-step

The semantic rules in this section describe how the four-fold round robin scheduling works. In rule C.128, the inductive case of the FIFO-generating rule is given, while rule C.129 handles the base case. Rule C.130 matches FIFO-box memory execution, rule C.131 determines the state evolution as boxes are (atomically) executed, and rule C.132 gives the behaviour for box-FIFO executing memories. Each rule of the super-step can be thought of as specifying a pattern match on a coinductive type: this is why each inner trace under a constructor is one step *forward* in time.

A match on the inductive constructor of the input stream drives evaluation; when the input stream matches the base case that marks the end of input (if there is one — executions may be infinite), then the coordination trace terminates in an empty constructor. It is important to note that in rule C.128, the time of temporal validity of the input memory map and the temporal validity of the inner stream are of the *next* time slice with respect to the slice in which the match is taking place. It can also be observed that the inductive FIFO match has two premises. The first is a match on the FIFO execution step; the second is a match that uses this match to force evaluation of the next step in the trace. The choice of the phrase 'force evaluation' is no accident: the operational semantics are rendered as an interpreter by direct translation into a functional language that uses lazy evaluation to examine the contents of a trace. Since rule C.129 is a base case, it has no premises. The type of the input stream is constrained to supply, in each memory map arriving on each clock tick, all values for all FIFOs in the coordination object that would otherwise block on input. Not all FIFOs are enabled on each step, only those whose frequencies divide exactly the input map frequency.

The rest of the super-step rules have the property that they will never block on input until the next FIFO-step is reached. As a consequence, traces always begin and end on FIFO steps, the last one being an empty step with an empty trace constructor. The first step of this type is the rule that constructs the trace object with the FIFO-box memory step uppermost. This is described by rule C.130. Memory execution is a temporal firewall which allows a memory to digest its input values and assert the correct readable values. As with the FIFO step, only enabled memories (those whose frequency exactly divides the input map frequency) may execute. The FIFO-box memory step calls the next boxes step with the coordination object clock incremented by one tick, since box computation is notionally the first thing to take place in one global time step. Again, there are two premises, one for the next trace element construction, and one for the step computation that is an argument to that trace construction.

We now review the super-step rules in detail, starting with the FIFO step.

Rule C.131 and rule C.132 determine the box executions and box-FIFO memory executions, and proceed in exactly the same way as the FIFO-box memory steps, save that they do not have the complication of incrementing the time.

$$
\dfrac{
\begin{array}{l}
\left\{\begin{array}{l}
E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \dfrac{K_{\top}}{K_{\mathsf{S}\top}},\ \dfrac{I_{\top}}{I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \dfrac{t_{\top}}{f_{freq}},\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f'_{freq},\ f_{freq}\ \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq}\ \right)\\
\mathsf{timeEq}_{\mathsf{Prop}}\left(\ \dfrac{t_{\top}}{f_{freq}},\ \dfrac{t'_{\top}}{f'_{freq}}\ \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}}\ \right)\\
\vdash \left\{\ \dfrac{N_{\mathrm{MOI}\top}}{t'_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}},\ \dfrac{C_{\mathrm{FIFOs}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\ \right\} \Rightarrow \left\{\dfrac{C_{\mathrm{memFB}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\right\}
\end{array}\right\}\\
\left\{\begin{array}{l}
E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \dfrac{K_{\top}}{K_{\mathsf{S}\top}},\ \dfrac{I_{\top}}{I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}}\\
\dfrac{C_{\mathrm{memFB}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f'_{freq},\ f_{freq}\ \right)\\
\mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq}\ \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}}\ \right)\\
\vdash \left\{\ \dfrac{S_{\top}}{t''_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}}\ \right\} \Rightarrow \left\{\dfrac{Tr_{\mathrm{memFB}\top}}{C_{\mathrm{memFB}\top},t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\right\}
\end{array}\right\}
\end{array}
}{
\left\{\begin{array}{l}
E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \dfrac{K_{\top}}{K_{\mathsf{S}\top}},\ \dfrac{I_{\top}}{I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \dfrac{t_{\top}}{f_{freq}},\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}}\\
\dfrac{C_{\mathrm{FIFOs}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f'_{freq},\ f_{freq}\ \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq}\ \right)\\
\mathsf{timeEq}_{\mathsf{Prop}}\left(\ \dfrac{t_{\top}}{f_{freq}},\ \dfrac{t'_{\top}}{f'_{freq}}\ \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}}\ \right)\\
\vdash \left\{\ \mathscr{S}_{\top}\left(\ \dfrac{N_{\mathrm{MOI}\top}}{t'_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}},\ \dfrac{S_{\top}}{t''_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}}\ \right)\right\}\\
\Rightarrow \left\{\dfrac{\mathscr{T}\!r_{\mathrm{FIFOs}\top}\left(\ \dfrac{C_{\mathrm{MemFB}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \dfrac{Tr_{\mathrm{MemFB}\top}}{C_{\mathrm{memFB}\top},t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\ \right)}{C_{\mathrm{FIFOs}\top},t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\right\}
\end{array}\right\}
}
\tag{C.128}
$$

The purpose of the inductive FIFO step of rule C.128 is to construct the trace in the implicand of the conclusion. It has the FIFOS subscript to show that it is a trace whose dependent type is fixed by the current 'FIFOS enabled' coordination state, which can be seen in the list of environment variables for this statement. There is a one-to-one correspondence between the values of this type of trace and the values of the input stream. The operational semantic rules give a procedure for finding the inhabitant of this trace type that corresponds to the input stream given in the implicant of the conclusion. The premises must generate a 'memFB' type trace so that the 'FIFO' type trace can be constructed in the conclusion. In order to provide the necessary information, the conclusion's implicant is deconstructed to form a snapshot of the input memory map for the current time slice and the next co-inductive part of the input stream, which we can see has a type dependent in the time of the *next* execution slice. The type environment, instance signature, instance closure signature and instance closure are only of static interest and fix the dependent type of the logical instance, which itself fixes the type of the coordination state object.

The first premise invokes the FIFO-step rule, which we shall meet shortly. A number of important predicates ensure that the input map is structurally and temporally consistent with the FIFO-enabled coordination state it is about to update. $\mathsf{freqDivides}_{\mathsf{Prop}}$ requires that the frequency of the input map is an exact divisor of the frequency of the coordination state object. $\mathsf{allFreqsDivide}_{\mathsf{Prop}}$ ensures that the frequency of the input map is the lowest common multiple of all the frequencies of memories in that map. The input map in question is dereferenced from the logical instance that defines the static structure of the coordination object using the 'dot' notation.$\mathsf{timeEq}_{\mathsf{Prop}}$ stipluates that the current time $\frac{t_{\top}}{f_{freq}}$ (dependent in the frequency of the coordination state) is the same as the current time $\frac{t'_{\top}}{f'_{freq}}$ (dependent in the frequency of the input map), notwithstanding the fact that the underlying natural numbers representing the time in cycles since time zero are different. $\mathsf{timeNext}_{\mathsf{Prop}}$ fixes the requirement that $\frac{t''_{\top}}{f'_{freq}}$ is the time one cycle later than $\frac{t'_{\top}}{f'_{freq}}$, at the frequency of the input stream, f'_{freq}.

The second premise invokes the next step in trace construction, namely the 'memFB' trace, which adds the coordination state after FIFO-box memory execution. The time of the next trace is that of the current time slice, but the accompanying input trace is that of the *next* time slice: it will not be consumed until the FIFO execution step of the next time slice. The new 'memFB' coordination state we obtained in the first premise appears in the second premise as one of the dependent type arguments of the next trace, constraining that trace to be one that follows from the given 'memFB' coordination state and the next input stream.

$$\overline{\left\{\begin{array}{l} E_{\top\top},\ f_{freq},\ f'_{freq},\ \frac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \frac{K_{\top}}{K_{\mathrm{S}\top}},\ \frac{I_{\top}}{I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}},\ \frac{t_{\top}}{f_{freq}},\ \frac{t'_{\top}}{f'_{freq}},\ \frac{C_{\mathrm{FIFOs}\top}}{t_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq} \right) \\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\top}}{f_{freq}},\ \frac{t'_{\top}}{f'_{freq}} \right) \\ \vdash \left\{ \frac{\mathsf{S}_{\top}}{t_{\top},I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}} \right\} \Rightarrow \left\{ \frac{\mathscr{T\!r}_{\mathrm{FIFOs}\top}()}{C_{\mathrm{FIFOs}\top},t_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\}} \tag{C.129}$$

The second match for the FIFO step given in rule C.129 has no premises. The stream object that has matched in this case is the final stream object, which carries no data, but signifies the end of a finite stream. The version of the FIFO state trace object constructor that is invoked also has no arguments, but has the correct dependent type. The type is dependent on arguments in the environment: these are the same as for the non-terminal case of rule C.128.

$$\frac{\left\{\begin{array}{l} E_{\mathsf{TT}},\ f_{freq},\ \dfrac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \dfrac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \dfrac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \dfrac{t_{\mathsf{T}}}{f_{freq}},\ \dfrac{t'_{\mathsf{T}}}{f_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f'_{freq} \right) \\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t_{\mathsf{T}}}{f_{freq}},\ \dfrac{t'_{\mathsf{T}}}{f_{freq}} \right) \\ \vdash \left\{ \dfrac{C_{\mathrm{MemFB}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \Rightarrow \left\{ \dfrac{C_{\mathrm{Boxes}\mathsf{T}}}{t'_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \end{array}\right\} \quad \left\{\begin{array}{l} E_{\mathsf{TT}},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \dfrac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \dfrac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \dfrac{t'_{\mathsf{T}}}{f_{freq}},\ \dfrac{t''_{\mathsf{T}}}{f'_{freq}} \\ \dfrac{C_{\mathrm{Boxes}\mathsf{T}}}{t'_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right) \\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f'_{freq} \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\dfrac{t'_{\mathsf{T}}}{f_{freq}},\ \dfrac{t''_{\mathsf{T}}}{I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}}.freq} \right) \\ \vdash \left\{ \dfrac{S_{\mathsf{T}}}{t''_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},f'_{freq}} \right\} \Rightarrow \left\{ \dfrac{Tr_{\mathrm{Boxes}\mathsf{T}}}{C_{\mathrm{Boxes}\mathsf{T}},t'_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathsf{TT}},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \dfrac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \dfrac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \dfrac{t_{\mathsf{T}}}{f_{freq}},\ \dfrac{t'_{\mathsf{T}}}{f_{freq}},\ \dfrac{t''_{\mathsf{T}}}{f'_{freq}} \\ \dfrac{C_{\mathrm{MemFB}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f'_{freq} \right) \\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\dfrac{t'_{\mathsf{T}}}{f_{freq}},\ \dfrac{t''_{\mathsf{T}}}{f'_{freq}} \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t_{\mathsf{T}}}{f_{freq}},\ \dfrac{t'_{\mathsf{T}}}{f_{freq}} \right) \\ \vdash \left\{ \dfrac{S_{\mathsf{T}}}{t''_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},f'_{freq}} \right\} \\ \Rightarrow \left\{ \dfrac{\mathscr{Tr}_{\mathrm{MemFB}\mathsf{T}}\left(\dfrac{C_{\mathrm{Boxes}\mathsf{T}}}{t'_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \dfrac{Tr_{\mathrm{Boxes}\mathsf{T}}}{C_{\mathrm{Boxes}\mathsf{T}},t'_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right)}{C_{\mathrm{MemFB}\mathsf{T}},t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \end{array}\right\}} \tag{C.130}$$

The trace construction for the FIFO-box memory execution of rule C.130 step is similar to that for the FIFO step of rule C.128. The main difference is that the input stream is not deconstructed in the conclusion implicant, but is passed on unaltered to the implicant of the second premise, which invokes the next phase of the super-step trace construction. The environment variables are the same, except $\dfrac{t'_{\mathsf{T}}}{f_{freq}}$. The primed time is this time dependent in the frequency of the coordination state, *not* the input map, and the time it represents is the *next* time state for the coordination state. The arguments of $\mathsf{timeEq}_{\mathsf{Prop}}$ and $\mathsf{timeNext}_{\mathsf{Prop}}$ are permuted accordingly. The box step is unique among the four steps in that it is the only step at which the global clock advances.

As in rule C.128, the first premise generates the next coordination state, while the second premise uses this to build the next (boxes step) trace object, which is then itself used in the conclusion to build the current trace object. Compared with rule C.128, the FIFO-BOX memory step of rule C.130 has coordination objects subscripted 'memFB' instead of 'FIFO', and 'Boxes' instead of 'memFB'. Accordingly the next step is dependent in the type of the 'Boxes' enabled step rather than in the 'memFB' enabled step.

$$\frac{\begin{array}{l}\left\{\begin{array}{l} E_{\mathsf{TT}},\ f_{freq},\ \frac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \frac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \frac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{t_{\mathsf{T}}}{f_{freq}} \\ \vdash \left\{ \frac{C_{\mathrm{Boxes}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \Rightarrow \left\{ \frac{C_{\mathrm{MemBF}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathsf{TT}},\ f_{freq},\ f'_{freq},\ \frac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \frac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \frac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t'_{\mathsf{T}}}{f'_{freq}} \\ \frac{C_{\mathrm{MemBF}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right) \\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f'_{freq} \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t'_{\mathsf{T}}}{f'_{freq}} \right) \\ \vdash \left\{ \frac{S_{\mathsf{T}}}{t'_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},f'_{freq}} \right\} \Rightarrow \left\{ \frac{Tr_{\mathrm{MemBF}\mathsf{T}}}{C_{\mathrm{MemBF}\mathsf{T}},t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{\mathsf{TT}},\ f_{freq},\ f'_{freq},\ \frac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \frac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \frac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t'_{\mathsf{T}}}{f'_{freq}} \\ \frac{C_{\mathrm{Boxes}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right) \\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f'_{freq} \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t'_{\mathsf{T}}}{f'_{freq}} \right) \\ \vdash \left\{ \frac{S_{\mathsf{T}}}{t'_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},f'_{freq}} \right\} \\ \Rightarrow \left\{ \frac{\mathscr{Tr}_{\mathrm{Boxes}\mathsf{T}}\left(\frac{C_{\mathrm{MemBF}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{Tr_{\mathrm{MemBF}\mathsf{T}}}{C_{\mathrm{MemBF}\mathsf{T}},t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right)}{C_{\mathrm{Boxes}\mathsf{T}},t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}} \right\} \end{array}\right\}} \tag{C.131}$$

The trace construction of the step that adds the result of the boxes step (rule C.131) is identical to that for the FIFO-box memory step of rule C.130, except that the time variables and relationships are again different. This time the boxes-enabled state, the next (box-FIFO memory enabled) state, and the input stream time are all the same. The input stream will be processed in the FIFO step of *this* time slice. The two time variables differ only by the frequency in which they are dependent, and the $\mathsf{timeEq}_{\mathsf{Prop}}$ predicate appears in the environment accordingly. This time the new coordination trace is dependent in the new coordination state of box-FIFO memories enabled, which appears in the implicand of the first premise. The current trace is dependent in the currrent coordination state of enabled boxes, which appears in the environment of the conclusion.

$$\frac{\begin{array}{c}\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_\top}{K_{\mathrm{S}\top}},\ \dfrac{I_\top}{I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t_\top}{f_{freq}} \\ \vdash \left\{ \dfrac{C_{\mathrm{MemBF}\top}}{t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}} \right\} \Rightarrow \left\{ \dfrac{C_{\mathrm{FIFOs}\top}}{t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_\top}{K_{\mathrm{S}\top}},\ \dfrac{I_\top}{I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t_\top}{f_{freq}},\ \dfrac{t'_\top}{f'_{freq}} \\ \dfrac{C_{\mathrm{FIFOs}\top}}{t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right) \\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq} \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\dfrac{t_\top}{f_{freq}},\ \dfrac{t'_\top}{I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top}.freq} \right) \\ \vdash \left\{ \dfrac{S_\top}{t'_\top,I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}} \right\} \Rightarrow \left\{ \dfrac{Tr_{\mathrm{FIFOs}\top}}{C_{\mathrm{FIFOs}\top},t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\}\end{array}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_\top}{K_{\mathrm{S}\top}},\ \dfrac{I_\top}{I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t_\top}{f_{freq}},\ \dfrac{t'_\top}{f'_{freq}} \\ \dfrac{C_{\mathrm{MemBF}\top}}{t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right) \\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq} \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\dfrac{t_\top}{f_{freq}},\ \dfrac{t'_\top}{f'_{freq}} \right) \\ \vdash \left\{ \dfrac{S_\top}{t'_\top,I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}} \right\} \\ \Rightarrow \left\{ \dfrac{\mathscr{Tr}_{\mathrm{MemBF}\top}\left(\dfrac{C_{\mathrm{FIFOs}\top}}{t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}},\ \dfrac{Tr_{\mathrm{FIFOs}\top}}{C_{\mathrm{FIFOs}\top},t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}} \right)}{C_{\mathrm{MemBF}\top},t_\top,I_\top,I_{\mathrm{S}\top},K_\top,K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\}} \tag{C.132}$$

The trace construction step that adds the result of the box-FIFO memory execution step (of rule C.132) is almost identical to the rule that does the same for the boxes step (rule C.131), except for the permutation of coordination states invoked. The coordination step invocation this time produces a FIFOs-enabled coordination state, and the new FIFO trace generated from the second premise is dependent in this type. The whole trace is dependent in the type of coordination state for enabled box-FIFO memory executions: this appears in the environment of the conclusion. The time relationships are the same as for the boxes step.

C.3.2 FIFO step

The FIFO step of rule C.133 has two premises. The first deals with updating the local input memory maps; the second tackles the nested case, recursing through the coordination object and updating any memories that are exposed from inner instances but are not exported by the current instance: in other words, they must be updated at the current scope, and that is what this premise specifies. As well as dealing with input supplied as a separate parameter, the FIFO step also executes FIFOs where the box-FIFO memory to be read is within scope. In the present formalization, FIFOs do not have any state associated with them: as soon as a value is available in the box-FIFO memory, it is appended to the FIFO-box (input) memory at the other end of the FIFO. This simplifies the state space, which does not need an explicit FIFO state store, but is at the expense of requiring those input memories that will be connected to longer FIFOs to have longer buffers. This is something that would be changed in a future version of the language.

$$
\dfrac{
\begin{array}{l}
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \dfrac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \dfrac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ \dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOIT}},\ f'_{freq} \right) \\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \right) \\
\vdash \left\{ \dfrac{N_{\mathrm{MOIT}}}{t'_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOIT}},f'_{freq}},\ \dfrac{N'_{\mathrm{MOIT}}}{t'_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOIT}},f'_{freq}} \right\} \Rightarrow \left\{ N''_{\mathrm{MOIT}} \right\}
\end{array}\right\} \\
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \dfrac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \dfrac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ \dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOIT}},\ f'_{freq} \right) \\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \right) \\
\vdash \left\{ \dfrac{\mu C_{\mathrm{FIFOsT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \right\} \Rightarrow \left\{ \mu C_{\mathrm{InnerFIFOsT}} \right\}
\end{array}\right\} \\
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \dfrac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \dfrac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ \dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOIT}},\ f'_{freq} \right) \\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \right) \\
\vdash \left\{ \dfrac{\mathscr{C}_{\mathrm{InnerFIFOsT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \left(\begin{array}{l} N''_{\mathrm{MOIT}},\ N_{\mathrm{MOOT}},\ \mu C_{\mathrm{InnerFIFOsT}}, \\ \mathsf{MemFBCorrect}_{\mathsf{Prop}} \left(\begin{array}{l} N''_{\mathrm{MOIT}} \\ N_{\mathrm{MOOT}} \\ \mu C_{\mathrm{InnerFIFOsT}} \\ t_{\mathrm{T}},\ I_{\mathrm{T}},\ I_{\mathrm{ST}},\ K_{\mathrm{T}},\ K_{\mathrm{ST}},\ f_{freq} \end{array} \right) \end{array} \right) \right\} \\
\Rightarrow \left\{ \dfrac{C_{\mathrm{memFBT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \right\}
\end{array}\right\}
\end{array}
}{
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \dfrac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \dfrac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ \dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOIT}},\ f'_{freq} \right) \\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t_{\mathrm{T}}}{f_{freq}},\ \dfrac{t'_{\mathrm{T}}}{f'_{freq}} \right) \\
\vdash \left\{ \dfrac{N_{\mathrm{MOIT}}}{t'_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOIT}},f'_{freq}},\ \dfrac{\mathscr{C}_{\mathrm{FIFOsT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \left(N'_{\mathrm{MOIT}},\ N_{\mathrm{MOOT}},\ \mu C_{\mathrm{FIFOsT}} \right) \right\} \\
\Rightarrow \left\{ \dfrac{C_{\mathrm{memFBT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \right\}
\end{array}\right\}
}
\tag{C.133}
$$

The implicant of the conclusion of rule C.133 deconstructs the FIFO coordination state object into its three constituents. N'_{MOIT} is the old input memory map from the coordination object. N_{MOOT} is the output map, as modified by the last boxes and box-FIFO execution steps. μC_{FIFOsT} is the map of nested coordination map objects in the FIFOs-enabled state.

The implication of the first premise shows the invocation of a rule that constructs an updated input memory map $N^{''}_{\text{MOI}\textsc{t}}$ from the old map $N^{'}_{\text{MOI}\textsc{t}}$ and the map of new inputs $N_{\text{MOI}\textsc{t}}$. This operation involves the old input map component $N^{'}_{\text{MOI}\textsc{t}}$ of $C_{\text{FIFOs}\textsc{t}}$ being updated with the new values from $N^{'}_{\text{MOI}\textsc{t}}$. For each new memory, in $N^{'}_{\text{MOI}\textsc{t}}$, its list of data values is prepended to the old memory map. For the sake of brevity, we do not show the rule for updating these memories, as it was straightforward to describe in English. We also elide the dependent arguments on input maps: the input stream contains data that is new on this time slice, while the old memory map from the coordination state contains old data. A further predicate would be necessary here to reduce this rule to a form without premises.

The implication of the second premise concerns the update of nested input memory maps. The point of interest here is that the coordination state is subscripted *inner* FIFOS. This is because the outward-facing FIFOS (those whose terminating memories have been exported to the enclosing instance) are updated in this step at the enclosing scope: in effect, the first premise is recursively applied throughout the tree of nested coordination objects. Only the inner FIFOS are left in a condition to be executed on invocation of the nested step, which does not happen until the invocation of the boxes step. Again, we elide the need for a predicate on the input map in a rule implementing the FIFO appending operation, which we omit.

The third premise uses the new values inferred from the premises to build the next *inner* FIFOS coordination object, in which FIFO-box memories are ready to be executed without the need for any external input. This term (in the implicant) looks similar to the object that was deconstructed in the conclusion implicant, except that the input and nested maps are now updated versions. We show the requirement of a predicate to ensure that the new object is well-formed. The structure of this predicate has not been formally defined: to do so would require operational structural semantic rules that would yield pure logical propositions. This is deferred to further work, since it is a refinement that would very substantially add to the size of the rules, but does not affect the operational behaviour. The premise invokes a rule that, for those memories corresponding to enabled FIFOS in $N_{\text{MOO}\textsc{t}}$, copies values that were not read on the previous FIFO run. Each copied value is appended to the memory in $N^{'}_{\text{MOI}\textsc{t}}$ that is statically specified by the FIFO definition. Again, we do not write out this rule in operational notation. The result is the next 'MemFB' coordination object, which is immediately returned by the implicand of the conclusion.

C.3.3 MemFB step

The memory execution step (rule C.134) has a single premise. This is because the memory execution only occurs at *local* scope: memory execution within the nested case is taken care of by the nested boxes step. In the present formalization, memories are lists, and their execution involves reversing the list. If the memories are double-buffered, only the FIFO-box lists are reversed on a FIFO-box step; otherwise, both lists are reversed on both steps. The approaches are bisimilar, and we adopt the latter because it is simpler, although the former is more elegant. It is a crucial element of HBCL that communication by these memories is *not* based on message passing, but synchronized entirely by the global clock. A memory must be specified by what should be observable at a particular time. The observable part of a memory is a bounded-length buffer in which there are limits on how far in advance of the current time a value *may* be present, how far in advance of the current time a value *must* be present, how far in arrears of the current time old values *must* still be visible, and how far in arrears of the current time old values *may* still be visible.

$$
\dfrac{
\begin{array}{l}
\left\{
\begin{array}{l}
E_{\mathsf{TT}},\ f_{freq},\ f^{'}_{freq},\ \frac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \frac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \frac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t^{''}_{\mathsf{T}}}{f^{'}_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f^{'}_{freq},\ f_{freq}\ \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f^{'}_{freq}\ \right)\\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t^{''}_{\mathsf{T}}}{f^{'}_{freq}}\ \right)\\
\vdash \left\{\ \frac{N_{\mathrm{MOI}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},f^{'}_{freq}}\ \right\} \Rightarrow \left\{\frac{N^{'}_{\mathrm{MOI}\mathsf{T}}}{t^{''}_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},f^{'}_{freq}}\right\}
\end{array}
\right\}\\
\left\{
\begin{array}{l}
E_{\mathsf{TT}},\ f_{freq},\ f^{''}_{freq},\ \frac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \frac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \frac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t^{'''}_{\mathsf{T}}}{f^{''}_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f^{''}_{freq},\ f_{freq}\ \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{ST}}.M_{\mathrm{MOO}\mathsf{T}},\ f^{'}_{freq}\ \right)\\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t^{'''}_{\mathsf{T}}}{f^{''}_{freq}}\ \right)\\
\vdash \left\{\ \frac{N_{\mathrm{MOO}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOO}\mathsf{T}},f^{''}_{freq}}\ \right\} \Rightarrow \left\{\frac{N^{'}_{\mathrm{MOO}\mathsf{T}}}{t^{'''}_{\mathsf{T}},I_{\mathsf{ST}}.M_{\mathrm{MOO}\mathsf{T}},f^{''}_{freq}}\right\}
\end{array}
\right\}
\end{array}
}{
\left\{
\begin{array}{l}
E_{\mathsf{TT}},\ f_{freq},\ f^{'}_{freq},\ \frac{I_{\mathsf{ST}}}{f_{freq}},\ K_{\mathsf{ST}},\ \frac{K_{\mathsf{T}}}{K_{\mathsf{ST}}},\ \frac{I_{\mathsf{T}}}{I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}},\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t^{'}_{\mathsf{T}}}{f^{'}_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f^{'}_{freq},\ f_{freq}\ \right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{ST}}.M_{\mathrm{MOI}\mathsf{T}},\ f^{'}_{freq}\ \right)\\
\mathsf{timeNext}_{\mathsf{Prop}}\left(\ \frac{t_{\mathsf{T}}}{f_{freq}},\ \frac{t^{'}_{\mathsf{T}}}{f_{freq}}\ \right)\\
\vdash \left\{\ \frac{\mathscr{C}_{\mathrm{memFB}\mathsf{T}}}{t_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}}\left(\ N_{\mathrm{MOI}\mathsf{T}},\ N_{\mathrm{MOO}\mathsf{T}},\ \mu C_{\mathrm{InnerFIFOs}\mathsf{T}}\ \right)\ \right\}\\
\Rightarrow \left\{
\frac{\mathscr{C}_{\mathrm{boxes}\mathsf{T}}}{t^{'}_{\mathsf{T}},I_{\mathsf{T}},I_{\mathsf{ST}},K_{\mathsf{T}},K_{\mathsf{ST}},f_{freq}}
\left(
\begin{array}{l}
N^{'}_{\mathrm{MOI}\mathsf{T}},\ N^{'}_{\mathrm{MOO}\mathsf{T}},\ \mu C_{\mathrm{InnerFIFOs}\mathsf{T}},\\
\mathsf{BoxesCorrect}_{\mathsf{Prop}}\left(
\begin{array}{l}
N^{'}_{\mathrm{MOI}\mathsf{T}}\\
N^{'}_{\mathrm{MOO}\mathsf{T}}\\
\mu C_{\mathrm{InnerFIFOs}\mathsf{T}}\\
t^{'}_{\mathsf{T}},\ I_{\mathsf{T}},\ I_{\mathsf{ST}},\ K_{\mathsf{T}},\ K_{\mathsf{ST}},\ f_{freq}
\end{array}
\right)
\end{array}
\right)
\right\}
\end{array}
\right\}
}
\tag{C.134}
$$

The FIFO-box execution step of rule C.134 is similar to that of rule C.133, except that there is now no interaction between input and output memories, and nested coordination state objects are unaffected by the application of the rule. The implicant of the conclusion again deconstructs the coordination object, but this time that coordination object is subscripted 'memFB' to signify that the FIFO-box memories are ready to be executed.

The first premise's implicant shows the old memory map being converted to the new (primed) memory map. The new memory map is dependent on the *primed* time variable, indicating that it now belongs to the next time slice, ready to be read by the next boxes step and updated with the next FIFO step. The original input memory map appears to have become dependent in the current time and static memory map description since being extracted from the implicant of the conclusion. This is because this dependence is generated purely by up-casting to a parametrized σ-type. The dependence is only due to the parameters of the underlying predicate, which is inferred from the destruction of the conclusion implicant's coordination state. We have not specified the structure of this predicate, but we know it will have to be capable of being deconstructed to yield the predicate we need. The possibility of such deconstruction gives a lemma implying that the memory map predicate must follow from the predicate over the whole coordination object from which the memory map is obtained. The operation on the underlying concrete maps that is occurring in this transition is that stale data values are discarded (ones that will have been read by a box in the previous step). This is determined by the maximum memory length value in the static specification, although a more sophisticated approach would be to infer this value from the frequency relationship of the FIFO and box memories. In the present version of the semantics, execution also involves reversing a list representing a single buffer of timed data of both input and output buffers. The latter operation is invoked by the second premise.

The implicand of the conclusion builds the new coordination object using the updated input memory map in the same way as rule C.133.

C.3.4 Boxes step

The boxes step has four rules. Rule C.135 describes the top level match, which decomposes the problem into the local execution of boxes and the nested execution of boxes. The nested case is dealt with in appendix C.3.5, so the remaining three steps describe the local case, where what is observed from FIFO-box memories produces a new set of box-FIFO memories.

Rule C.136 is a map reduction rule, where a new premise is spawned for each enabled harmonic box. The rule extracts the specification of the box, and the selection of the required input and output memories needed by the box in question.

Rule C.137 shows how the case of the untimed box binding (the only binding we currently have) is extracted from the general harmonic box type. Rule C.137 also shows the extraction of the mapping between object identifiers, which give handles to the memories on the outside of the box, and variable identifiers, which have semantic meaning inside the untimed box.

Finally, rule C.138 shows how the binding operates and how the untimed box language is called. Again, we see that the box language is a first order entity in these semantics. There are three premises to this rule. The first handles the conversion of data values from the coordination language to the box language; the second invokes the box language with these data values; the third converts the result back into the memory types of the coordination language.

$$
\frac{
\begin{array}{l}
\left\{\begin{array}{l}
E_{\mathrm{T}\top},\ f_{freq},\ f^{'}_{freq},\ f^{''}_{freq},\ \frac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \frac{K_{\top}}{K_{\mathsf{S}\top}},\ \frac{I_{\top}}{I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \frac{t_{\top}}{f_{freq}},\ \frac{t^{'}_{\top}}{f^{'}_{freq}},\ \frac{t^{''}_{\top}}{f^{''}_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f^{'}_{freq},\ f_{freq}\ \right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f^{''}_{freq},\ f_{freq}\ \right)\\
\mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{S}\top}.M_{\mathrm{MOO}\top},\ f^{'}_{freq}\ \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\ \frac{t_{\top}}{f_{freq}},\ \frac{t^{'}_{\top}}{f^{'}_{freq}}\ \right)\\
\mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(\ I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},\ f^{''}_{freq}\ \right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\ \frac{t_{\top}}{f_{freq}},\ \frac{t^{''}_{\top}}{f^{''}_{freq}}\ \right)\\
\vdash \left\{\ \frac{N_{\mathrm{MOI}\top}}{t^{''}_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOI}\top},f^{''}_{freq}},\ \frac{N_{\mathrm{MOO}\top}}{t^{'}_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOO}\top},f^{'}_{freq}}\ \right\} \Rightarrow \left\{\frac{N^{'}_{\mathrm{MOO}\top}}{t^{'}_{\top},I_{\mathsf{S}\top}.M_{\mathrm{MOO}\top},f^{'}_{freq}}\right\}
\end{array}\right\}\\
\left\{\begin{array}{l}
E_{\mathrm{T}\top},\ f_{freq},\ f^{'''}_{freq},\ \frac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \frac{K_{\top}}{K_{\mathsf{S}\top}},\ \frac{I_{\top}}{I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \frac{t_{\top}}{f_{freq}},\ \frac{t^{'''}_{\top}}{f^{'''}_{freq}},\ \frac{t^{4'}_{\top}}{f^{'''}_{freq}}\\
\mathsf{timeEq}_{\mathsf{Prop}}\left(\ \frac{t_{\top}}{f_{freq}},\ \frac{t^{'''}_{\top}}{f^{'''}_{freq}}\ \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\ \frac{t^{4'}_{\top}}{f^{'''}_{freq}},\ \frac{t^{'''}_{\top}}{f^{'''}_{freq}}\ \right)\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f^{'''}_{freq},\ f_{freq}\ \right)\\
\vdash \left\{\ \frac{\mu C_{\mathrm{InnerFIFOs}\top}}{t^{4'}_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f^{'''}_{freq}}\ \right\} \Rightarrow \left\{\frac{\mu C_{\mathrm{FIFOs}\top}}{t^{'''}_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f^{'''}_{freq}}\right\}
\end{array}\right\}
\end{array}
}{
\left\{\begin{array}{l}
E_{\mathrm{T}\top},\ f_{freq},\ f^{'}_{freq},\ \frac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \frac{K_{\top}}{K_{\mathsf{S}\top}},\ \frac{I_{\top}}{I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}},\ \frac{t_{\top}}{f_{freq}},\ \frac{t^{'}_{\top}}{f^{'}_{freq}}\\
\vdash \left\{\ \frac{\mathscr{C}_{\mathrm{Boxes}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\left(\ N_{\mathrm{MOI}\top},\ N_{\mathrm{MOO}\top},\ \mu C_{\mathrm{InnerFIFOs}\top}\ \right)\ \right\}\\
\Rightarrow \left\{\frac{\mathscr{C}_{\mathrm{MemBF}\top}}{t_{\top},I_{\top},I_{\mathsf{S}\top},K_{\top},K_{\mathsf{S}\top},f_{freq}}\left(\begin{array}{l}
N_{\mathrm{MOI}\top},\ N^{'}_{\mathrm{MOO}\top},\ \mu C_{\mathrm{FIFOs}\top},\\
\mathsf{MemBFCorrect}_{\mathsf{Prop}}\left(\begin{array}{l}
N_{\mathrm{MOI}\top}\\
N^{'}_{\mathrm{MOO}\top}\\
\mu C_{\mathrm{FIFOs}\top}\\
t_{\top},\ I_{\top},\ I_{\mathsf{S}\top},\ K_{\top},\ K_{\mathsf{S}\top},\ f_{freq}
\end{array}\right)
\end{array}\right)\right\}
\end{array}\right\}
}
\quad \text{(C.135)}
$$

The conclusion of the main rule for matching the change of coordination state on a box step (rule C.135) deconstructs the coordination state, which is in the boxes step-enabled state. The inner map is still in the inner FIFOs-enabled state, since the FIFO step for the nested instances cannot be run until the local boxes have had a chance to produce their input: which may be needed by that inner FIFO step.

The implicant of the first premise takes the input and output maps, with the former ready for reading by the currently enabled local boxes, and the latter ready for writing.

The second premise takes the nested coordination state map and precipitates the invocation of the nested box step to bring the coordination state from the 'inner FIFOs-enabled' state of the last time slice to the 'FIFOs-enabled' state of the current time slice. This reflects the execution of an entire four-fold cycle within the nested instance. It will become clear how this works when we present the semantic rule for nested execution.

The implicand of the conclusion builds the next coordination state using the old input map, the new output map and the new coordination state object.

$$\frac{\begin{array}{l}
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'_{freq},\ f'''_{freq},\ f^{4'}_{freq},\ \frac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \frac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \frac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ M'_{\mathrm{MOIT}},\ M'_{\mathrm{MOOT}}\\
\frac{t'_{\mathrm{T}}}{f'_{freq}},\ \frac{t^{4'}_{\mathrm{T}}}{f^{4'}_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'''_{freq},\ f_{freq}\right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f^{4'}_{freq},\ f'''_{freq}\right)\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq}\right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f^{4'}_{freq},\ f'_{freq}\right)\\
\mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t'_{\mathrm{T}}}{f'_{freq}},\ \frac{t^{4'}_{\mathrm{T}}}{f^{4'}_{freq}}\right)\\
\vdash \left\{\frac{N_{\mathrm{MOIT}}}{t'_{\mathrm{T}},I_{\mathrm{ST}}.M'_{\mathrm{MOIT}},f'_{freq}},\ \frac{H_{\mathrm{OT}}}{M'_{\mathrm{MOOT}},M'_{\mathrm{MOIT}},f'''_{freq},E_{\mathrm{TT}}}\right\} \Rightarrow \left\{\frac{N'_{\mathrm{MOIT}}}{t^{4'}_{\mathrm{T}},I_{\mathrm{ST}}.M'_{\mathrm{MOIT}},f^{4'}_{freq}}\right\}
\end{array}\right\}\\
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'''_{freq},\ f^{4'}_{freq},\ f^{5'}_{freq},\ \frac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \frac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \frac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ M'_{\mathrm{MOIT}},\ M'_{\mathrm{MOOT}}\\
\frac{H_{\mathrm{OT}}}{M'_{\mathrm{MOOT}},M'_{\mathrm{MOIT}},f'''_{freq},E_{\mathrm{TT}}},\ \frac{t^{4'}_{\mathrm{T}}}{f^{4'}_{freq}},\ \frac{t^{5'}_{\mathrm{T}}}{f^{5'}_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'''_{freq},\ f_{freq}\right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f^{4'}_{freq},\ f'''_{freq}\right)\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f^{5'}_{freq},\ f'''_{freq}\right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t^{4'}_{\mathrm{T}}}{f^{4'}_{freq}},\ \frac{t^{5'}_{\mathrm{T}}}{f^{5'}_{freq}}\right)\\
\vdash \left\{\frac{N'_{\mathrm{MOIT}}}{t^{4'}_{\mathrm{T}},I_{\mathrm{ST}}.M'_{\mathrm{MOIT}},f^{4'}_{freq}},\ \frac{H_{\mathrm{OT}}}{M'_{\mathrm{MOOT}},M'_{\mathrm{MOIT}},f'''_{freq},E_{\mathrm{TT}}}\right\} \Rightarrow \left\{\frac{N'_{\mathrm{MOOT}}}{t^{5'}_{\mathrm{T}},I_{\mathrm{ST}}.M'_{\mathrm{MOOT}},f^{5'}_{freq}}\right\}
\end{array}\right\}\\
\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f''_{freq},\ f^{5'}_{freq},\ \frac{t''_{\mathrm{T}}}{f''_{freq}},\ \frac{t^{5'}_{\mathrm{T}}}{f^{5'}_{freq}},\ M_{\mathrm{MOOT}},\ M'_{\mathrm{MOOT}}\\
\vdash \left\{\frac{N_{\mathrm{MOOT}}}{t''_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOOT}},f''_{freq}},\ \frac{N'_{\mathrm{MOOT}}}{t^{5'}_{\mathrm{T}},I_{\mathrm{ST}}.M'_{\mathrm{MOOT}},f^{5'}_{freq}}\right\} \Rightarrow \left\{\frac{N''_{\mathrm{MOOT}}}{t^{5'}_{\mathrm{T}},I_{\mathrm{ST}}.M'_{\mathrm{MOOT}},f^{5'}_{freq}}\right\}
\end{array}\right\}
\end{array}}{\left\{\begin{array}{l}
E_{\mathrm{TT}},\ f_{freq},\ f'_{freq},\ f''_{freq},\ \frac{I_{\mathrm{ST}}}{f_{freq}},\ K_{\mathrm{ST}},\ \frac{K_{\mathrm{T}}}{K_{\mathrm{ST}}},\ \frac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}},\ \frac{t_{\mathrm{T}}}{f_{freq}},\ \frac{t'_{\mathrm{T}}}{f'_{freq}},\ \frac{t''_{\mathrm{T}}}{f''_{freq}}\\
\mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq}\right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f''_{freq},\ f_{freq}\right)\\
\mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOIT}},\ f'_{freq}\right),\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOOT}},\ f''_{freq}\right)\\
\mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\mathrm{T}}}{f_{freq}},\ \frac{t'_{\mathrm{T}}}{f'_{freq}}\right),\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\mathrm{T}}}{f_{freq}},\ \frac{t''_{\mathrm{T}}}{f''_{freq}}\right)\\
\vdash \left\{\begin{array}{l}
I_{\mathrm{T}}.hboxes\left(\bigoplus O_{\mathrm{hboxT}} \to \prod \left|\begin{array}{l} f'''_{freq}\\ M'_{\mathrm{MOIT}}\\ M'_{\mathrm{MOOT}}\\ \frac{H_{\mathrm{OT}}}{M'_{\mathrm{MOOT}},M'_{\mathrm{MOIT}},f'''_{freq},E_{\mathrm{TT}}}\end{array}\right|\right)\\
\frac{N_{\mathrm{MOIT}}}{t'_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOIT}},f'_{freq}},\ \frac{N_{\mathrm{MOOT}}}{t''_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOOT}},f''_{freq}}
\end{array}\right\}\\
\Rightarrow \left\{\frac{\mathscr{N}_{\mathrm{MOOT}}}{t''_{\mathrm{T}},I_{\mathrm{ST}}.M_{\mathrm{MOOT}},f''_{freq}}\left(N_{\mathrm{MOOT}} + \left(\bigoplus N''_{\mathrm{MOOT}}\right)\right)\right\}
\end{array}\right\}} \tag{C.136}$$

The conclusion of rule C.136 deconstructs three objects: the static instance definition of the local harmonic boxes $I_{\mathrm{T}}.hboxes$, the input map $N_{\mathrm{MOI_T}}$ and the output map $N_{\mathrm{MOO_T}}$. We have introduced the n-ary exclusive sum operator $\bigoplus$ to deconstruct these objects further according to an implicit higher-order map function in our semantic rule notation. $N^{'}_{\mathrm{MOO_T}}$ is not one variable, but takes on as many values as there are keys in the map. The premise is instantiated once with each variable. The results are recombined using the same $\bigoplus$ operator in the implicand of the conclusion. The notation is interpreted differently depending on whether the operand of the $\bigoplus$ symbol is a single value or a function. In the case of the deconstruction of values, it may only be a function, and this makes both the key and value available to the premises with the bindings shown. On recombination, the reverse operation is possible, but we also use a single operand to show a map that contains multiple bindings: this represents the merging of maps. There is no obvious inverse to this operation, which is why this construction can only appear in an implicand. In the case of the reassembled map in the implicand of the conclusion it appears on the right-hand side of the non-exclusive sum operator. This is because the freshly recombined map only holds new values for memories that were updated for harmonic boxes that were enabled to run on the current time slice. The old values are overwritten by the straight sum, but the unmodified values are left undisturbed.

The first premise of rule C.136 precipitates the invocation of a rule that filters the input memory map of the current logical instance to produce the subset of it that is needed by the harmonic box (the rule is uninteresting and not shown). It matches on the static harmonic box description object H_{OT} in order to do this. There are a number of frequencies involved. f_{freq} is the frequency of the underlying instance. $f^{'}_{freq}$ is the frequency of the logical instance map. $f^{'''}_{freq}$ is the lowest common multiple frequency of the minimal input and output memories needed by the harmonic box in question. $f^{4'}_{freq}$ is the lowest common multiple frequency of the minimal *input* memories needed by the harmonic box in question. The predicates in the environment express the consistency of these relationships.

The second premise invokes rule C.137, which is the rule for executing individual harmonic boxes. The implicant consists of the reduced input map required by the first premise and again features the static harmonic box object. It generates an updated output map of memories used by the harmonic box. This has the frequency $f^{4'}_{freq}$, which is the lowest common multiple of these output memories.

The third premise filters the old output map by the key set for the harmonic box that has just been executed, and then merges this map with the new map of memories, appending new timed data slices to any that are already there.

$$\dfrac{\begin{array}{l}\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f'_{freq},\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{M_{\mathrm{MOI}\top}}{f'_{freq}},\ M_{\mathrm{MV}\top} \\ \vdash \left\{\ \mu\mu i(M_{\mathrm{MV}\top} \leftrightarrow M_{\mathrm{MOI}\top}),\ \dfrac{N_{\mathrm{MOI}\top}}{t'_{\top}, M_{\mathrm{MOI}\top}, f'_{freq}}\ \right\} \Rightarrow \left\{\dfrac{N_{\mathrm{MV}\top}}{M_{\mathrm{MV}\top}}\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ f''_{freq},\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f''_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f'_{freq},\ f_{freq}\ \right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f''_{freq},\ f_{freq}\ \right) \\ \vdash \left\{\ M_{\mathrm{MV}\top},\ M'_{\mathrm{MV}\top},\ \dfrac{H_{\top}}{M'_{\mathrm{MV}\top}, M_{\mathrm{MV}\top}, f_{freq}, E_{\mathrm{T}\top}},\ \dfrac{N_{\mathrm{MV}\top}}{M_{\mathrm{MV}\top}}\ \right\} \\ \Rightarrow \left\{\dfrac{N'_{\mathrm{MV}\top}}{M'_{\mathrm{MV}\top}}\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{T}\top},\ f''_{freq},\ \dfrac{t''_{\top}}{f''_{freq}},\ M'_{\mathrm{MV}\top},\ \dfrac{M_{\mathrm{MOO}\top}}{f''_{freq}} \\ \vdash \left\{\ \mu\mu o\left(M'_{\mathrm{MV}\top} \leftrightarrow M_{\mathrm{MOO}\top}\right),\ \dfrac{N'_{\mathrm{MV}\top}}{M'_{\mathrm{MV}\top}}\ \right\} \Rightarrow \left\{\dfrac{N_{\mathrm{MOO}\top}}{t''_{\top}, M_{\mathrm{MOO}\top}, f''_{freq}}\right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ f''_{freq},\ M'_{\mathrm{MOI}\top},\ M'_{\mathrm{MOO}\top},\ \dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f''_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f'_{freq},\ f_{freq}\ \right),\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(\ f''_{freq},\ f_{freq}\ \right) \\ \vdash \left\{\begin{array}{l} \dfrac{N_{\mathrm{MOI}\top}}{t'_{\top}, I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top}, f'_{freq}} \\ \dfrac{\mathscr{H}_{\mathrm{O}\top}}{M_{\mathrm{MOO}\top}, M_{\mathrm{MOI}\top}, f_{freq}, E_{\mathrm{T}\top}}\left(\begin{array}{l} M_{\mathrm{MV}\top},\ M'_{\mathrm{MV}\top},\ \dfrac{H_{\top}}{M'_{\mathrm{MV}\top}, M_{\mathrm{MV}\top}, f_{freq}, E_{\mathrm{T}\top}} \\ \mu\mu i(M_{\mathrm{MV}\top} \leftrightarrow M_{\mathrm{MOI}\top}),\ \mu\mu o\left(M'_{\mathrm{MV}\top} \leftrightarrow M_{\mathrm{MOO}\top}\right) \end{array}\right) \end{array}\right\} \\ \Rightarrow \left\{\dfrac{N_{\mathrm{MOO}\top}}{t''_{\top}, I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top}, f''_{freq}}\right\} \end{array}\right\}} \tag{C.137}$$

Rule C.137 handles the extraction of a harmonic box binding. We need now to differentiate between the various different subscripts to harmonic box variables. The implicant of the conclusion of rule C.137 deconstructs the harmonic box specification object $H_{O\top}$ to yield an untimed harmonic box binding $H_{U\top}$. This untimed box binding is not semantically aware of OIDs, and so the constructor of the harmonic box specification type $H_{\mathrm{O}\top}$ contains two memory maps by *variable identifier*, one for input ($M_{\mathrm{MV}\top}$) and one for output ($M'_{\mathrm{MV}\top}$), as well as two bijective mappings from OID to variable identifiers. The bijections are denoted by the double-ended arrows. The $\mu\mu$ denotes a bijective map. The maps in this rule are named *i* and *o* to denote input and output memory bindings respectively.

The first premise invokes the conversion of an OID map to one indexed by variables, according to the conversion map *i*. The operation is trivial so we do not provide an explicit rule.

The second premise uses the new variable identifier-indexed memory map to call rule C.138, which unpacks the untimed box binding, executes the untimed box and returns a variable-indexed memory map, suitably updated with any new values. The third premise does the reverse operation of the first, using the other bijective mapping to re-index the output map by OIDs. The result is passed straight to the implicand of the conclusion.

$$\frac{\begin{array}{c}\left\{\begin{array}{l} E_{\mathrm{TT}},\ \frac{A_{\mathrm{BT}}}{E_{\mathrm{TT}}},\ \frac{A'_{\mathrm{BT}}}{E_{\mathrm{TT}}},\ f_{freq},\ \frac{M_{\mathrm{MVT}}}{E_{\mathrm{TT}}},\ \frac{M'_{\mathrm{MVT}}}{E_{\mathrm{TT}}},\ \frac{B_{\mathrm{T}}}{M'_{\mathrm{MVT}},M_{\mathrm{MVT}},f_{freq},A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{T}}} \\ \vdash \left\{ \frac{N_{\mathrm{MVT}}}{M_{\mathrm{MVT}},E_{\mathrm{TT}}} \right\} \Rightarrow \left\{\sigma\left(B_{\mathrm{TT}}.\mathsf{boxDataConvIn}_{\mathsf{Prop}}\left(\frac{N_{\mathrm{MVT}}}{M_{\mathrm{MVT}},E_{\mathrm{TT}}} \right)\right)\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{TT}},\ \frac{A_{\mathrm{BT}}}{E_{\mathrm{TT}}},\ \frac{A'_{\mathrm{BT}}}{E_{\mathrm{TT}}},\ \frac{A_{\mathrm{T}}}{A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{T}}} \vdash \left\{ \frac{X_{\mathrm{AT}}}{A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{TT}}},\ \frac{D_{\mathrm{UMT}}}{A_{\mathrm{BT}},E_{\mathrm{TT}}} \right\} \\ \Rightarrow \left\{\sigma\left(A_{\mathrm{TT}}.\mathsf{exprSem}_{\mathsf{Prop}}\left(\frac{X_{\mathrm{AT}}}{A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{TT}}},\ \frac{D_{\mathrm{UMT}}}{A_{\mathrm{BT}},E_{\mathrm{TT}}} \right)\right)\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{TT}},\ \frac{A_{\mathrm{BT}}}{E_{\mathrm{TT}}},\ \frac{A'_{\mathrm{BT}}}{E_{\mathrm{TT}}},\ f_{freq},\ \frac{M_{\mathrm{MVT}}}{E_{\mathrm{TT}}},\ \frac{M'_{\mathrm{MVT}}}{E_{\mathrm{TT}}},\ \frac{B_{\mathrm{T}}}{M'_{\mathrm{MVT}},M_{\mathrm{MVT}},f_{freq},A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{T}}} \\ \vdash \left\{ \frac{D_{\mathrm{UMT}}}{A'_{\mathrm{BT}},E_{\mathrm{TT}}} \right\} \Rightarrow \left\{\sigma\left(B_{\mathrm{TT}}.\mathsf{boxDataConvOut}_{\mathsf{Prop}}\left(\frac{D_{\mathrm{UMT}}}{A'_{\mathrm{BT}},E_{\mathrm{TT}}} \right)\right)\right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{\mathrm{T}},\ f_{freq},\ \frac{M_{\mathrm{MVT}}}{E_{\mathrm{T}}},\ \frac{M'_{\mathrm{MVT}}}{E_{\mathrm{T}}} \\ \vdash \left\{\begin{array}{l} \frac{\mathscr{H}_{\mathrm{UT}}}{M'_{\mathrm{MVT}},M_{\mathrm{MVT}},f_{freq},E_{\mathrm{T}}}\left(\begin{array}{l} A_{\mathrm{BT}},\ A'_{\mathrm{BT}},\ \frac{A_{\mathrm{T}}}{A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{T}}} \\ \frac{B_{\mathrm{T}}}{M'_{\mathrm{MVT}},M_{\mathrm{MVT}},f_{freq},A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{T}}} \\ \frac{X_{\mathrm{AT}}}{A'_{\mathrm{BT}},A_{\mathrm{BT}},E_{\mathrm{TT}}} \end{array}\right) \\ \frac{N_{\mathrm{MVT}}}{M_{\mathrm{MVT}},E_{\mathrm{T}}} \end{array}\right\} \\ \Rightarrow \left\{ \frac{N'_{\mathrm{MVT}}}{M'_{\mathrm{MVT}},E_{\mathrm{T}}} \right\} \end{array}\right\}} \tag{C.138}$$

Rule C.138 specifies how an untimed box language is invoked from an untimed harmonic box binding. The conclusion implicant matches on two things: the first is the specification of the function over memories that is to be computed by the harmonic box; the second is the set of input memories to which this function will be applied to produce output memories. The harmonic box specification object is unpacked according to its structure defined in the semantic domain. In order, it matches on the input and output data type specification for the expression language, the expression language binding (A), the harmonic box binding (B), and a static semantic object for a program in the language specifed by A with a binding that matches the memory environment. The function fields of the harmonic box specification object are not shown: they are first order functions, rendered as rules in our structural operational semantics. These rules (such as the expression language specification) are higher order, but notationally they appear of the same form. They are invoked by the premises to this rule C.138. We could instead have made this a rule with no premises, and invoked these functions by name directly in the implicand of the conclusion. The effect would be the same, but the latter would have the disadvantage that we would have to embed our method of representing the structural operational semantics in the structural operational semantics themselves, which would leave us without a base case. This is a symptom of defining a specification system that is powerful enough to describe itself, and is a consequence of the Gödel issues we addressed in chapter 2.

The three premises of rule C.138 are chained together. The first premise invokes the conversion function translating coordination language memories into raw data types. The second uses the product of this operation to invoke the expression language, taking the static semantic object as a Curried argument. The third premise converts this back into the form of memories, to be carried to the implicand of the conclusion. It is the predicates contained within A and B which enable all of these processes to be defined as injective functions. The inclusion of B ensures that, by construction, a harmonic box specification can only be created if the signatures of its input and output types in terms of coordination language memories and plain datatypes are convertible using the relevant functions. The final point of interest about rule C.138 is that each of the implicands of the premises is presented as a σ-type, but the predicate of the σ-type has been discarded when the argument is picked up in the implicant of the next premise (or in the case of the last premise, picked up in the implicand of the conclusion). This is because each successive construction does not need to know that the previous rule application generated a result compatible with its allied predicate. If we were to characterize rule C.138 as a predicate and refine the rule to generate a derived σ-type of that predicate, then the fact that each premise-invoked rule had produced a σ-type of its own would be used in constructing the necessary proof for the σ-type of that modified rule.

C.3.5 Nested boxes step

The nested boxes step consists of two sub-rules. Rule C.139 is a map reduction rule, handling how the execution of each nested coordination instance is combined into the map of nested coordination instances for the enclosing instance. Rule C.140 sets up a recursive trace fragment and deconstructs it to return the final coordination state.

An alternative approach to describing the nested case does not rely on delving into the structure of the nested coordination object in order to effect input and output (where an interface of a nested component is exposed by an enclosing component). We can instead pass these values as separate arguments and return values as we descend and ascend the nested structure. In this work, we adopt the latter approach in the Coq formalization because it makes for simpler code, but we take the former approach in the semantic rules shown in this section because it is more concise. Superficially, the latter approach might have the benefit that it would scale better when, instead of nesting HBCL, we mutually recursed with another box language, in which case the structure of the nested object ought to be invisible to HBCL for data-hiding hygiene reasons. However, on closer examination, this is not justified, since an instantiation of HBCL inside another language has a floating ontology relative to HBCL at global scope. As a consequence, inner FIFOs would not be accessible at all in this circumstance: it would be up to the intervening box language to connect together timed HBCL memories from these different ontologies.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \frac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \frac{K_\top}{K_{\mathsf{S}\top}},\ \frac{I_\top}{I_{\mathsf{S}\top},K_\top,K_{\mathsf{S}\top},f_{freq}},\ \frac{t_\top}{f_{freq}} \\ f'_{freq},\ \frac{I'_{\mathsf{S}\top}}{f'_{freq}},\ K_{\mathsf{S}\top},\ \frac{K'_\top}{K'_{\mathsf{S}\top}},\ \frac{I'_\top}{I'_{\mathsf{S}\top},K'_\top,K'_{\mathsf{S}\top},f'_{freq}},\ \frac{t'_\top}{f'_{freq}},\ \frac{t''_\top}{f'_{freq}} \\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_\top}{f_{freq}},\ \frac{t''_\top}{f'_{freq}} \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\frac{t'_\top}{f_{freq}},\ \frac{t''_\top}{f'_{freq}} \right),\ O_{\mathrm{Inst}\top} \\ \mathsf{isNestedInst}_{\mathsf{Prop}}\left(\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \frac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \frac{K_\top}{K_{\mathsf{S}\top}},\ \frac{I_\top}{I_{\mathsf{S}\top},K_\top,K_{\mathsf{S}\top},f_{freq}} \\ f'_{freq},\ \frac{I'_{\mathsf{S}\top}}{f'_{freq}},\ K_{\mathsf{S}\top},\ \frac{K'_\top}{K'_{\mathsf{S}\top}},\ \frac{I'_\top}{I'_{\mathsf{S}\top},K'_\top,K'_{\mathsf{S}\top},f'_{freq}},\ O_{\mathrm{Inst}\top} \end{array} \right) \\ \vdash \left\{ \frac{C_{\mathrm{InnerFIFOs}\top}}{t'_\top,I'_\top,I'_{\mathsf{S}\top},K'_\top,K'_{\mathsf{S}\top},f'_{freq}} \right\} \Rightarrow \left\{ \frac{C_{\mathrm{FIFOs}\top}}{t''_\top,I'_\top,I'_{\mathsf{S}\top},K'_\top,K'_{\mathsf{S}\top},f'_{freq}} \right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \frac{I_{\mathsf{S}\top}}{f_{freq}},\ K_{\mathsf{S}\top},\ \frac{K_\top}{K_{\mathsf{S}\top}},\ \frac{I_\top}{I_{\mathsf{S}\top},K_\top,K_{\mathsf{S}\top},f_{freq}},\ \frac{t_\top}{f_{freq}} \\ \vdash \left\{ \frac{\mu\mathscr{C}_{\mathrm{InnerFIFOs}\top}}{t_\top,I_\top,I_{\mathsf{S}\top},K_\top,K_{\mathsf{S}\top},f_{freq}} \left(\bigoplus O_{\mathrm{Inst}\top} \to C_{\mathrm{InnerFIFOs}\top} \right) \right\} \\ \Rightarrow \left\{ \frac{\mu\mathscr{C}_{\mathrm{FIFOs}\top}}{t_\top,I_\top,I_{\mathsf{S}\top},K_\top,K_{\mathsf{S}\top},f_{freq}} \left(\begin{array}{l} \bigoplus O_{\mathrm{Inst}\top} \to C_{\mathrm{FIFOs}\top}, \\ \mathsf{NestedFIFOsCorrect}_{\mathsf{Prop}}\left(\begin{array}{l} \bigoplus O_{\mathrm{Inst}\top} \to C_{\mathrm{FIFOs}\top} \\ t_\top,\ I_\top,\ I_{\mathsf{S}\top} \\ K_\top,\ K_{\mathsf{S}\top},\ f_{freq} \end{array} \right) \end{array} \right) \right\} \end{array}\right\}} \tag{C.139}$$

The implicant of the conclusion of rule C.139 deconstructs the map of nested coordination objects, using the same notational conventions that we used in rule C.135 when we deconstructed maps of harmonic boxes. The single premise is invoked once for each of these matches, each with a matching pair of a nested instance identifier and the associated nested coordination state. The implicant of the first premise requires a rule that updates a coordination state object in the 'InnerFIFOs' state from the *previous* time slice to one in the 'FIFOs' state of the current time slice. The resulting objects from each invocation of the premise are recombined into a map in the implicand of the conclusion. A propositional component is added to form the σ-type of the result.

$$\frac{\begin{array}{c}\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_{\top}}{K_{\mathrm{S}\top}},\ \dfrac{I_{\top}}{I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t'_{\top}}{f_{freq}} \\ \vdash \left\{ \dfrac{C_{\mathrm{InnerFIFOs}\top}}{t'_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \Rightarrow \left\{ \dfrac{C_{\mathrm{MemFB}\top}}{t'_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ f'_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_{\top}}{K_{\mathrm{S}\top}},\ \dfrac{I_{\top}}{I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t'_{\top}}{f_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}} \\ \dfrac{C_{\mathrm{MemFB}\top}}{t'_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}},\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq},\ f_{freq} \right) \\ \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},\ f'_{freq} \right),\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t'_{\top}}{f'_{freq}},\ \dfrac{t''_{\top}}{f'_{freq}} \right) \\ \vdash \left\{ \dfrac{\mathrm{S}_{\top\varnothing}}{t''_{\top},I_{\mathrm{S}\top}.M_{\mathrm{MOI}\top},f'_{freq}} \right\} \Rightarrow \left\{ \dfrac{Tr_{\mathrm{MemFB}\top}}{C_{\mathrm{memFB}\top},t'_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_{\top}}{K_{\mathrm{S}\top}},\ \dfrac{I_{\top}}{I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t_{\top}}{f_{freq}},\ \dfrac{t'_{\top}}{f_{freq}} \\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t'_{\top}}{f_{freq}},\ \dfrac{t_{\top}}{f_{freq}} \right) \\ \vdash \left\{ \dfrac{Tr_{\mathrm{MemFB}\top}}{C_{\mathrm{MemFB}\top},t'_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \Rightarrow \left\{ \dfrac{C_{\mathrm{FIFOs}\top}}{t_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ f_{freq},\ \dfrac{I_{\mathrm{S}\top}}{f_{freq}},\ K_{\mathrm{S}\top},\ \dfrac{K_{\top}}{K_{\mathrm{S}\top}},\ \dfrac{I_{\top}}{I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}},\ \dfrac{t_{\top}}{f_{freq}},\ \dfrac{t'_{\top}}{f_{freq}} \\ \mathsf{timeNext}_{\mathsf{Prop}}\left(\dfrac{t'_{\top}}{f_{freq}},\ \dfrac{t_{\top}}{f_{freq}} \right) \\ \vdash \left\{ \dfrac{C_{\mathrm{InnerFIFOs}\top}}{t'_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \Rightarrow \left\{ \dfrac{C_{\mathrm{FIFOs}\top}}{t_{\top},I_{\top},I_{\mathrm{S}\top},K_{\top},K_{\mathrm{S}\top},f_{freq}} \right\} \end{array}\right\}} \tag{C.140}$$

Rule C.140 executes a nested instance, which has been left in the ‘inner FIFOs-enabled’ state for the *previous* time slice, bringing it to the FIFOs-enabled state.

The first premise precipitates the execution of local FIFOs. This is the same operation that is called by the third premise of rule C.133, producing a new coordination state object in the ‘MEMFB-enabled’ state. This object still belongs to the previous time slice. The second premise uses this object to set up a fragment of a new trace, with an empty input stream. This trace will run for one slice (it will terminate on the next FIFO step for lack of input). The premise thus calls the whole trace-building structure by mutual recursion. The third premise deconstructs the new trace object to extract the FIFO-enabled coordination state associated with the terminal object in that trace. This object reflects the state of the nested coordination state after the execution of its own boxes step, so the time slice of the new FIFO-enabled object is that of the current time slice of the calling premise. The result is used directly in the implicand of the conclusion.

C.3.6 MemBF step

The execution of box-FIFO memories is exactly like that of the FIFO-box case, except that boxes and FIFOS are transposed.

$$\frac{\left\{ \begin{array}{l} E_{\mathrm{TT}}, f_{freq}, f'_{freq}, \frac{I_{\mathrm{ST}}}{f_{freq}}, K_{\mathrm{ST}}, \frac{K_{\mathrm{T}}}{K_{\mathrm{ST}}}, \frac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}}, \frac{t_{\mathrm{T}}}{f_{freq}}, \frac{t'_{\mathrm{T}}}{f'_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq}, f_{freq} \right), \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOOT}}, f'_{freq} \right) \\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\mathrm{T}}}{f_{freq}}, \frac{t'_{\mathrm{T}}}{f'_{freq}} \right) \\ \vdash \left\{ \frac{N_{\mathrm{MOOT}}}{t_{\mathrm{T}}, I_{\mathrm{ST}}.M_{\mathrm{MOOT}}, f'_{freq}} \right\} \Rightarrow \left\{ \frac{N'_{\mathrm{MOOT}}}{t'_{\mathrm{T}}, I_{\mathrm{ST}}.M_{\mathrm{MOOT}}, f'_{freq}} \right\} \end{array} \right\}}{\left\{ \begin{array}{l} E_{\mathrm{TT}}, f_{freq}, f'_{freq}, \frac{I_{\mathrm{ST}}}{f_{freq}}, K_{\mathrm{ST}}, \frac{K_{\mathrm{T}}}{K_{\mathrm{ST}}}, \frac{I_{\mathrm{T}}}{I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}}, \frac{t_{\mathrm{T}}}{f_{freq}}, \frac{t'_{\mathrm{T}}}{f'_{freq}} \\ \mathsf{freqDivides}_{\mathsf{Prop}}\left(f'_{freq}, f_{freq} \right), \mathsf{allFreqsDivide}_{\mathsf{Prop}}\left(I_{\mathrm{ST}}.M_{\mathrm{MOOT}}, f'_{freq} \right) \\ \mathsf{timeEq}_{\mathsf{Prop}}\left(\frac{t_{\mathrm{T}}}{f_{freq}}, \frac{t'_{\mathrm{T}}}{f'_{freq}} \right) \\ \vdash \left\{ \frac{\mathscr{C}_{\mathrm{memBFT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \left(N_{\mathrm{MOIT}}, N_{\mathrm{MOOT}}, \mu C_{\mathrm{FIFOsT}} \right) \right\} \\ \Rightarrow \left\{ \frac{\mathscr{C}_{\mathrm{FIFOsT}}}{t_{\mathrm{T}},I_{\mathrm{T}},I_{\mathrm{ST}},K_{\mathrm{T}},K_{\mathrm{ST}},f_{freq}} \left(\begin{array}{l} N_{\mathrm{MOIT}}, N'_{\mathrm{MOOT}}, \mu C_{\mathrm{MemBFT}}, \\ \mathsf{FIFOsCorrect}_{\mathsf{Prop}} \left(\begin{array}{l} N_{\mathrm{MOIT}} \\ N'_{\mathrm{MOOT}} \\ \mu C_{\mathrm{FIFOsT}} \\ t'_{\mathrm{T}}, I_{\mathrm{T}}, I_{\mathrm{ST}}, K_{\mathrm{T}}, K_{\mathrm{ST}}, f_{freq} \end{array} \right) \end{array} \right) \right\} \end{array} \right\}} \tag{C.141}$$

The operation of rule C.141 is almost identical to that of rule C.134, except that inputs are swapped with outputs, and unlike rule C.134, both the premises and conclusion of the rule now concern the same time slice. Otherwise, *mutatus mutandis*, they are the same.

C.4 Expression language dynamic semantics

We now give operational semantics of our simple Boolean expression language for HBCL in the same style as appendix C.3. Details of the expression costings are omitted for clarity.

C.4.1 Expression static semantic object and argument match

Rule C.142 extracts the function closure from the static semantic object containing the main expression, and invokes it by binding values for its arguments to the variable environment. The main function is called '`main`', and v_{varid} is constrained to take this value. This rule handles the binding of our example expression language to the coordination language; the former is a first order object in the context of the latter.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ E'_{\mathrm{T}\top},\ \frac{T_\top}{\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{T'_\top}{\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{c_\top}{T_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{c'_\top}{T'_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \\ \frac{R_\top}{\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \\ \vdash \left\{ v_{varid},\ \frac{W_{\mathrm{Clos}\top}}{R_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{D_\top}{T_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \right\} \\ \Rightarrow \left\{ \frac{D'_\top}{T'_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ A_{\mathrm{BitLangB}\top},\ A'_{\mathrm{BitLangB}\top} \\ \vdash \left\{\begin{array}{l} \frac{\mathscr{X}_{\mathrm{ABitLang}\top}}{A'_{\mathrm{BitLangB}\top},A_{\mathrm{BitLangB}\top},E_{\mathrm{T}\top}}\left(\begin{array}{l} E'_{\mathrm{T}\top},\ \frac{T_\top}{\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{T'_\top}{\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \\ \frac{c_\top}{T_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{c'_\top}{T'_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \\ v_{varid},\ \frac{R_\top}{\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ \frac{W_{\mathrm{Clos}\top}}{R_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \end{array}\right) \\ \frac{D_\top}{T_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top} \end{array}\right\} \\ \Rightarrow \left\{ \sigma\left(\frac{D'_\top}{T'_\top,\left(E_{\mathrm{T}\top}+E'_{\mathrm{T}\top}\right).U_\top},\ A_{\mathrm{T}\top}.\mathsf{exprSem}_{\mathsf{Prop}}\left(\frac{X_{\mathrm{A}\top}}{A'_{\mathrm{B}\top},A_{\mathrm{B}\top},E_{\mathrm{T}\top}},\ \frac{D_{\mathrm{UM}\top}}{A_{\mathrm{B}\top},E_{\mathrm{T}\top}} \right) \right) \right\} \end{array}\right\}} \tag{C.142}$$

The implicant of the conclusion of rule C.142 matches on two arguments: the first is a static semantic object; the second is the piece of data that is the argument to the main function specified by this object. The piece of data is no longer a subscripted 'UM', which denotes data dependent in a box binding, but is instead dependent in a plain type variable: the type variable is directly inferred from the binding A_B. This shifting of dependent arguments is possible and practical because we have constrained our dependent types to be defined only in terms of parametrized σ-types. The type-casting happens purely in the type of propositions: the validity of the cast can be established by a lemma in the context of the rule, and thus the conversion has no computational content. This approach also avoids the need to consider weak (non-Leibniz) equality definitions for dependent types. When we deconstruct the static semantic object in the implicant of the conclusion, we expose the objects needed to invoke a function in our expression language. The type environment is given in a binary form, with the '+' symbol functioning as a union operator over the enclosing type environment and any types defined at the scope of the expression language. The plus is not circled because it is a non-exclusive operation: local bindings to type identifiers overwrite ones from the environment. T is constrained (in the construction of the static semantic object X) to match the type mandated by A_B, and likewise T' and A'_B. The cost variables c and c' are, respectively, the cost functions giving the minimum computational potential available with the input type and the maximum allowable potential remaining after a computation (in the spirit of Tarjan [176]). Our cost function is a stub implementation using an uninteresting flat function of the type, counting function invocations and construction operations: we omit the formal semantics for this.

The implicant of the first premise precipitates the calling of the function execution rule, giving the name of the function, a definition closure, and the piece of data which is to be applied to the function. The result immediately enters the conclusion of the rule, equipped with a predicate asserting that the returned data realises the semantics of the expression language A. We describe the function of R and W variables with the next rule.

C.4.2 Function invocation

The function invocation rule (rule C.143) handles the binding of a function's argument in a new value and function definition closure, and the subsequent reduction of the resulting expression, which is now closed under its operands.

$$\frac{\begin{array}{l}\left\{\begin{array}{l} E_{\mathrm{TT}}, \frac{R_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c_{\mathrm{T}}}{T_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{Y_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} \\ \left(\frac{Y_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} = \frac{\mathscr{Y}_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\left(\frac{T_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right) \\ \left(\frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} = \frac{\mathscr{Y}_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\left(\frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right) \\ \vdash \left\{ v_{varid}, \frac{W_{\mathrm{Clos}\mathrm{T}}}{R_{\mathrm{T}},E_{\mathrm{TT}}} \right\} \\ \Rightarrow \left\{ \frac{R''_{\mathrm{T}}}{E_{\mathrm{TT}}}, \frac{W_{\mathrm{Clos}\mathrm{T}}}{R''_{\mathrm{T}},E_{\mathrm{TT}}},, \frac{\mathscr{Z}_{\mathrm{T}}}{Y_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\left(v'_{varid}, \frac{K_{\mathrm{T}}}{Y'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}},\left(\frac{R_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}+\left(v'_{varid}\rightarrow\frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right),E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{TT}}, \frac{R_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c_{\mathrm{T}}}{T_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{Y_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} \\ \left(\frac{Y_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} = \frac{\mathscr{Y}_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\left(\frac{T_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right) \\ \left(\frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} = \frac{\mathscr{Y}_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\left(\frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right), v'_{varid} \\ \vdash \left\{\begin{array}{l} \frac{\mathscr{W}_{\mathrm{Clos}\mathrm{T}}}{\left(\frac{R''_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}+\left(v'_{varid}\rightarrow\frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right)}\left(\begin{array}{l} E_{\mathrm{TT}}, \frac{R''_{\mathrm{T}}}{E_{\mathrm{TT}}} \\ \left(v'_{varid} \rightarrow \frac{\mathscr{Z}_{\mathrm{T}}}{Y'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\left(\frac{D_{\mathrm{T}}}{T_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right) \\ \frac{W''_{\mathrm{Clos}\mathrm{T}}}{R''_{\mathrm{T}},E_{\mathrm{TT}}} \end{array}\right) \\ \frac{K_{\mathrm{T}}}{Y'_{\mathrm{T}},\left(\frac{R''_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}+\left(v'_{varid}\rightarrow\frac{Y'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}\right)\right),E_{\mathrm{TT}}.U_{\mathrm{T}}} \end{array}\right\} \\ \Rightarrow \left\{\frac{D'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{\mathrm{TT}}, \frac{T_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{T'_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c_{\mathrm{T}}}{T_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{c'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{R_{\mathrm{T}}}{E_{\mathrm{TT}}.U_{\mathrm{T}}} \\ \vdash \left\{ v_{varid}, \frac{W_{\mathrm{Clos}\mathrm{T}}}{R_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}, \frac{D_{\mathrm{T}}}{T_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}} \right\} \\ \Rightarrow \left\{\frac{D'_{\mathrm{T}}}{T'_{\mathrm{T}},E_{\mathrm{TT}}.U_{\mathrm{T}}}\right\} \end{array}\right\}} \tag{C.143}$$

Examining the environment of the conclusion, we have the familiar type environment, two types (input and output) for the function, their associated costs and a record environment. The record environment R is the environment that maps variable identifiers to value and function *declarations* in the current closure W_{Clos}, in whose scope the function is being executed. The rule matches on the name of the variable referring to the function being invoked, the closure of bindings in which the function will be found, and the data value to be passed to this function.

The first premise concerns the result of a look-up operation as described in the next two rules. Z is the function definition that is found. It is deconstructed in the implicand for use in the next premise. It contains the name of the function's free variable v_{varid}'

and the expression that is valid in the new environment. Z is dependent in the required argument and return types specified by Y. Syntactically, there could be four matches on the function and variable object: that for a built-in constant or function, or a user-defined variable or function. Predicates should ensure that a function will never be looked up with an identifier that is actually assigned to a plain variable. This makes certain that interpreter fixpoints realizing this rule can be well-founded. We omit treatment of built-in functions, as they implement the basic Boolean operations of the primitive truth tables suggested by their names.

The expression K is dependent in a type environment to which the argument of the function has been added as a bound variable. The data type is wrapped in a new primed type declaration object. The change to the type environment is indicated by the '+' sign after the record environment (the new closure masks any old binding) and the new mapping from the primed variable identifier to the new primed type description.

The implicand of the first premise also contains a new (double-primed) type declaration environment and definition closure that is applicable to the scope at which the definition was found.

The second premise specifies how the value of the function argument is bound in a new definition closure so that the expression can be matched by the reduction machinery. A new closure is constructed from the old closure W_{Clos} and a new record environment containing just the new binding from the primed variable identifier to the data value, wrapped in the type of definition of the *new* type (the Y in the dependent type is now primed). The double-primed record object R is present in order to index the double-primed closure's dependent type. The expression reduction that is triggered by this second premise takes place in this newly constructed definition closure. The result of the expression reduction that is occasioned by this premise is passsed straight to the implicand of the conclusion.

C.4.3 Function and data resolution

Rule C.144 and rule C.145 resolve a function or simple data object from the definition closure. The closure is a list of environments, and the closest one to the head of the list that contains the function or variable with the required variable name is the one that is matched.

Rule C.144 is the base case in which the value is found in the local mapping. Rule C.145 handles the recursive case.

Predicates prescribe what must and must not be present in maps, although we elide them here. These predicates would provide the means to prove that a fixpoint realizing the look-up in an interpreter is primitively recursive (in the structure of the predicate), and therefore terminates.

$$\frac{}{\left\{\begin{array}{l} E_{\mathrm{T}\top}, \frac{R_\top}{E_{\mathrm{T}\top}.U_\top}, \frac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{ v_{varid}, \frac{\mathscr{W}_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}\left(E'_{\mathrm{T}\top}, \frac{R'_\top}{E'_{\mathrm{T}\top}}, \frac{W'_{\mathrm{Clos}\top}}{R'_\top,E'_{\mathrm{T}\top}}, \left(v_{varid} \rightarrow \frac{Z_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\right)\right)\right\} \\ \Rightarrow \left\{\frac{R_\top}{E_{\mathrm{T}\top}}, \frac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}, \frac{Z_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\right\}\end{array}\right\}} \tag{C.144}$$

In rule C.144, we see in the implicant of the conclusion that the matched variable identifier v_{varid} is present in the local definition map, bound to$Z$. Z is therefore returned immediately in the implicand. Being a base case, there is no need for any premises. We also return the record and closure environments for the scope at which the function or data object was found. If we did not do this, but instead tried to reduce an expression that was defined in the scope from which the look-up is performed, the bindings might not match. This occurs because outer scopes mask the inner static look-up scopes at which particular functions or expressions were defined.

$$\frac{\left\{\begin{array}{l} \left(E_{\mathrm{T}\top} + E'_{\mathrm{T}\top}\right), \frac{R'_\top}{E_{\mathrm{T}\top}.U_\top}, \frac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{ v_{varid}, \frac{R'_\top}{E'_{\mathrm{T}\top}}, \frac{W'_{\mathrm{Clos}\top}}{R'_\top,E'_{\mathrm{T}\top}} \right\} \\ \Rightarrow \left\{\frac{R''_\top}{E_{\mathrm{T}\top}}, \frac{W_{\mathrm{Clos}\top}}{R''_\top,E_{\mathrm{T}\top}}, \frac{Z_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\right\}\end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top}, \frac{R_\top}{E_{\mathrm{T}\top}.U_\top}, \frac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{ v_{varid}, \frac{\mathscr{W}_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}\left(\begin{array}{l} E'_{\mathrm{T}\top}, \frac{R'_\top}{E'_{\mathrm{T}\top}}, \frac{W'_{\mathrm{Clos}\top}}{R'_\top,E'_{\mathrm{T}\top}} \\ \bigoplus\left(v'_{varid} \rightarrow \frac{Z'_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\right), \\ \\ v_{varid} \not= v'_{varid}\end{array}\right)\right\} \\ \Rightarrow \left\{\frac{R''_\top}{E_{\mathrm{T}\top}}, \frac{W_{\mathrm{Clos}\top}}{R''_\top,E_{\mathrm{T}\top}}, \frac{Z_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\right\}\end{array}\right\}} \tag{C.145}$$

Rule C.145 applies to the case where v_{varid} does not occur in the local bindings. This is denoted by using the *n*-ary '$\bigoplus$' operator to show that of all the variables in the map (now shown by a primed v_{varid}), none of them are equal to v_{varid}. This requirement is stated explicitly in the accompanying condition.

Although the binding for v_{varid} has not been found at the local scope, given that W_{Clos} is dependent in R (in which v_{varid} *is* bound), we know that it must be bound by a map before we reach the empty closure. The inner primed closure is thus passed to the premise, which recursively re-invokes the rule, passing the result to the implicand of the conclusion. For the same reason discussed below rule C.144, we see that the returned declaration (record) and definition closures from the premise are passed on. These may differ from all of the records and closures seen at the current scope, hence the new double-primed meta-variables.

C.4.4 Data look-up

Rule C.146 and rule C.147 use the rules of appendix C.4.3 to retrieve a data definition object in the same way that appendix C.4.2 uses the same rules to access function objects in the closure environment. However, in this case, the values of these definition objects are restricted to having simple types.

Rule C.146 covers the case where the value stored is a piece of data. Rule C.147 deals with the case where it is an expression of the correct type that must be reduced first to yield a simple data object. This requires the rules of appendix C.4.5.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{R_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{\ v_{varid},\ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}\ \right\} \\ \Rightarrow \left\{\dfrac{\mathscr{Z}_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\left(\ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\ \right)\right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\ \right)\right) \\ \vdash \left\{\ v_{varid},\ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}\ \right\} \Rightarrow \left\{\dfrac{R'_\top}{E_{\mathrm{T}\top}},\ \dfrac{W_{\mathrm{Clos}\top}}{R'_\top,E_{\mathrm{T}\top}},\ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\right\} \end{array}\right\}} \tag{C.146}$$

In rule C.146, the implicant of the conclusion triggers the piece of data referenced by v_{varid} to be resolved from the environment. The destruction in the implicand of the premise shows that this branch of the rule deals with the case where the piece of data is a plain value, and does not need to be further reduced before it can be returned. It is therefore passed straight to the implicand of the conclusion. The new (primed) closure environment produced by the premise is therefore ignored.

$$\frac{\begin{array}{l} \left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{R_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{\ v_{varid},\ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}\ \right\} \\ \Rightarrow \left\{\dfrac{R'_\top}{E_{\mathrm{T}\top}},\ \dfrac{W_{\mathrm{Clos}\top}}{R'_\top,E_{\mathrm{T}\top}},\ \dfrac{\mathscr{Z}_\top}{Y_\top,E_{\mathrm{T}\top}.U_\top}\left(\ \dfrac{K_\top}{Y_\top,R'_\top,E_{\mathrm{T}\top}.U_\top}\ \right)\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{R'_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\ \right)\right) \\ \vdash \left\{\ \dfrac{W'_{\mathrm{Clos}\top}}{R'_\top,E_{\mathrm{T}\top}},\ \dfrac{K_\top}{Y_\top,R'_\top,E_{\mathrm{T}\top}.U_\top}\ \right\} \\ \Rightarrow \left\{\dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\ \right)\right) \\ \vdash \left\{\ v_{varid},\ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}}\ \right\} \Rightarrow \left\{\dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\right\} \end{array}\right\}} \tag{C.147}$$

Rule C.147 is similar to rule C.146 but instead deals with the situation where the definition found in the implicand of the first premise is an expression, which must be reduced to yield a value. The second premise triggers this operation. It takes the current closure and an expression K, which is dependent in the record environment that matches the dependent argument of that closure, and invokes the expression reduction rule, which we describe next. The expression shares the same type Y as the data type to which it will be reduced. The expression is evaluated in the new closure obtained from the first premise. The result is passed straight from the implicand of the second premise to the implicand of the conclusion.

C.4.5 Expression evaluation

Rule C.148, rule C.149 and rule C.150 deconstruct expressions as a first step in their reduction to concrete values.

Rule C.148 delegates to the pattern rules in appendix C.4.6. Rule C.149 does the same for the constructor rules of appendix C.4.7.

Rule C.150 handles function application. The first premise triggers reduction of the operand (we have a strictly evaluated language); the second invokes the required function, triggering the invocation rule of appendix C.4.2.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top}, \dfrac{R_\top}{E_{\mathrm{T}\top}.U_\top}, \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top}, \dfrac{c_\top}{T_\top, E_{\mathrm{T}\top}.U_\top}, \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top}, \dfrac{c_\top}{T_\top, E_{\mathrm{T}\top}.U_\top}\right)\right) \\ \vdash \left\{\dfrac{W_{\mathrm{Clos}\top}}{R_\top, E_{\mathrm{T}\top}}, \dfrac{Q_\top}{Y_\top, R_\top, E_{\mathrm{T}\top}.U_\top}\right\} \\ \Rightarrow \left\{\dfrac{D_\top}{T_\top, E_{\mathrm{T}\top}.U_\top}\right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top}, \dfrac{R_\top}{E_{\mathrm{T}\top}.U_\top}, \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top}, \dfrac{c_\top}{T_\top, E_{\mathrm{T}\top}.U_\top}, \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top}, \dfrac{c_\top}{T_\top, E_{\mathrm{T}\top}.U_\top}\right)\right) \\ \vdash \left\{\dfrac{W_{\mathrm{Clos}\top}}{R_\top, E_{\mathrm{T}\top}}, \dfrac{\mathscr{K}_\top}{Y_\top, R_\top, E_{\mathrm{T}\top}.U_\top}\left(\dfrac{Q_\top}{Y_\top, R_\top, E_{\mathrm{T}\top}.U_\top}\right)\right\} \\ \Rightarrow \left\{\dfrac{D_\top}{T_\top, E_{\mathrm{T}\top}.U_\top}\right\} \end{array}\right\}} \tag{C.148}$$

Rule C.148 deconstructs the expression K in its implicant. It is the branch that triggers pattern reduction, because the object under the constructor has the pattern meta-variable Q, dependent in the same environment as the enclosing expression. The pattern meta-variable is passed to the premise, along with the definition closure, which the pattern will need in order to dereference a piece of data. The data returned in the implicand is passed back to the conclusion.

$$\frac{\left\{\begin{array}{l} E_{T\top}, \frac{R_\top}{E_{T\top}.U_\top}, \frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}, \frac{Y_\top}{E_{T\top}.U_\top} \\ \left(\frac{Y_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}\right)\right) \\ \vdash \left\{\frac{W_{\mathrm{Clos}\top}}{R_\top,E_{T\top}}, \frac{C_\top}{Y_\top,R_\top,E_{T\top}.U_\top}\right\} \\ \Rightarrow \left\{\frac{D_\top}{T_\top,E_{T\top}.U_\top}\right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{T\top}, \frac{R_\top}{E_{T\top}.U_\top}, \frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}, \frac{Y_\top}{E_{T\top}.U_\top} \\ \left(\frac{Y_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}\right)\right) \\ \vdash \left\{\frac{W_{\mathrm{Clos}\top}}{R_\top,E_{T\top}}, \frac{\mathscr{K}_\top}{Y_\top,R_\top,E_{T\top}.U_\top}\left(\frac{C_\top}{Y_\top,R_\top,E_{T\top}.U_\top}\right)\right\} \\ \Rightarrow \left\{\frac{D_\top}{T_\top,E_{T\top}.U_\top}\right\} \end{array}\right\}} \tag{C.149}$$

The form of rule C.149 is identical to rule C.148, except that instead of dealing with a pattern meta-variable Q, there is now a constructor meta-variable C. The premise now triggers the construction rule rather than the pattern rule.

$$\frac{\begin{array}{c} \left\{\begin{array}{l} E_{T\top}, \frac{R_\top}{E_{T\top}.U_\top}, \frac{T'_\top}{E_{T\top}.U_\top}, \frac{c'_\top}{T'_\top,E_{T\top}.U_\top}, \frac{Y'_\top}{E_{T\top}.U_\top} \\ \left(\frac{Y'_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T'_\top}{E_{T\top}.U_\top}, \frac{c'_\top}{T'_\top,E_{T\top}.U_\top}\right)\right) \\ \vdash \left\{\frac{W_{\mathrm{Clos}\top}}{R_\top,E_{T\top}}, \frac{K'_\top}{Y'_\top,R_\top,E_{T\top}.U_\top}\right\} \\ \Rightarrow \left\{\frac{D'_\top}{T'_\top,E_{T\top}.U_\top}\right\} \end{array}\right\} \\ \left\{\begin{array}{l} E_{T\top}, \frac{T'_\top}{E_{T\top}.U_\top}, \frac{T_\top}{E_{T\top}.U_\top}, \frac{c'_\top}{T'_\top,E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}, \frac{R_\top}{E_{T\top}.U_\top} \\ \vdash \left\{v_{varid}, \frac{W_{\mathrm{Clos}\top}}{R_\top,E_{T\top}.U_\top}, \frac{D'_\top}{T'_\top,E_{T\top}.U_\top}\right\} \\ \Rightarrow \left\{\frac{D_\top}{T_\top,E_{T\top}.U_\top}\right\} \end{array}\right\} \end{array}}{\left\{\begin{array}{l} E_{T\top}, \frac{R_\top}{E_{T\top}.U_\top}, \frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}, \frac{Y_\top}{E_{T\top}.U_\top} \\ \left(\frac{Y_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}\right)\right) \\ \vdash \left\{\frac{W_{\mathrm{Clos}\top}}{R_\top,E_{T\top}}, \frac{\mathscr{K}_\top}{Y_\top,R_\top,E_{T\top}.U_\top}\left(\begin{array}{l} v_{varid}, \frac{T'_\top}{E_{T\top}.U_\top}, \frac{c'_\top}{T'_\top,E_{T\top}.U_\top} \\ \frac{Y'_\top}{E_{T\top}.U_\top}, \frac{K'_\top}{Y'_\top,R_\top,E_{T\top}.U_\top}, \\ \left(\frac{Y'_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T'_\top}{E_{T\top}.U_\top}, \frac{c'_\top}{T'_\top,E_{T\top}.U_\top}\right)\right) \end{array}\right)\right\} \\ \Rightarrow \left\{\frac{D_\top}{T_\top,E_{T\top}.U_\top}\right\} \end{array}\right\}} \tag{C.150}$$

Rule C.150 deals with expressions that specify a function application. The expression constructor in the implicant of the conclusion has a number of arguments, of which two are interesting: v_{varid} is the name of the function; K' is the expression that forms the argument to that function. The other arguments give the type and computational cost of the argument, and constrain equivalence of the Y', which is the union variable function type of the simple type and its cost. The first premise triggers reduction of the expression by recursively invoking the expression evaluation rule. The second premise uses the result of this and the definition closure to trigger the function invocation rule, passing the result straight back to the implicand of the conclusion.

C.4.6 Pattern evaluation

Rule C.151, rule C.152, Rule C.153 and Rule C.154 specify the pattern-matching semantics. Pattern matching proceeds using an index string of identifiers (for records) or tuple indices (for tuples). The rules deconstruct this string to retrieve the correct part of the data structure. The length of the string, whether a tuple or record is expected, and whether the index is valid in the context in which it occurs, are details enforced by predicates, again omitted for clarity. Rule C.151 uses the top-level variable identifier to look up a data value from the environment, using the data look-up rules of appendix C.4.5, before extracting the pattern string. Rule C.152 applies when a tuple is at the head of the pattern string; rule C.153 does so when a record is at the head of the pattern string. Rule C.154 is the recursive base case for pattern resolution: it indicates that the end of the list has been reached, and the data type most recently retrieved should be the same as the data type required by the whole pattern match. The pattern list constructor is empty for such lists, and its two type parameters are constrained to be equal.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{R_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{T'_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}},\ v_{varid} \right\} \Rightarrow \left\{ \dfrac{D'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\} \left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{T'_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{c'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{ \dfrac{J_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T'_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{D'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top} \right\} \\ \Rightarrow \left\{ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{T'_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{c'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right)\right) \\ \vdash \left\{ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}},\ \dfrac{Q_\top}{Y_\top,R_\top,E_{\mathrm{T}\top}.U_\top}\left(v_{varid},\ \dfrac{J_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T'_\top,E_{\mathrm{T}\top}.U_\top} \right) \right\} \\ \Rightarrow \left\{ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}} \tag{C.151}$$

Rule C.151 looks up the data object Q from the supplied definition closure W_{Clos} and passes it to the other pattern matching rules. The implicand of the conclusion deconstructs the only constructor to yield a variable name that will be used to look up the data from the closure, and a pattern string J, which will be used to extract the desired component of this data. The first premise precipitates a match on the data look-up rules, and passes the result to the second premise, along with the pattern list and (primed) data variable of the expected type. The second premise invokes the pattern-matching rules, and the returned data is passed straight to the implicand of the conclusion. J is dependently typed in terms of the type which is being deconstructed (preceded by its type environment) and the type which the pattern should eventually return when the recursive call stack unwinds (preceded by its type environment).

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ E'_{\mathrm{Tn}\top},\ \frac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \frac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \frac{T_{\mathrm{n}\top}}{E'_{\mathrm{Tn}\top}.U_\top},\ \frac{c_{\mathrm{n}\top}}{T_{\mathrm{n}\top},E'_{\mathrm{Tn}\top}.U_\top} \\ \vdash \left\{ \frac{D_{\mathrm{n}\top}}{T_{\mathrm{n}\top},E'_{\mathrm{Tn}\top}.U_\top},\ \frac{J'_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T_{\mathrm{n}\top},E'_{\mathrm{Tn}\top}.U_\top} \right\} \Rightarrow \left\{ \frac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ E'_{\mathrm{Tn}\top},\ \frac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \frac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \frac{T'_\top}{E_{\mathrm{T}\top}.U_\top},\ \frac{c'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top},\ \frac{T_{\mathrm{n}\top}}{E'_{\mathrm{Tn}\top}.U_\top},\ \frac{c_{\mathrm{n}\top}}{T_{\mathrm{n}\top},E'_{\mathrm{Tn}\top}.U_\top} \\ \vdash \left\{\begin{array}{l} \frac{\mathscr{J}_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T'_\top,E_{\mathrm{T}\top}.U_\top}\left(n_{\mathbb{N}},\ \frac{J'_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T_{\mathrm{n}\top},E'_{\mathrm{Tn}\top}.U_\top} \right) \\ \frac{\mathscr{D}_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top}\left(\prod_{\mathrm{i}=0}^{\mathrm{n}}\left(\frac{D_{\mathrm{n}\top}}{T_{\mathrm{n}\top},E'_{\mathrm{Tn}\top}.U_\top} \right) \right) \end{array}\right\} \\ \Rightarrow \left\{ \frac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}} \tag{C.152}$$

Rule C.152 is the recursive case of a tuple pattern match. The pattern is deconstructed in the implicant of the conclusion, yielding an index $n_{\mathbb{N}}$, giving the position of the dereferenced data within the tuple, and a further pattern specifying how this data is in turn to be deconstructed. There is a computational cost associated with this operation, which in the case of our flat cost function is related only to the length of the pattern. The type associated with the data extracted from the tuple is given by T_n. The data is also deconstructed from the tuple, passing only the relevant member indexed n to the premise. The first premise recursively invokes the pattern matching rules. The type of the data now being deconstructed is that of the corresponding n_{th} member of the deconstructed tuple type. The separate type environments with the 'n' subscript cater for the fact that a type inside a tuple may have been declared in a different scope.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ E'_{\mathrm{Tv}\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{T_{\mathrm{v}\top}}{E'_{\mathrm{Tv}\top}.U_\top},\ \dfrac{c_{\mathrm{v}\top}}{T_{\mathrm{v}\top},E'_{\mathrm{Tv}\top}.U_\top} \\ \vdash \left\{ \dfrac{D_{\mathrm{v}\top}}{T_{\mathrm{v}\top},E'_{\mathrm{Tv}\top}.U_\top},\ \dfrac{J'_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T_{\mathrm{v}\top},E'_{\mathrm{Tv}\top}.U_\top} \right\} \Rightarrow \left\{ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ E'_{\mathrm{Tv}\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{T'_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c'_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{T_{\mathrm{v}\top}}{E'_{\mathrm{Tv}\top}.U_\top},\ \dfrac{c_{\mathrm{v}\top}}{T_{\mathrm{v}\top},E'_{\mathrm{Tv}\top}.U_\top} \\ \vdash \left\{ \begin{array}{l} \dfrac{\mathscr{J}_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T'_\top,E_{\mathrm{T}\top}.U_\top}\left(v_{varid},\ \dfrac{J'_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T_{\mathrm{v}\top},E'_{\mathrm{Tv}\top}.U_\top} \right) \\ \dfrac{\mathscr{D}_\top}{T'_\top,E_{\mathrm{T}\top}.U_\top}\left(\bigoplus\left(v_{varid} \to \dfrac{D_{\mathrm{v}\top}}{T_{\mathrm{v}\top},E'_{\mathrm{Tv}\top}.U_\top} \right) \right) \end{array} \right\} \\ \Rightarrow \left\{ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}} \tag{C.153}$$

Rule C.153 has the same structure as rule C.152, except that instead of matching on a natural number index of a tuple, the head of the pattern list matches on a variable identifier v_{varid}, which identifies a member of a record data type. The data in the conclusion's implicant is also deconstructed to access the data referenced by this identifier, which is then passed to the premise. Everything else is the same as the tuple-matching rule, *mutatus mutandis*.

$$\frac{}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \\ \vdash \left\{ \dfrac{D_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{\mathscr{J}_\top}{T_\top,E_{\mathrm{T}\top}.U_\top,T_\top,E_{\mathrm{T}\top}.U_\top}() \right\} \\ \Rightarrow \left\{ \dfrac{D_\top}{T_\top,E'_{\mathrm{T}\top}.U_\top} \right\} \end{array}\right\}} \tag{C.154}$$

Rule C.154 is the pattern list base case. It is a trivial rule with no premises. Any match on an empty list constructor must have the same type in both the match and return positions, and so the matched data in the implicant is passed straight to the implicand.

C.4.7 Construction evaluation

Rule C.155, rule C.156 and rule C.157 are concerned with building new values.

Rule C.155 deals with the simple case where a base type is being built.

Rule C.156 builds a tuple from a sequence of expressions, which are recursively reduced using rules in appendix C.4.5.

Rule C.157 does the same for records, building a record from an equivalently indexed associative array of expressions.

$$\frac{}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \dfrac{R_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top},\ \dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} \\ \left(\dfrac{Y_\top}{E_{\mathrm{T}\top}.U_\top} = \dfrac{\mathscr{Y}_\top}{E_{\mathrm{T}\top}.U_\top}\left(\dfrac{T_\top}{E_{\mathrm{T}\top}.U_\top},\ \dfrac{c_\top}{T_\top,E_{\mathrm{T}\top}.U_\top} \right)\right) \\ \vdash \left\{ \dfrac{W_{\mathrm{Clos}\top}}{R_\top,E_{\mathrm{T}\top}},\ \dfrac{\mathscr{C}_\top}{Y_\top,R_\top,E_{\mathrm{T}\top}.U_\top}\left(b_{boolconst} \right) \right\} \\ \Rightarrow \left\{ \dfrac{\mathscr{D}_\top}{T_\top,E_{\mathrm{T}\top}.U_\top}\left(b_{boolconst} \right) \right\} \end{array}\right\}} \tag{C.155}$$

Rule C.155 matches on the first of three constructors: the base case, which wraps a piece of Boolean data from the syntactic domain. The implicant of the conclusion matches this syntactic primitive as $b_{boolconst}$, and wraps it with a data constructor in the conclusion, placing it in the semantic domain. The definition closure matched in the implicant is ignored, as no values from the environment are needed.

$$\frac{\left\{ \begin{array}{l} E_{T\top}, \frac{R_\top}{E_{T\top}.U_\top}, \frac{T'_{n\top}}{E'_{T\top}.U_\top}, \frac{c'_{n\top}}{T'_{n\top},E'_{T\top}.U_\top}, \frac{Y'_\top}{E_{T\top}.U_\top} \\ \left(\frac{Y'_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T'_{n\top}}{E_{T\top}.U_\top}, \frac{c'_{n\top}}{T'_{n\top},E_{T\top}.U_\top}\right)\right) \\ \vdash \left\{ \frac{W_{Clos\top}}{R_\top,E_{T\top}}, \frac{K_{n\top}}{Y'_\top,R_\top,E_{T\top}.U_\top} \right\} \\ \Rightarrow \left\{ \frac{D_{n\top}}{T'_{n\top},E_{T\top}.U_\top} \right\} \end{array} \right\}}{\left\{ \begin{array}{l} E_{T\top}, \frac{R_\top}{E_{T\top}.U_\top}, \frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}, \frac{T'_{n\top}}{E_{T\top}.U_\top}, \frac{c'_{n\top}}{T'_{n\top},E_{T\top}.U_\top}, \frac{Y_\top}{E_{T\top}.U_\top}, \frac{Y'_\top}{E_{T\top}.U_\top} \\ \left(\frac{Y_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T_\top}{E_{T\top}.U_\top}, \frac{c_\top}{T_\top,E_{T\top}.U_\top}\right)\right) \\ \left(\frac{Y'_\top}{E_{T\top}.U_\top} = \frac{\mathscr{Y}_\top}{E_{T\top}.U_\top}\left(\frac{T'_{n\top}}{E_{T\top}.U_\top}, \frac{c'_{n\top}}{T'_{n\top},E_{T\top}.U_\top}\right)\right) \\ \vdash \left\{ \frac{W_{Clos\top}}{R_\top,E_{T\top}}, \frac{\mathscr{C}_\top}{Y_\top,R_\top,E_{T\top}.U_\top}\left(\prod_{i=0}^{n}\left(\frac{K_{n\top}}{Y'_\top,R_\top,E_{T\top}.U_\top} \right)\right) \right\} \\ \Rightarrow \left\{ \frac{\mathscr{D}_\top}{T_\top,E_{T\top}.U_\top}\left(\prod_{i=0}^{n}\left(\frac{D_{n\top}}{T'_{n\top},E_{T\top}.U_\top} \right)\right) \right\} \end{array} \right\}} \tag{C.156}$$

Rule C.156 constructs a tuple of values. The implicant of the conclusion matches on the definition closure and the tuple branch of the constructor for specifying the construction of tuple expressions. This constructor takes as its single argument the *n*-ary Cartesisan product of expressions, which are combined according to their index to build a data tuple of type *T* in the conclusion. The premise is invoked *n* times, once for each expression in the tuple. Each such premise precipitates a recursive match on the expression-matching rule.

$$\frac{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \frac{R_{\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{T'_{\mathrm{v}\top}}{E'_{\mathrm{T}\top}.U_{\top}},\ \frac{c'_{\mathrm{v}\top}}{T'_{\mathrm{v}\top},E'_{\mathrm{T}\top}.U_{\top}},\ \frac{Y'_{\top}}{E_{\mathrm{T}\top}.U_{\top}} \\ \left(\frac{Y'_{\top}}{E_{\mathrm{T}\top}.U_{\top}} = \frac{\mathscr{Y}_{\top}}{E_{\mathrm{T}\top}.U_{\top}}\left(\frac{T'_{\mathrm{v}\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{c'_{\mathrm{v}\top}}{T'_{\mathrm{v}\top},E_{\mathrm{T}\top}.U_{\top}}\right)\right) \\ \vdash \left\{\frac{W_{\mathrm{Clos}\top}}{R_{\top},E_{\mathrm{T}\top}},\ \frac{K_{\mathrm{v}\top}}{Y'_{\top},R_{\top},E_{\mathrm{T}\top}.U_{\top}}\right\} \\ \Rightarrow \left\{\frac{D_{\mathrm{v}\top}}{T'_{\mathrm{v}\top},E_{\mathrm{T}\top}.U_{\top}}\right\} \end{array}\right\}}{\left\{\begin{array}{l} E_{\mathrm{T}\top},\ \frac{R_{\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{T_{\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{c_{\top}}{T_{\top},E_{\mathrm{T}\top}.U_{\top}},\ \frac{T'_{\mathrm{v}\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{c'_{\mathrm{v}\top}}{T'_{\mathrm{v}\top},E_{\mathrm{T}\top}.U_{\top}},\ \frac{Y_{\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{Y'_{\top}}{E_{\mathrm{T}\top}.U_{\top}} \\ \left(\frac{Y_{\top}}{E_{\mathrm{T}\top}.U_{\top}} = \frac{\mathscr{Y}_{\top}}{E_{\mathrm{T}\top}.U_{\top}}\left(\frac{T_{\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{c_{\top}}{T_{\top},E_{\mathrm{T}\top}.U_{\top}}\right)\right) \\ \left(\frac{Y'_{\top}}{E_{\mathrm{T}\top}.U_{\top}} = \frac{\mathscr{Y}_{\top}}{E_{\mathrm{T}\top}.U_{\top}}\left(\frac{T'_{\mathrm{v}\top}}{E_{\mathrm{T}\top}.U_{\top}},\ \frac{c'_{\mathrm{v}\top}}{T'_{\mathrm{v}\top},E_{\mathrm{T}\top}.U_{\top}}\right)\right) \\ \vdash \left\{\frac{W_{\mathrm{Clos}\top}}{R_{\top},E_{\mathrm{T}\top}},\ \frac{\mathscr{C}_{\top}}{Y_{\top},R_{\top},E_{\mathrm{T}\top}.U_{\top}}\left(\bigoplus v_{varid} \rightarrow \left(\frac{K_{\mathrm{v}\top}}{Y'_{\top},R_{\top},E_{\mathrm{T}\top}.U_{\top}}\right)\right)\right\} \\ \Rightarrow \left\{\frac{\mathscr{D}_{\top}}{T_{\top},E_{\mathrm{T}\top}.U_{\top}}\left(\bigoplus v_{varid} \rightarrow \frac{D_{\mathrm{v}\top}}{T'_{\mathrm{v}\top},E_{\mathrm{T}\top}.U_{\top}}\right)\right\} \end{array}\right\}} \tag{C.157}$$

Rule C.157 is very similar to rule C.156, but instead deals with building a record from an associative array of expressions. We use the now familiar notation for deconstructing maps under an n-ary exclusive sum. Exchanging tuple indices for variable names bound to v_{varid}, the record rule has the same form as the tuple one, *mutatus mutandis*.

Appendix D

Further Harmonic Box Coordination Language formalization in Coq

D.1 Module types

D.1.1 Absract type of identifiers

Listing D.1: The identifier module signature

```
Require Import Coq.Structures.Equalities.
Require Coq.FSets.FMapInterface.
Require Import HBCL.Util.FMapRawIface.

Module Type Ids.

  Parameter Id : Set.

  Declare Module IdDecidable : DecidableTypeFull
  with Definition t := Id.

End Ids.

Module Type IdPredType(idmod : Ids).

  Parameter Pred : idmod.Id → Prop.

End IdPredType.

Module Type IdsPred (idmod : Ids)(idpmod : IdPredType idmod).

  Definition PredID := sig idpmod.Pred.

  Declare Module PredidDecidable : DecidableTypeFull
    with Definition t := sig idpmod.Pred.

  Axiom IdPredRelPirrel : ∀ x y p q,
      idmod.IdDecidable.eq x y →
      PredidDecidable.eq (exist idpmod.Pred x p) (exist idpmod.Pred y q).

End IdsPred.

Module Type VaridPredType(idmod : Ids) := IdPredType idmod.

Module Type Varid(idmod : Ids)(vpt : VaridPredType idmod)
  := IdsPred idmod vpt.

Module Type TypidPredType(idmod : Ids) := IdPredType idmod.

Module Type Typid(idmod : Ids)(vpt : TypidPredType idmod)
  := IdsPred idmod vpt.

Module Type BoxidPredType(idmod : Ids) := IdPredType idmod.

Module Type Boxid(idmod : Ids)(vpt : BoxidPredType idmod)
  := IdsPred idmod vpt.

Module Type IdPreds.
```

```
  Declare Module ids : Ids.
  Declare Module varidPredType : VaridPredType ids.
  Declare Module varidPred : Varid ids varidPredType.
  Declare Module typidPredType : TypidPredType ids.
  Declare Module typidPred : Typid ids typidPredType.
  Declare Module boxidPredType : BoxidPredType ids.
  Declare Module boxidPred : Boxid ids boxidPredType.
  Definition Varid := varidPred.PredID.
  Definition Typid := typidPred.PredID.
  Definition Boxid := boxidPred.PredID.
  Require Import Coq.FSets.FMapWeakList.
  Declare Module VaridMapModRaw : FMapIfaceRaw varidPred.PredidDecidable
    with Definition t := fun elt ⇒ list (Varid × elt).
  Declare Module VaridMapModPred : FMapModDatImplPred varidPred.PredidDecidable
    VaridMapModRaw.
  Declare Module VaridMapMod :
    FMapIfaceRF varidPred.PredidDecidable
    VaridMapModRaw VaridMapModPred with Definition key := Varid
  with Module Raw := VaridMapModRaw.
End IdPreds.
```

D.1.2 Absract type of OIDs

Listing D.2: The OID module signature

```
Require Import Coq.Structures.Equalities.
Require Coq.FSets.FMapInterface.
Require Import HBCL.Util.FMapRawIface.
Module Type Oids.
  Parameter Oid : Set.
  Declare Module OidDecidable : DecidableTypeFull
  with Definition t := Oid.
  Parameter OidLength : Oid → nat.
End Oids.
Module Type OidPredType(oidmod : Oids).
  Parameter Pred : oidmod.Oid → Prop.
End OidPredType.
Module Type OidsPred (oidmod : Oids)(oidpt : OidPredType oidmod).
  Definition PredOid := sig oidpt.Pred.
  Declare Module PredoidDecidable : DecidableTypeFull
  with Definition t := PredOid.
End OidsPred.
Module Type OidUTPredType(oidmod : Oids) := OidPredType oidmod.
Module Type OidUTPred(oidmod : Oids)(utpt : OidUTPredType oidmod) :=
  OidsPred oidmod utpt.
Module Type OidTTPredType(oidmod : Oids) := OidPredType oidmod.
Module Type OidTTPred(oidmod : Oids)(ttpt : OidTTPredType oidmod) :=
  OidsPred oidmod ttpt.
Module Type OidMemBFPredType(oidmod : Oids) := OidPredType oidmod.
Module Type OidMemBFPred(oidmod : Oids)(mbfpt : OidMemBFPredType oidmod) :=
  OidsPred oidmod mbfpt.
Module Type OidMemFBPredType(oidmod : Oids) := OidPredType oidmod.
Module Type OidMemFBPred(oidmod : Oids)(mfbpt : OidMemFBPredType oidmod) :=
  OidsPred oidmod mfbpt.
```

```
Module Type OidLInstPredType(oidmod : Oids) := OidPredType oidmod.

Module Type OidLInstPred(oidmod : Oids)(lipt : OidLInstPredType oidmod) :=
  OidsPred oidmod lipt.

Module Type OidLLibPredType(oidmod : Oids) := OidPredType oidmod.

Module Type OidLLibPred(oidmod : Oids)(llpt : OidLLibPredType oidmod) :=
  OidsPred oidmod llpt.

Module Type OidPreds.

  Declare Module oids : Oids.
  Declare Module oidUTPredType : OidUTPredType oids.
  Declare Module oidUTPred : OidUTPred oids oidUTPredType.
  Declare Module oidTTPredType : OidTTPredType oids.
  Declare Module oidTTPred : OidTTPred oids oidTTPredType.
  Declare Module oidMemBFPredType : OidMemBFPredType oids.
  Declare Module oidMemBFPred : OidMemBFPred oids oidMemBFPredType.
  Declare Module oidMemFBPredType : OidMemFBPredType oids.
  Declare Module oidMemFBPred : OidMemFBPred oids oidMemFBPredType.
  Declare Module oidLInstPredType : OidLInstPredType oids.
  Declare Module oidLInstPred : OidLInstPred oids oidLInstPredType.
  Declare Module oidLLibPredType : OidLLibPredType oids.
  Declare Module oidLLibPred : OidLLibPred oids oidLLibPredType.

  Definition HBCL_OidUT := oidUTPred.PredOid.
  Definition HBCL_OidTT := oidTTPred.PredOid.
  Definition HBCL_OidLLib := oidLLibPred.PredOid.
  Definition HBCL_OidLInst := oidLInstPred.PredOid.
  Definition HBCL_OidMemBF := oidMemBFPred.PredOid.
  Definition HBCL_OidMemFB := oidMemFBPred.PredOid.

  Declare Module OidMapMod(opt : OidPredType oids)
    (op : OidsPred oids opt) : FMapInterface.WSfun op.PredoidDecidable.

  Declare Module LInstMapModRaw : FMapIfaceRaw oidLInstPred.PredoidDecidable
    with Definition t := fun elt ⇒ list (HBCL_OidLInst × elt).

  Declare Module LInstMapModPred :
    FMapModDatImplPred oidLInstPred.PredoidDecidable LInstMapModRaw.

  Declare Module LInstMapMod :
    FMapIfaceRF oidLInstPred.PredoidDecidable
    LInstMapModRaw LInstMapModPred with Definition key := HBCL_OidLInst
  with Module Raw := LInstMapModRaw.

  Declare Module LLibMapModRaw : FMapIfaceRaw oidLLibPred.PredoidDecidable
    with Definition t := fun elt ⇒ list (HBCL_OidLLib × elt).

  Declare Module LLibMapModPred :
    FMapModDatImplPred oidLLibPred.PredoidDecidable LLibMapModRaw.

  Declare Module LLibMapMod :
    FMapIfaceRF oidLLibPred.PredoidDecidable
    LLibMapModRaw LLibMapModPred with Definition key := HBCL_OidLLib
  with Module Raw := LLibMapModRaw.

  Parameter liblessInst : HBCL_OidLInst → Prop.

  Parameter instLessMemBF : HBCL_OidMemBF → Prop.

  Parameter instLessMemFB : HBCL_OidMemFB → Prop.


  Parameter concatLibInst : HBCL_OidLLib → HBCL_OidLInst →
     HBCL_OidLInst.

  Parameter concatInstMemBF : sig liblessInst → HBCL_OidMemBF →
     HBCL_OidMemBF.

  Parameter concatInstMemFB : sig liblessInst → HBCL_OidMemFB →
     HBCL_OidMemFB.

  Parameter splitLInstOid : ∀ (i : HBCL_OidLInst),
    sig liblessInst + (HBCL_OidLLib × HBCL_OidLInst).

  Parameter splitLiblessLInstOid : sig liblessInst →
    sig liblessInst + (sig liblessInst × sig liblessInst).

  Parameter splitMemBFOid : ∀ (m : HBCL_OidMemBF),
    sig instLessMemBF + (sig liblessInst × HBCL_OidMemBF).
  Parameter splitMemFBOid : ∀ (m : HBCL_OidMemFB),
    sig instLessMemFB + (sig liblessInst × HBCL_OidMemFB).
End OidPreds.
```

D.1.3 The untimed (but sized) type system

The structure of the semantics is parametrized on a type system expressed in a module with the signature of Listing D.3.

Listing D.3: The untimed type system module signature

```
Module Type UTypeSys.
  Parameter Size : Type.
  Parameter TypeS : Size → Type.
  Definition ProtoT := sigT TypeS.
  Parameter ProtoEqT : ProtoT → ProtoT → Prop.
  Parameter DataR : Type.
  Parameter DataP : ProtoT → DataR → Prop.
  Definition ProtoU(t : ProtoT) := sig (DataP t).
  Definition UDataPST(t : sigT TypeS) := sig (DataP t).
End UTypeSys.
```

Exploring this module type to OCaml produces uncompilable code, which has to be patched in order to work. The problem seems to be caused by the depth of the dependent typing, which Coq cannot map to OCaml's type system.

D.1.4 The untimed OID type system

Listing D.4: The untimed OID type system module signature

```
Module Type UTypeSysOid(Import uts : UTypeSys)(opm : OidPreds).

  Parameter T : opm.HBCL_OidUT → ProtoT → Type.
  Definition TEq(ut1 ut2 : sigTD T) :=
    opm.oidUTPred.PredoidDecidable.eq (projTD1 ut1) (projTD1 ut2) ∧
    ProtoEqT (projTD2 ut1) (projTD2 ut2).

  Parameter V : ∀ o t, T o t → ProtoU t → Type.
End UTypeSysOid.
```

D.1.5 The timed type system

Any module instantiating the untimed type system module type can be turned into a timed type system using a module functor with the signature of Listing D.5.

Listing D.5: The harmonic type system signature

```
Module Type HTypeSys (Import uts : UTypeSys)(Import opm : OidPreds)
  (UTSparam : UTypeSysOid uts opm).

  Inductive TimedTLocal :
    HBCL_OidTT → Freq → sigTD UTSparam.T → Type :=
  | TType : ∀ (ou : HBCL_OidUT)(ot : HBCL_OidTT)
    (f : Freq)(ut : sigTD UTSparam.T),
    TimedTLocal ot f ut.

  Definition TimedT := TimedTLocal.

  Parameter TimedTEq : sigTT TimedT → sigTT TimedT → Prop.

  Definition TimedTF(f : Freq) := fun o u ⇒ TimedT o f u.

  Definition TimedTFEq (tt1 tt2 : sigTT TimedTF) : Prop.
Admitted.

  Inductive TimedVLocal(ttimed : sigTT TimedT)(ttime : TTime (projTT2 ttimed)) :
    Type :=
    | MakeTimedV(u : option (uts.ProtoU (projTD2 (projTT3 ttimed)))) :

        TimedVLocal ttimed ttime.

  Definition TimedV := TimedVLocal.

  Definition TimedVF(f : Freq) := sigTD (TimedTF f) → TTime f → Type.

End HTypeSys.
```

D.1.6 The untimed box abstraction

Listing D.6: The untimed box signature

```
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.ModSignatures.Ids.
Require Import HBCL.HBCL_0_1.ModSignatures.Oids.
Require Import HBCL.HBCL_0_1.ModSignatures.UTypeSys.
Require Import HBCL.HBCL_0_1.ModSignatures.UTypeSysOid.
Require Import HBCL.HBCL_0_1.ModSignatures.UCost.

Module Type UBox(Import ipm : IdPreds)(Import opm : OidPreds)
  (Import UTSparam : UTypeSys)(Import UTSOidParam : UTypeSysOid UTSparam opm)
  (Import uc : UCost UTSparam ).

  Parameter Encoding : Set.

  Implicit Arguments UPot [CTDT CTDTP cb].

  Implicit Arguments existT [A P].

  Definition InpOutpTypes(CTDT : Type)
      (CTDTP : ProtoT → CTDT → Prop) := ipm.VaridMapMod.t
    ({t : UTSparam.ProtoT &
      {o : opm.HBCL_OidUT & UTSOidParam.T o t} &
      sig (CTDTP t)} × nat × nat).

  Definition UDataPSTMatchesInpOutpTypes(CTDT : Type)
    (CTDTP : ProtoT → CTDT → Prop)
    (inpTypes : InpOutpTypes CTDT CTDTP)
    (udat : ipm.VaridMapMod.t (sigT UDataPST)) : Prop :=
    ipm.VaridMapMod.Equiv ProtoEqT
    (ipm.VaridMapMod.map (projT1 (P := UDataPST)) udat)
    (ipm.VaridMapMod.map
      (fun inpOutpType ⇒ (sigTypes.projT1sigT2
        (P := fun t ⇒
          {o : opm.HBCL_OidUT & UTSOidParam.T o t})
        (Q := fun t ⇒ sig (CTDTP t)
      )) (fst (fst inpOutpType))) inpTypes).

  Record UExprLang := {

    CTDT : Type;
    CTDTP : ProtoT → CTDT → Prop;
    costB : CostBase TypeS CTDT CTDTP DataR DataP;
    AST : Set;

    parse : Encoding → AST;
    sso : InpOutpTypes CTDT CTDTP → InpOutpTypes CTDT CTDTP →
      Type;
    compile (itypes otypes : InpOutpTypes CTDT CTDTP) :
    option (sso itypes otypes);


    reduce (itypes otypes : InpOutpTypes CTDT CTDTP) :
    sso itypes otypes →
    sig (UDataPSTMatchesInpOutpTypes CTDT CTDTP itypes) →
    sig (UDataPSTMatchesInpOutpTypes CTDT CTDTP otypes)


  }.
End UBox.
```

D.1.7 The harmonic box abstraction

Listing D.7: The harmonic box signature

```
Require Import Coq.QArith.QArith_base. Close Scope Q_scope.
Require Export HBCL.HBCL_0_1.ModSignatures.HTypeSys.
Require Export HBCL.HBCL_0_1.ModSignatures.UCost.
Require Export HBCL.HBCL_0_1.ModSignatures.UBox.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.Util.Freq.
Require Coq.FSets.FMapInterface.
Require Coq.FSets.FMapWeakList.
Require Coq.Structures.DecidableType.
Require Coq.Structures.Equalities.

Module Type MDataType
  (ipm : IdPreds)(Import opm : OidPreds)(Import uts : UTypeSys)
  (Import UTSOidParam : UTypeSysOid uts opm)
  (Import HTSparam : HTypeSys uts opm UTSOidParam).

  Record MDatBoxFreqEltBase : Type :=
    { MDBFE_Base_Oid : oids.Oid;
      MDBFE_Base_Freq : Freq;
      MDBFE_Base_timt : sigTT HTSparam.TimedT;
      MDBFE_Base_TTFL : TTFL;
      MDBFE_Base_minSize : N;
      MDBFE_Base_maxSize : positive
    }.

  Definition MDatBoxElt := MDatBoxFreqEltBase.

  Inductive MemDatMode : Set := ReadEnabled | WriteEnabled.

  Definition memDatModeTimeRel(m : MemDatMode) : Z :=
    match m with
      | ReadEnabled ⇒ (-1)%Z
      | WriteEnabled ⇒ (1)%Z
    end.

  Definition MemDatListRaw :=
    list (sigTD (TimedV )).
  Implicit Arguments existTD [A B P].
  Implicit Arguments existT [A P].
  Inductive MemDatListPred(mode : MemDatMode)
    (mdfe: MDatBoxFreqEltBase )
    (baseTime : TTime (MDBFE_Base_Freq mdfe))
    (ttime : TTime (projTT2 (MDBFE_Base_timt mdfe))) :
    MemDatListRaw → Prop :=
  | MemDatListBasePred
    (tv : (TimedV (MDBFE_Base_timt mdfe) ttime)) :
      TTseq ttime baseTime →
      MemDatListPred mode mdfe baseTime ttime
      (cons (existTD (MDBFE_Base_timt mdfe) ttime tv) nil)
  | MemDatInd(prevLastTime : TTime (projTT2 (MDBFE_Base_timt mdfe)))
    (tv : (TimedV (MDBFE_Base_timt mdfe) ttime))
    (mdlr' : MemDatListRaw ) :
    MemDatListPred mode mdfe baseTime prevLastTime mdlr' →
    ( (getTimeZ ttime) =
      (getTimeZ prevLastTime) + memDatModeTimeRel mode)%Z →
    ( (1 + (getTimeZ ttime) - (getTimeZ baseTime)))%Z
    = Zpos (MDBFE_Base_maxSize mdfe) →
    MemDatListPred mode mdfe baseTime ttime
      (cons (existTD (MDBFE_Base_timt mdfe) ttime tv) mdlr').
  Definition MemDatTime(mode : MemDatMode)
    (mdfe: MDatBoxFreqEltBase )
    (baseTime : TTime (MDBFE_Base_Freq mdfe))
    (ttime : TTime (projTT2 (MDBFE_Base_timt mdfe))) :=
    sig (MemDatListPred mode mdfe baseTime ttime).
  Definition MDatTimeElt :=

    (MemDatListRaw )%type.
End MDataType.

Module Type MemModBox
  (ipm : IdPreds)(Import opm : OidPreds)(Import uts : UTypeSys)
  (Import UTSOidParam : UTypeSysOid uts opm)
  (Import HTSparam : HTypeSys uts opm UTSOidParam)
  (ott : OidPredType opm.oids)
  (ot : OidsPred opm.oids ott)
  (mdi : MDataType ipm opm uts UTSOidParam HTSparam).
  Declare Module otm : FMapInterface.WSfun(ot.PredoidDecidable).
  Module otmWPties :=
    FMapFacts.WProperties_fun ot.PredoidDecidable otm.
  Import mdi.
  Definition MDatTimeMapRaw :=
    otm.t ( MDatTimeElt ).
  Definition MDatBoxTimeMapPred(mode : MemDatMode)
    (freqm : otm.t MDatBoxElt)(f : Freq)(t : TTime f)
    (mdm : MDatTimeMapRaw) : Prop.
Admitted.
  Definition MDatBoxTime(mode : MemDatMode)
    (freqm : otm.t MDatBoxElt)(f : Freq)(t : TTime f) :=
    sig (MDatBoxTimeMapPred mode freqm f t).
End MemModBox.
Module Type HBox (ipm : IdPreds)(Import opm : OidPreds)(Import uts : UTypeSys)
  (Import uc : UCost uts)(Import UTSOidParam : UTypeSysOid uts opm)
  (Import ubox : UBox ipm opm uts UTSOidParam uc)
  (Import HTSparam : HTypeSys uts opm UTSOidParam).
  Declare Module MDataInst : MDataType ipm opm uts UTSOidParam HTSparam.

  Declare Module BoxTypeIdMapMod :
    FMapInterface.WSfun(ipm.boxidPred.PredidDecidable).
  Module ipmModVaridLCM :=
    MapLCM ipm.varidPred.PredidDecidable ipm.VaridMapMod.
```

```
Module ipmModBoxidLCM :=
  MapLCM ipm.boxidPred.PredidDecidable BoxTypeIdMapMod.

Declare Module InMemModBox : MemModBox ipm opm uts UTSOidParam HTSparam
  opm.oidMemFBPredType opm.oidMemFBPred

  MDataInst.

Declare Module OutMemModBox : MemModBox ipm opm uts UTSOidParam HTSparam
  opm.oidMemBFPredType opm.oidMemBFPred

 MDataInst.

Definition InMapVaridConvertPred
  (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
  (f : Freq)
  (ivm : InMemModBox.otm.t ipm.Varid)
  (vm : sig (vpred f))
  (im : InMemModBox.otm.t MDataInst.MDatBoxElt)
  := ∃ v,
∃ m, ipm.VaridMapMod.MapsTo v m (proj1_sig vm) →
      ∃ i, InMemModBox.otm.MapsTo i v ivm ∧
        InMemModBox.otm.MapsTo i m im.

  Definition InTypePredConvert
    (uexprlang : ubox.UExprLang)
    (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
    (f : Freq)
    (im : InMemModBox.otm.t MDataInst.MDatBoxElt)
    (utypes : InpOutpTypes _ _)
    (inpred : ∀(f : Freq)
      (freqm : sig (vpred f ))
      (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop)
    (ivm : InMemModBox.otm.t ipm.Varid) :=
    ∃ vm, InMapVaridConvertPred vpred f ivm vm im ∧
      inpred f vm utypes.

Definition OutMapVaridConvertPred
  (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
  (f : Freq)
  (ovm : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
  (om : OutMemModBox.otm.t MDataInst.MDatBoxElt)
(vm : sig (vpred f)) :=
  ∀ v, ∃ m, ipm.VaridMapMod.MapsTo v m (proj1_sig vm) →
    ∃ o, ipm.VaridMapMod.MapsTo v o ovm ∧
      OutMemModBox.otm.MapsTo o m om.

  Definition OutTypePredConvert
    (uexprlang : ubox.UExprLang)
    (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
    (f : Freq)
    (om : OutMemModBox.otm.t MDataInst.MDatBoxElt)
    (utypes : InpOutpTypes _ (ubox.CTDTP uexprlang))
    (outpred : ∀(f : Freq)
      (freqm : sig (vpred f ))
      (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop)
    (ovm : ipm.VaridMapMod.t opm.HBCL_OidMemBF) :=
    ∃ vm, OutMapVaridConvertPred vpred f ovm om vm ∧
      outpred f vm utypes.

  Parameter HBoxSSORaw : Type.

Module VaridMapWPties :=
  FMapFacts.WProperties_fun ipm.varidPred.PredidDecidable ipm.VaridMapMod.

Parameter HBoxSSOPred : ∀ (f fi fo : Freq)

  (ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
  (ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
,
  HBoxSSORaw → Prop.
  Definition HBoxSSO(f fi fo : Freq)

    (ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
    (ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
    :=
    sig (HBoxSSOPred f fi fo ttmfIn ttmfOut).
  Definition HBoxStepPred(f fmi fmo : Freq)
    (t : TTime f )
    (ti : TTime fmi)(to : TTime fmo)
    (ttmfIn : _ )
    (ttmfOut : _ )
    (memvarmap : InMemModBox.otm.t ipm.Varid)
    (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
    (ttmfIn′ : _)
    (ttmfOut′ : _)
:
    HBoxSSO f fmi fmo ( ttmfIn) ( ttmfOut) →
    InMemModBox.MDatBoxTime MDataInst.ReadEnabled ( ttmfIn′)
    fmi ti →
    OutMemModBox.MDatBoxTime MDataInst.WriteEnabled ( ttmfOut′)
    fmo to →
    Prop.
Admitted.
  Parameter HBoxStep : ∀(f fmi fmo : Freq)
    (t : TTime f )
    (ti : TTime fmi)(to : TTime fmo)
    (ttmfIn : _ )
    (ttmfOut : _ )
    (memvarmap : InMemModBox.otm.t ipm.Varid)
    (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
    (ttmfIn′ : _ )
    (ttmfOut′ : _ )
    (hbox : HBoxSSO f fmi fmo ttmfIn ttmfOut)
    (inp : InMemModBox.MDatBoxTime MDataInst.ReadEnabled (ttmfIn′) fmi ti),
    sig (HBoxStepPred f fmi fmo t ti to
      ttmfIn ttmfOut memvarmap varmemmap ttmfIn′ ttmfOut′
      hbox inp).
  Record HBoxSSONonDep : Type := {
    HBoxSSONonDep_f : Freq;
    HBoxSSONonDep_fi : Freq;
    HBoxSSONonDep_fo : Freq;

    HBoxSSONonDep_ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt
 ;
    HBoxSSONonDep_ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt
 ;
    HBoxSSONonDep_HBoxSSO : HBoxSSO HBoxSSONonDep_f HBoxSSONonDep_fi
      HBoxSSONonDep_fo HBoxSSONonDep_ttmfIn
      HBoxSSONonDep_ttmfOut
  }.
End HBox.
```

D.1.8 The coordination language

Listing D.8: The coordination language

```
Module Type MemDataTypeInstType (ipm : IdPreds)(Import opm : OidPreds)
  (Import uts : UTypeSys)
  (Import UTSOidParam : UTypeSysOid uts opm)
  (Import HTSparam : HTypeSys uts opm UTSOidParam)
  (Import mDataType : MDataType ipm opm uts UTSOidParam HTSparam).

  Record MDatFreqElt : Type := {
    MDFE_boxMemDat : MDatBoxFreqEltBase;
    MDFE_visible : bool
  }.
End MemDataTypeInstType.

Module MemDataTypeInst (ipm : IdPreds)(opm : OidPreds)
  (uts : UTypeSys)
  (UTSOidParam : UTypeSysOid uts opm)
  (HTSparam : HTypeSys uts opm UTSOidParam)
  (mDataType : MDataType ipm opm uts UTSOidParam HTSparam) <:
  MemDataTypeInstType ipm opm uts UTSOidParam HTSparam mDataType.

  Record MDatFreqElt : Type := {
    MDFE_boxMemDat : mDataType.MDatBoxFreqEltBase;
    MDFE_visible : bool
  }.
End MemDataTypeInst.

Module Type MemModInst (ipm : IdPreds)
  (Import opm : OidPreds)
  (Import uts : UTypeSys)(Import UTSOidParam : UTypeSysOid uts opm)
  (Import HTSparam : HTypeSys uts opm UTSOidParam)
  (ott : OidPredType opm.oids)
  (ot : OidsPred opm.oids ott)
  (Import mDataType : MDataType ipm opm uts UTSOidParam HTSparam)
  (Import mDataInstType :
    MemDataTypeInstType ipm opm uts UTSOidParam HTSparam mDataType)
  (Import memBoxes : MemModBox ipm opm uts UTSOidParam HTSparam ott ot mDataType
    ).


  Definition MDatFreqMapRaw := otm.t (MDatFreqElt).

  Parameter RawFreqMapPred : ∀(f : Freq)(mf : MDatFreqMapRaw), Prop.
  Definition MDatFreqMap(f : Freq) := sig (RawFreqMapPred f).

  Parameter MDatFreqMapIOPred : ∀ (f : Freq)(mdf : MDatFreqMap f),
    Prop.

  Definition MDatFreqMapIO(f : Freq) := sig (MDatFreqMapIOPred f).

  Parameter MDatFreqMapElt_seq : MDatFreqElt → MDatFreqElt → Prop.

  Definition MDatFreqMapEltOpt_seq(o1 : option (MDatFreqElt ))
    (o2 : option (MDatFreqElt )) :=
    match o1, o2 with
      | Some mdf1, Some mdf2 ⇒ MDatFreqMapElt_seq mdf1 mdf2
      | _, _ ⇒ False
    end.

  Parameter MDatMapFreqTimePred : ∀ (f : Freq)
      (mf :
      MDatFreqMap f), MDatTimeMapRaw → Prop.

  Definition MDatMapTime(f : Freq)
    (mf : MDatFreqMap f) :=
  sig (MDatMapFreqTimePred f mf).

  Parameter MDatMapModeReadPred
      : ∀ (f : Freq) (t : TTime f) (mf : MDatFreqMap f),
        MDatMapTime f mf → Prop.

  Parameter MDatMapModeWritePred
      : ∀ (f : Freq) (t : TTime f) (mf : MDatFreqMap f),
        MDatMapTime f mf → Prop.
End MemModInst.

Module Type Coord (Import ipm : IdPreds)
  (Import opm : OidPreds)(Import uts : UTypeSys)
  (Import uc : UCost uts)(Import UTSOidParam : UTypeSysOid uts opm)
  (Import ubox : UBox ipm opm uts UTSOidParam uc)
  (Import HTSparam : HTypeSys uts opm UTSOidParam)
```

```
(Import hbox : HBox ipm opm uts uc UTSOidParam ubox HTSparam).

Parameter CoordAST : Set.
Parameter parse : Encoding → CoordAST.

Module MDataTypeInst <:
  MemDataTypeInstType
  ipm opm uts
  UTSOidParam HTSparam MDataInst :=
  MemDataTypeInst ipm opm uts
  UTSOidParam HTSparam MDataInst.

Module SF_MemModInst(ott : OidPredType opm.oids)
  (ot : OidsPred opm.oids ott)
  (memBoxes : MemModBox ipm opm uts

  UTSOidParam HTSparam ott ot MDataInst
    ) <:
  MemModInst ipm opm uts
  UTSOidParam HTSparam ott ot MDataInst MDataTypeInst memBoxes.

  Definition MDatFreqMapRaw := memBoxes.otm.t (MDataTypeInst.MDatFreqElt).

  Definition RawFreqMapPred(f : Freq)(mf : MDatFreqMapRaw) : Prop.
Admitted.
  Definition MDatFreqMap(f : Freq) := sig (RawFreqMapPred f).

  Definition MDatFreqMapIOPred(f : Freq)(mdf : MDatFreqMap f) :
  Prop.
Admitted.

  Definition MDatFreqMapIO(f : Freq) := sig (MDatFreqMapIOPred f).

  Definition MDatFreqMapElt_seq : MDataTypeInst.MDatFreqElt →
    MDataTypeInst.MDatFreqElt → Prop.
Admitted.

  Definition MDatFreqMapEltOpt_seq(o1 : option (MDataTypeInst.MDatFreqElt))
    (o2 : option (MDataTypeInst.MDatFreqElt )) :=
    match o1, o2 with
      | Some mdf1, Some mdf2 ⇒ MDatFreqMapElt_seq mdf1 mdf2
      | _, _ ⇒ False
    end.

  Definition MDatMapFreqTimePred : ∀ f : Freq,
    ∀ mf :
    MDatFreqMap f, memBoxes.MDatTimeMapRaw → Prop.
Admitted.

  Definition MDatMapTime(f : Freq)
    (mf : MDatFreqMap f) := sig (MDatMapFreqTimePred f mf).

Print MDataTypeInst.MDatFreqElt.
Print MDataInst.MDatBoxFreqEltBase.
Print MDataInst.MDatTimeElt.

  Definition MDatMapModeReadPred(f : Freq)
   (t : TTime f)(mf : MDatFreqMap f)(mt : MDatMapTime f mf) :=
   (∀ oid : (sig ott.Pred), memBoxes.otm.In oid (proj1_sig mt)) ∧
   ∀ (oid : (sig ott.Pred))(tme : MDataInst.MDatTimeElt ),
     memBoxes.otm.MapsTo oid tme (proj1_sig mt) →
     ∃ mfe, memBoxes.otm.MapsTo oid mfe (‘mf) ∧
       ∃ t′, ∃ t″, TTseq t t′ ∧ TTseq t t″ ∧
     MDataInst.MemDatListPred MDataInst.ReadEnabled
     (MDataTypeInst.MDFE_boxMemDat mfe) t′ t″ tme.

  Definition MDatMapModeWritePred(f : Freq)
   (t : TTime f)(mf : MDatFreqMap f)(mt : MDatMapTime f mf) :=
   (∀ oid : (sig ott.Pred), memBoxes.otm.In oid (proj1_sig mt)) ∧
   ∀ (oid : (sig ott.Pred))(tme : MDataInst.MDatTimeElt ),
     memBoxes.otm.MapsTo oid tme (proj1_sig mt) →
     ∃ mfe, memBoxes.otm.MapsTo oid mfe (‘mf) ∧
       ∃ t′, ∃ t″, TTseq t t′ ∧ TTseq t t″ ∧
     MDataInst.MemDatListPred MDataInst.WriteEnabled
     (MDataTypeInst.MDFE_boxMemDat mfe) t′ t″ tme.
```

```
End SF_MemModInst.
Module InMemModInst : MemModInst ipm opm uts

   UTSOidParam HTSparam
   oidMemFBPredType oidMemFBPred MDataInst MDataTypeInst InMemModBox :=
   SF_MemModInst oidMemFBPredType oidMemFBPred InMemModBox.
Module OutMemModInst : MemModInst ipm opm uts

   UTSOidParam HTSparam
   oidMemBFPredType oidMemBFPred MDataInst MDataTypeInst OutMemModBox :=
   SF_MemModInst oidMemBFPredType oidMemBFPred OutMemModBox.
Record LInstSignatureRaw : Type := {
   InstSigFreqMemIn : Freq;
   InstSigFreqMemOut : Freq;
   InstSigInputMems : InMemModInst.MDatFreqMapIO InstSigFreqMemIn;
   InstSigOutputMems : OutMemModInst.MDatFreqMapIO InstSigFreqMemOut
}.
Definition LInstSignature(f : Freq) :=
   { lisr : LInstSignatureRaw |
      FreqDivide (InstSigFreqMemIn lisr) f ∧
      FreqDivide (InstSigFreqMemOut lisr) f
   }.
Inductive LInstSSORaw : Type :=
   LInstSSORaw_make(fmi fmo fmil fmol fmin fmon fl fn f : Freq) :

   InMemModInst.MDatFreqMap fmil →
   OutMemModInst.MDatFreqMap fmol →
   InMemModInst.MDatFreqMap fmin →
   OutMemModInst.MDatFreqMap fmon →
   BoxTypeIdMapMod.t
   (HBoxSSONonDep × (InMemModBox.otm.t Varid)
      × (VaridMapMod.t opm.HBCL_OidMemBF)) →
   InMemModBox.otm.t (HBCL_OidMemFB) →
   OutMemModBox.otm.t (HBCL_OidMemBF) →
   VaridMapMod.t (HBCL_OidMemBF × HBCL_OidMemFB) →

   LInstMapMod.Raw.t
      ( LInstSSORaw ) →
   LInstMapMod.t HBCL_OidLInst →
LInstMapMod.t
((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw) →

   LLibMapMod.Raw.t (LibSSORaw ) →
LInstMapMod.t TTFL →

LLibMapMod.t ( LInstMapMod.t LInstSignatureRaw) →

LInstSSORaw

with LibSSORaw : Type :=
   LLibSSORaw_make :

   VaridMapMod.t (sigTD T) →
   VaridMapMod.t (sigTT TimedT) →
   LInstMapMod.Raw.t (LInstSSORaw ) →
LInstMapMod.t
((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw) →

   LLibMapMod.Raw.t (LibSSORaw ) →
LLibMapMod.t ( LInstMapMod.t LInstSignatureRaw) →

   LibSSORaw .
Module opmModLInstLCM :=
   MapLCM oidLInstPred.PredoidDecidable opm.LInstMapMod.
Definition FMapLCMQualLInst(f : Freq) := sig (opmModLInstLCM.FreqIsLCMMap f).
Definition HBoxMapPredF(f : Freq)(h : BoxTypeIdMapMod.t
   (HBoxSSONonDep × (InMemModBox.otm.t ipm.Varid)
      × (ipm.VaridMapMod.t opm.HBCL_OidMemBF))) :=
ipmModBoxidLCM.FreqIsLCMMapFunc _ HBoxSSONonDep_f f
(BoxTypeIdMapMod.map (fun m ⇒ fst (fst m)) h).

Definition HBoxLCMQual(f : Freq) := sig (HBoxMapPredF f).

Inductive LInstSSOPred :
   ∀ f, LInstSignature f → LInstMapMod.t LInstSignatureRaw →
      LInstSSORaw
   → Prop :=
   LInstSSO_intro

   (f fmi fmo fmil fmol fmin fmon fl fn f : Freq)

   (hboxSSO : sigT HBoxLCMQual )
   (obsMap : InMemModBox.otm.t (HBCL_OidMemFB) )
   (manifMap : OutMemModBox.otm.t
      (HBCL_OidMemBF) )
   (fifoMap : VaridMapMod.t (HBCL_OidMemBF × HBCL_OidMemFB))
   (instNestMap : LInstMapMod.t (LInstSSORaw))

   (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)

   (instTypeMap : LInstMapMod.t HBCL_OidLInst)
   (instTShiftMap : LInstMapMod.t TTFL)
(instSigDatMap : LInstMapMod.t
 ((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw))

   (libMap : LLibMapMod.t LibSSORaw)
   (libmapsigmap : LLibMapMod.t
      ( LInstMapMod.t LInstSignatureRaw))
   (instMemIn : InMemModInst.MDatFreqMapIO fmi)
   (instMemOut : OutMemModInst.MDatFreqMapIO fmo)
   (instMemInLoc : InMemModInst.MDatFreqMap fmil)
   (instMemOutLoc : OutMemModInst.MDatFreqMap fmol)
   (instMemInNest : InMemModInst.MDatFreqMap fmin)
   (instMemOutNest : OutMemModInst.MDatFreqMap fmon)
   (fmnest : sigT FMapLCMQualLInst)
   (fmtref : sigT FMapLCMQualLInst):

   FreqIsLCMList f (cons fmi (cons fmo (cons (projT1 hboxSSO)
      (cons (projT1 fmnest) (cons (projT1 fmtref) nil))))) →

   ((∀ v, LInstMapMod.In v (proj1_sig (projT2 fmnest)) ↔
      LInstMapMod.In v instNestMap) →
      (∀ v,
         ∀ lir, LInstMapMod.MapsTo v lir instNestMap →
            ∀ f′, LInstMapMod.MapsTo v f′ (proj1_sig (projT2 fmnest)) →
               ∃ fmi′, ∃ fmo′,
               ∃ inmems, ∃ outmems, ∃ instTypeScopeMap′,
                  ∃ fsmatch,
               LInstSSOPred (f′) (exist (fun lisr ⇒
                  FreqDivide (InstSigFreqMemIn lisr) f′ ∧
                  FreqDivide (InstSigFreqMemOut lisr) f′)
               (Build_LInstSignatureRaw fmi′ fmo′ inmems outmems) fsmatch)
               instTypeScopeMap′ ( lir))
   ) →





   ∀ lisp,
   LInstSSOPred (f)
   (exist _ (Build_LInstSignatureRaw fmi fmo instMemIn instMemOut) lisp)
   instTypeScopeMap (LInstSSORaw_make fmi fmo fmil fmol
        fmin fmon fl fn f
        (instMemInLoc) (instMemOutLoc) (instMemInNest) (instMemOutNest)
        (proj1_sig (projT2 hboxSSO)) obsMap manifMap fifoMap
        (LInstMapMod.this instNestMap) instTypeMap
        instSigDatMap
        (LLibMapMod.this libMap)
     instTShiftMap libmapsigmap)

  with LLibSSOPred : LInstMapMod.t LInstSignatureRaw → LibSSORaw → Prop :=
     LLibSSO_intro
     (libsig : LInstMapMod.t LInstSignatureRaw)
     (libenv : LInstMapMod.Raw.t LInstSignatureRaw)
     (utypes : VaridMapMod.t (sigTD T))
     (ttypes : VaridMapMod.t (sigTT TimedT))
     (instMap : LInstMapMod.t LInstSSORaw)
(instSigDatMap : LInstMapMod.t
 ((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw))
     (fminst : sigT FMapLCMQualLInst)
     (libNestMap : LLibMapMod.t LibSSORaw)
     (libmapsigmap :
        LLibMapMod.t ( LInstMapMod.t LInstSignatureRaw)) :



((∀ v, LInstMapMod.In v (proj1_sig (projT2 fminst)) ↔
        LInstMapMod.In v instMap) →
        (∀ v,
           ∀ lir, LInstMapMod.MapsTo v lir instMap →
              ∀ f′, LInstMapMod.MapsTo v f′ (proj1_sig (projT2 fminst)) →
                 ∃ fmi′, ∃ fmo′,
                 ∃ inmems, ∃ outmems, ∃ instTypeScopeMap,
                    ∃ lisp,
                 LInstSSOPred (f′)
                 (exist _ (Build_LInstSignatureRaw fmi′ fmo′ inmems outmems) lisp)
                 instTypeScopeMap lir)
     )
     →

  ((∀ v, VaridMapMod.In v ttypes → VaridMapMod.In v utypes) →
  ∀ v t u, VaridMapMod.MapsTo v t ttypes →
  VaridMapMod.MapsTo v u utypes → UTSOidParam.TEq u (projTT3 t)) →
     LLibSSOPred libsig (LLibSSORaw_make utypes ttypes
        (LInstMapMod.this instMap) instSigDatMap
        (LLibMapMod.this libNestMap) libmapsigmap).
  Definition LInstSSO(f : Freq)(linstsig : LInstSignature f)
     (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
   :=
     sig (LInstSSOPred (f)
  linstsig instTypeScopeMap).
  Definition LLibSSO(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw) :=
     sig (LLibSSOPred instTypeScopeMap).
  Inductive LissoLibPred : ∀
     (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw),
     LLibMapMod.t (sigT LLibSSO) → Prop :=.
  Definition LissoLibDep(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw) :=
     sig (LissoLibPred instTypeScopeMap).
  Definition LibClosRaw :=
     list ((LInstMapMod.t LInstSignatureRaw) × (sigT LissoLibDep)).
  Inductive LibClosPred :
     LInstMapMod.t LInstSignatureRaw → LibClosRaw → Prop :=.
  Definition LibClos(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw) :=
     sig (LibClosPred instTypeScopeMap).
  Definition Lisso(f : Freq)(linstsig : LInstSignature f)
     (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
     (lissoLibMap : LibClos instTypeScopeMap) :=
  LInstSSO f linstsig instTypeScopeMap.
Inductive CoordStateRaw : Type :=
| CoordStateRaw_make :
  InMemModBox.MDatTimeMapRaw → OutMemModBox.MDatTimeMapRaw
  →
  LInstMapModRaw.t ( CoordStateRaw) → CoordStateRaw.
Inductive CoordStateStaticPred(f : Freq)

  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)

  (lisso : Lisso f linstsig instTypeScopeMap lissolib)

:
  CoordStateRaw → Prop :=
| CoordStatePred_intro(f′ fl fn fmi′ fmo′ : Freq)
  (mfi′ : InMemModInst.MDatFreqMap fmi′)
  (mfo′ : OutMemModInst.MDatFreqMap fmo′)
  (mfiTrace′ : InMemModInst.MDatMapTime fmi′ mfi′)
  (mfoTrace′ : OutMemModInst.MDatMapTime fmo′ mfo′)
  (nestCSR : LInstMapMod.t ( CoordStateRaw)) :

  CoordStateStaticPred f linstsig instTypeScopeMap
  lissolib lisso
  (CoordStateRaw_make
     (‘mfiTrace′) (‘mfoTrace′)
     (LInstMapMod.this nestCSR)).
 Inductive CoordStateBoxesEnabled(f : Freq)
  (t : TTime f)(linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=


 with CoordStateMemBFEnabled(f : Freq)
  (t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=


 with CoordStateInnerFIFOsEnabled(f : Freq)
  (t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=


 with CoordStateInnerMemFBEnabled(f : Freq)
  (t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=.
Definition CoordStateOuterFIFOsEnabled := CoordStateBoxesEnabled.
```

```
  Definition CoordStateOuterMemFBEnabled := CoordStateInnerMemFBEnabled.
  Record CSTempCorrectND
  : Type := {
    CSTempCorrectND_f : Freq;
    CSTempCorrectND_t : TTime CSTempCorrectND_f;
    CSTempCorrectND_linstsig : LInstSignature CSTempCorrectND_f;
    CSTempCorrectND_instTypeScopeMap : LInstMapMod.t LInstSignatureRaw;
    CSTempCorrectND_lissolib : LibClos CSTempCorrectND_instTypeScopeMap;
    CSTempCorrectND_lisso : Lisso CSTempCorrectND_f
      CSTempCorrectND_linstsig
      CSTempCorrectND_instTypeScopeMap
      CSTempCorrectND_lissolib;
    CSTempCorrectND_CS :
      sig (CoordStateStaticPred CSTempCorrectND_f
        CSTempCorrectND_linstsig
      CSTempCorrectND_instTypeScopeMap CSTempCorrectND_lissolib
      CSTempCorrectND_lisso)

  }.
  Definition InstMapMatchSSO(f fn : Freq)(linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
    (lisso : Lisso f linstsig instTypeScopeMap lissolib)
    (csTemporalPred : ∀ f : Freq,
                          TTime f →
                          ∀ (linstsig : LInstSignature f)
                            (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
                            (lissolib : LibClos instTypeScopeMap)
                            (lisso : Lisso f linstsig instTypeScopeMap lissolib),
                          sig
                            (CoordStateStaticPred f linstsig instTypeScopeMap
                              lissolib lisso) → Prop)
    (nestCSR : LInstMapMod.t (
      (CSTempCorrectND) )) : Prop.
Admitted.
  Inductive InstBoxesNestPred
    (f fn : Freq)(t : TTime f)(linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
    (lisso : Lisso f linstsig instTypeScopeMap lissolib)

    (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso
      CoordStateInnerFIFOsEnabled ))
    (nestCSR' : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso
      CoordStateOuterFIFOsEnabled ))
    (traceEnab : ∀ (f : Freq)(t : TTime f)
      (linstsig : LInstSignature f)
      (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
      (lissolib : LibClos instTypeScopeMap)
      (lisso : Lisso f linstsig instTypeScopeMap lissolib)
      (cstate :
        sig (CoordStateBoxesEnabled
          f t linstsig instTypeScopeMap lissolib lisso))
      ,
      Type) : Prop :=.
  Definition isNextTimeStep(f : Freq)(t t' : TTime f) :=
      TTseq (tNext f t) t'.
  Module BoxesStep .

    Inductive StepSSOPred(f fmi fmo : Freq)
      (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
      (ndmap : BoxTypeIdMapMod.t
        (HBoxSSONonDep × (InMemModBox.otm.t ipm.Varid)
      × (ipm.VaridMapMod.t opm.HBCL_OidMemBF))) : Prop :=
      | StepSSO_intro :

        StepSSOPred f fmi fmo mfi mfo ndmap.
      Definition StepSSO(f fmi fmo : Freq)
        (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
        := sig (StepSSOPred f fmi fmo mfi mfo).
      Definition StepPred(f fmi fmo : Freq)
        (t t' : TTime f )
        (ti ti' : TTime fmi)(to to' : TTime fmo)
        (mfi : InMemModInst.MDatFreqMap fmi)
        (mfo : OutMemModInst.MDatFreqMap fmo) :
        StepSSO f fmi fmo mfi mfo →
        isNextTimeStep f t t' →
        sig (InMemModInst.MDatMapModeReadPred _ ti mfi) →

        sig (OutMemModInst.MDatMapModeWritePred _ to mfo) →



        Prop.
Admitted.
   End BoxesStep.
      CoInductive TraceBoxesEnab(f : Freq)(t : TTime f)
      (linstsig : LInstSignature f)
      (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
      (lissolib : LibClos instTypeScopeMap)
      (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
      ∀

      (cstate : sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap
        lissolib lisso))


,
      Type :=
   | TraceBoxesStep (fl fn : Freq)(fmti fmto : Freq)
      (tfl tfl' : TTime fl)
      (t' : TTime f)(ti' : TTime (InSigFreqMemIn ('linstsig)))
      (tti tti' : TTime fmti)(tto tto' : TTime fmto)
      (mfti : InMemModInst.MDatFreqMap fmti)
      (mfto : OutMemModInst.MDatFreqMap fmto)
      (bssso : BoxesStep.StepSSO fl fmti fmto mfti mfto)
      (mftiState : InMemModInst.MDatMapTime fmti mfti)
      (mftoState : OutMemModInst.MDatMapTime fmto mfto)
      (mftiState' : InMemModInst.MDatMapTime fmti mfti)
      (mftoState' : OutMemModInst.MDatMapTime fmto mfto)
      (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
        lisso
        CoordStateInnerFIFOsEnabled))
      (nestCSR' : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
        lisso
        CoordStateOuterFIFOsEnabled))

      (prfnxtim : isNextTimeStep fl tfl tfl')
      (bsteppre :
        InMemModInst.MDatMapModeReadPred _ tti mfti mftiState)

      (bsteppost :
        OutMemModInst.MDatMapModeWritePred _ tto mfto mftoState)

      (cstate :
        sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap lissolib lisso))
      (cstateNext :
        sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap lissolib lisso))
      :

      BoxesStep.StepPred fl fmti fmto tfl tfl' tti tti' tto tto'
```

```
  mfti mfto bssso prfnxtim
  (exist _ _ bsteppre)
  (exist _ _ bsteppost)

  →
  (InstBoxesNestPred f fn t linstsig instTypeScopeMap lissolib lisso
nestCSR nestCSR′ TraceBoxesEnab) →
  TraceMemBFEnab f t linstsig instTypeScopeMap lissolib lisso cstateNext
  →
  TraceBoxesEnab f t linstsig instTypeScopeMap lissolib lisso cstate


  with TraceMemBFEnab(f : Freq)(t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  ∀
  (cstate : sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap
    lissolib lisso))
  ,
  Type :=
| TraceMemBFStep

  (cstate :
    sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap lissolib lisso))
  (cstateNext :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
      lisso)) :
  TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso cstateNext
  →
  TraceMemBFEnab f t linstsig instTypeScopeMap lissolib lisso cstate


with TraceFIFOsEnab(f : Freq)(t : TTime f)
(linstsig : LInstSignature f)
(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
(lissolib : LibClos instTypeScopeMap)
(lisso : Lisso f linstsig instTypeScopeMap lissolib) :
∀
  (cstate :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  ,
  Type
  :=
| TraceFIFOsStep
  (cstate :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  (cstateNext :
    sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  :
    TraceMemFBEnab f t linstsig instTypeScopeMap lissolib lisso cstateNext →
  TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso cstate
  | TraceFIFOsFinal (cstate :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  : OutMemModInst.MDatMapTime _ (‘ (InstSigOutputMems (‘linstsig))) →
    TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso cstate


  with TraceMemFBEnab(f : Freq)(t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  ∀
  (cstate :
    sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  ,
  Type :=
  | TraceMemFBStepSkipIO
  (cstate :
    sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  (cstateNext :
    sig (CoordStateBoxesEnabled f (tNext f t) linstsig instTypeScopeMap
      lissolib lisso))
  :
  TraceBoxesEnab f (tNext _ t) linstsig instTypeScopeMap lissolib lisso
  cstateNext →
  TraceMemFBEnab f t linstsig instTypeScopeMap lissolib lisso cstate.
Module LInstMapModWPties :=
  FMapFacts.WProperties_fun opm.oidLInstPred.PredoidDecidable
  LInstMapMod.
Definition StatSemObj := sigTQ Lisso.
Module InMemBoxWPties :=
  FMapFacts.WProperties_fun opm.oidMemFBPred.PredoidDecidable
  InMemModBox.otm.
Module OutMemBoxWPties :=
  FMapFacts.WProperties_fun opm.oidMemBFPred.PredoidDecidable
  OutMemModBox.otm.
Lemma inclReflBoxid : ∀ elt m,
    SetoidList.inclA (@BoxTypeIdMapMod.eq_key_elt elt) m m.
CoInductive InputStream(f : Freq)(mfi : InMemModInst.MDatFreqMapIO f)
  (t : TTime f) : Type :=
  | InputStreamFinal :

    InputStream f mfi t
  | InputStreamInd(tnext : TTime f) :
    TTseq (tNext f t) tnext →
    sig (InMemModInst.MDatMapModeReadPred f t ('mfi)) →
    InputStream f mfi tnext → InputStream f mfi t.


End Coord.
```

D.2 Correspondence of coordination language operational semantics with Coq code

Rule	Type-theoretical operational semantic form	Coq form
FIFO step and entry point	Rule C.133	Listing D.18
FIFO-box memory step	Rule C.134	Listing D.20
Box step top level	Rule C.135	Listing D.21

Rule	Type-theoretical operational semantic form	Coq form
Box step local	Rule C.136	Listing D.22
Box-FIFO memory step	Rule C.141	Listing D.24
Nested box step	Rule C.139	Listing D.25

Table D.1: Coordination language formalization correspondence

D.3 Correspondence of expression language operational semantics with Coq code

Rule	Type-theoretical operational semantic form	Coq form
Function invocation	Rule C.143	Listing D.36
Expression reduction	Rule C.148, rule C.149, and rule C.150	The three branches of ‘`reduce`’ in Listing D.37
Expression pattern application	Rule C.151	Listing D.33
Expression tuple construction	Rule C.156	Listing D.34
Expression record construction	Rule C.157	Listing D.35

Table D.2: Expression language formalization correspondence

D.4 Module instantiations

D.4.1 The ID implementation

Listing D.9: ID implementation

```
Require Import HBCL.HBCL_0_1.ModSignatures.Ids.
Require Import Coq.Structures.Equalities.
Require Import Coq.Bool.Bool.
Require Import Coq.Lists.List.
Require Import HBCL.Util.FMapRawIface.

Module HBCL_0_1_Id_S <: IdPreds.
Module ids <: Ids.
  Require Import Coq.Strings.Ascii.
  Require Import Coq.Strings.String.
  Module Ascii_as_MDT : MiniDecidableType with Definition t := ascii.
    Definition t := ascii.
    Definition eq_dec := ascii_dec.
  End Ascii_as_MDT.
  Module Ascii_as_UDT : UsualDecidableTypeBoth with Definition t := ascii :=
    Make_UDT Ascii_as_MDT.
  Module Ascii_as_HEB : HasEqBool Ascii_as_UDT :=
    HasEqDec2Bool Ascii_as_UDT Ascii_as_UDT.
  Module AsciiDecidable : UsualDecidableTypeFull with Definition t := ascii :=
    Ascii_as_UDT <+ Ascii_as_HEB.
  Module String_as_MDT : MiniDecidableType with Definition t := string.
    Definition t := string.
    Definition eq_dec := string_dec.
  End String_as_MDT.
  Module String_as_UDT : UsualDecidableTypeBoth with Definition t := string :=
    Make_UDT String_as_MDT.
  Module String_as_HEB : HasEqBool String_as_UDT :=
    HasEqDec2Bool String_as_UDT String_as_UDT.
  Module StringDecidable : UsualDecidableTypeFull with Definition t :=
    string := String_as_UDT <+ String_as_HEB.
  Require Import Coq.NArith.BinNat.
  Require Import Coq.NArith.NOrderedType.
  Require Import Coq.Structures.Orders.
  Module N_as_OTF := OT_to_Full N_as_OT.
  Module N_as_TTLB := OTF_to_TTLB (N_as_OTF).
  Notation nOTF_le := N_as_OTF.le.
  Local  Open Scope N_scope.
  Inductive isLetterN(na : N) : Prop :=
  | isLetterUC : nOTF_le 65 na ∧ nOTF_le na 90 → isLetterN na
  | isLetterLC : nOTF_le 97 na ∧ nOTF_le na 122 → isLetterN na.
  Local  Close Scope N_scope.
  Definition isLetter(a : ascii) : Prop := isLetterN (N_of_ascii a).
  Definition isLetterBool(a : ascii) : bool :=
    let na := N_of_ascii a in
      (N_as_TTLB.leb 65 na && N_as_TTLB.leb na 90 | |
        N_as_TTLB.leb 97 na && N_as_TTLB.leb na 122)%N.
  Lemma isLetterCorrect : ∀(a : ascii),
    isLetterBool a = true ↔ isLetter a.
  Definition IdLocalPred(s : string) : Prop :=
    match s with
      | String a _ ⇒ isLetter a
      | EmptyString ⇒ False
    end.
  Definition Id := sig IdLocalPred.
  Require Import HBCL.Util.SigModuleFunctors.
  Module idBaseSigPred <: SigPred StringDecidable
    with Definition p := IdLocalPred.
    Definition p := IdLocalPred.
  End idBaseSigPred.
  Module IdDecidable <: DecidableTypeSigF String_as_MDT
    idBaseSigPred StringDecidable
      := MakeDTSFFromDT StringDecidable idBaseSigPred StringDecidable.
  Theorem predDecRelPirrel : ∀ x y p q,
    StringDecidable.eq x y →
    IdDecidable.eq (exist idBaseSigPred.p x p) (exist idBaseSigPred.p y q).
End ids.
```

```
  Module IdsPred(idpmod : IdPredType ids) : IdsPred ids idpmod.
    Require Import Equalities.
    Require Import HBCL.Util.SigModuleFunctors.
    Import ids.
    Definition PredID := sig idpmod.Pred.
    Module idTyp : Typ with Definition t := Id.
      Definition t := Id.
    End idTyp.
    Module idSigPred <: SigPred idTyp with Definition p := idpmod.Pred.
      Definition p := idpmod.Pred.
    End idSigPred.
    Module PredidDecidable <: DecidableTypeSigF idTyp idSigPred ids.IdDecidable
        := MakeDTSFFromDT idTyp idSigPred ids.IdDecidable.
    Theorem IdPredRelPirrel : ∀ x y p q,
      ids.IdDecidable.eq x y →
      PredidDecidable.eq (exist idSigPred.p x p) (exist idSigPred.p y q).
  End IdsPred.
  Module varidPredType <: VaridPredType ids.
    Import ids.
    Definition Pred := fun (_ : Id) ⇒ True.
  End varidPredType.
  Module varidPred <: Varid ids varidPredType := IdsPred varidPredType.
  Module typidPredType <: TypidPredType ids.
    Import ids.
    Definition Pred := fun (_ : Id) ⇒ True.
  End typidPredType.
  Module typidPred <: Typid ids typidPredType := IdsPred typidPredType.
  Module boxidPredType <: BoxidPredType ids.
    Import ids.
    Definition Pred := fun (_ : Id) ⇒ True.
  End boxidPredType.
  Module boxidPred <: Boxid ids boxidPredType := IdsPred boxidPredType.
  Definition Varid := varidPred.PredID.
  Definition Typid := typidPred.PredID.
  Definition Boxid := boxidPred.PredID.
  Require Coq.FSets.FMapInterface.
  Require Coq.FSets.FMapWeakList.
  Require Coq.Structures.DecidableType.
  Module VaridMapModComplete <:
    FMapInterface.WSfun varidPred.PredidDecidable
  :=
    FMapWeakList.Make varidPred.PredidDecidable.
  Module VaridMapModRaw <: FMapIfaceRaw varidPred.PredidDecidable
    with Definition t := fun elt ⇒ list (Varid × elt)
:=
      VaridMapModComplete.Raw.
  Module VaridMapModPred <: FMapModDatImplPred varidPred.PredidDecidable
      VaridMapModRaw.
    Definition NoDupType : ∀ elt,
      VaridMapModRaw.t elt → Prop := fun elt ⇒
        SetoidList.NoDupA (@VaridMapModRaw.PX.eqk elt).
  End VaridMapModPred.
Module VaridMapMod <:
    FMapIfaceRF varidPred.PredidDecidable
    VaridMapModRaw VaridMapModPred with Definition key := Varid
    with Module Raw := VaridMapModRaw.
  Include VaridMapModComplete.
  Definition Build_t := Build_slist.
```

```
End VaridMapMod.
End HBCL_0_1_Id_S.
```

D.4.2 The OID implementation

Listing D.10: OID implementation

```
Require Import HBCL.HBCL_0_1.ModSignatures.Oids.
Require Import Coq.NArith.BinNat.
Require Import Coq.Structures.Equalities.
Require Coq.FSets.FMapInterface.
Require Coq.FSets.FMapWeakList.

Require Import Coq.Strings.Ascii.
Require Import Coq.Strings.String.
Definition N := string.

Require Import Coq.Lists.List.
Require Import HBCL.Util.SigModuleFunctors.
Require Import HBCL.Util.FMapRawIface.

Module HBCL_0_1_Oid_S <: OidPreds with Definition oids.Oid := list N.

  Module oids : Oids with Definition Oid := list N.

    Definition Oid := list N.

    Lemma oid_dec : ∀ x y : Oid, {x = y} + {x ≠ y}.

    Module OID_as_MDT : MiniDecidableType with Definition t := Oid.

      Definition t := Oid.
      Definition eq_dec := oid_dec.

    End OID_as_MDT.

    Module OID_as_UDT : UsualDecidableTypeBoth with Definition t := Oid :=
      Make_UDT OID_as_MDT.

    Module OID_as_HEB : HasEqBool OID_as_UDT :=
      HasEqDec2Bool OID_as_UDT OID_as_UDT.

    Module OidDecidable : DecidableTypeFull with Definition t := Oid :=
      OID_as_UDT <+ OID_as_HEB.

    Definition OidLength(o : Oid) := length o.

  End oids.

  Module String_as_MDT : MiniDecidableType with Definition t := string.

      Definition t := string.
      Definition eq_dec := string_dec.

    End String_as_MDT.

    Module String_as_UDT : UsualDecidableTypeBoth with Definition t := string :=
      Make_UDT String_as_MDT.

    Module String_as_HEB : HasEqBool String_as_UDT :=
      HasEqDec2Bool String_as_UDT String_as_UDT.

    Module StringDecidable : DecidableTypeFull with Definition t := string :=
      String_as_UDT <+ String_as_HEB.

  Module Type OidPredPrefix.

    Parameter HBCL_Oid_Prefix : N.

  End OidPredPrefix.

  Module Type OidPredTypeBaseType
    (Import oidmod : Oids with Definition Oid := list N).

    Parameter HBCL_OidRootArc : Oid.
    Parameter HBCL_Esc_Tok : N.

  End OidPredTypeBaseType.

  Module Type OidPredTypeFuncRootType
    (Import oidmod : Oids with Definition Oid := list N)
  (Import bt : OidPredTypeBaseType oidmod).

  Definition Pred(o : Oid) :=
    ∃ append, OidDecidable.eq o (append ++ HBCL_OidRootArc).

End OidPredTypeFuncRootType.

Module Type OidPredTypeFuncStemType
  (Import oidmod : Oids with Definition Oid := list N)
  (opstem : OidPredType oidmod)
  (Import bt : OidPredTypeBaseType oidmod)
  (oppfix : OidPredPrefix)
  (optfr : OidPredTypeFuncRootType oidmod bt)
  <: OidPredType oidmod.

  Inductive PredStem : Oid → Prop :=
  | PredStemRoot(append : Oid) :
    PredStem (append ++ oppfix.HBCL_Oid_Prefix :: HBCL_OidRootArc)
  | PredStemNonRoot(stem append : Oid) : opstem.Pred stem →
    PredStem (append ++ oppfix.HBCL_Oid_Prefix :: HBCL_Esc_Tok :: stem).

  Definition Pred(o : Oid) := optfr.Pred o ∧ PredStem o.

End OidPredTypeFuncStemType.

Module Type OidPredTypeFuncStemExtTermType
  (Import oidmod : Oids with Definition Oid := list N)
  (opstem : OidPredType oidmod)
  (Import bt : OidPredTypeBaseType oidmod)
  (oppfix : OidPredPrefix)
  (optfr : OidPredTypeFuncRootType oidmod bt)

  <: OidPredType oidmod.

  Inductive PredStem : Oid → Prop :=
  | PredStemRoot(append : N) :
    PredStem (append :: oppfix.HBCL_Oid_Prefix :: HBCL_OidRootArc)
  | PredStemNonRoot(stem : Oid)(append : N) : opstem.Pred stem →
    PredStem (append :: oppfix.HBCL_Oid_Prefix :: HBCL_Esc_Tok :: stem).

  Definition Pred(o : Oid) := optfr.Pred o ∧ PredStem o.

End OidPredTypeFuncStemExtTermType.

Module OidPredTypeRootBase
  (Import oidmod : Oids with Definition Oid := list N)
  <: OidPredTypeBaseType oidmod.

  Definition HBCL_OidRootArc : Oid := nil.
  Definition HBCL_Esc_Tok := ("ESC")%string.

End OidPredTypeRootBase.

Module OidPredTypeFuncRoot
  (Import oidmod : Oids with Definition Oid := list N)
  (Import bt : OidPredTypeBaseType oidmod)
  <: OidPredTypeFuncRootType oidmod bt.

  Definition Pred(o : Oid) :=
    ∃ append, OidDecidable.eq o (append ++ HBCL_OidRootArc).

End OidPredTypeFuncRoot.

Module OidPredTypeFuncStem
  (Import oidmod : Oids with Definition Oid := list N)
  (opstem : OidPredType oidmod)
  (Import bt : OidPredTypeBaseType oidmod)
  (oppfix : OidPredPrefix)
  (optfr : OidPredTypeFuncRootType oidmod bt)
  <: OidPredTypeFuncStemType oidmod opstem bt oppfix optfr.

  Inductive PredStem : Oid → Prop :=
  | PredStemRoot(append : Oid) :
    PredStem (append ++ oppfix.HBCL_Oid_Prefix :: HBCL_OidRootArc)
  | PredStemNonRoot(stem append : Oid) : opstem.Pred stem →
    PredStem (append ++ oppfix.HBCL_Oid_Prefix :: HBCL_Esc_Tok :: stem).

  Definition Pred(o : Oid) := optfr.Pred o ∧ PredStem o.

  Theorem PredImplPredRoot : ∀ o, Pred o → optfr.Pred o.
```

```
End OidPredTypeFuncStem.
Module OidPredTypeFuncStemExtTerm
  (Import oidmod : Oids with Definition Oid := list N)
  (opstem : OidPredType oidmod)
  (Import bt : OidPredTypeBaseType oidmod)
  (optfr : OidPredTypeFuncRootType oidmod bt)
  (oppfix : OidPredPrefix)

  <: OidPredTypeFuncStemExtTermType oidmod opstem bt oppfix optfr .
  Inductive PredStem : Oid → Prop :=
  | PredStemRoot(append : N) :
    PredStem (append :: oppfix.HBCL_Oid_Prefix :: HBCL_OidRootArc)
  | PredStemNonRoot(stem : Oid)(append : N) : opstem.Pred stem →
    PredStem (append :: oppfix.HBCL_Oid_Prefix :: HBCL_Esc_Tok :: stem).
  Definition Pred(o : Oid) := optfr.Pred o ∧ PredStem o.

End OidPredTypeFuncStemExtTerm.
Module HBCL_0_1_OidsPred
  (oidpmod : OidPredType oids) : OidsPred oids oidpmod.
  Import oids.
  Definition PredOid := sig oidpmod.Pred.
  Module oidTyp : Typ with Definition t := Oid.
    Definition t := Oid.
  End oidTyp.
  Module oidSigPred : SigPred oidTyp with Definition p := oidpmod.Pred.
    Definition p := oidpmod.Pred.
  End oidSigPred.
  Module PredoidDecidable : DecidableTypeFull
    with Definition t := sig oidpmod.Pred
      := MakeDTSFFromDT oidTyp oidSigPred oids.OidDecidable.
End HBCL_0_1_OidsPred.
Module oidLLibPrefix <: OidPredPrefix.
  Definition HBCL_Oid_Prefix := ("LLib")%string.
End oidLLibPrefix.
Module OidPredTypeRootBaseConc <: OidPredTypeBaseType oids :=
  OidPredTypeRootBase oids.
Module OidPredTypeRootConc <: OidPredTypeFuncRootType oids
  OidPredTypeRootBaseConc :=
  OidPredTypeFuncRoot oids OidPredTypeRootBaseConc.
Module oidLLibPredType <: OidLLibPredType oids :=
  OidPredTypeFuncStem oids OidPredTypeRootConc OidPredTypeRootBaseConc
  oidLLibPrefix OidPredTypeRootConc.
Module oidLLibPred : OidLLibPred oids oidLLibPredType :=
  HBCL_0_1_OidsPred oidLLibPredType.
Module oidUTPrefix <: OidPredPrefix.
  Definition HBCL_Oid_Prefix := ("UT")%string.
End oidUTPrefix.
Module oidUTPredType <: OidUTPredType oids :=
  OidPredTypeFuncStemExtTerm oids oidLLibPredType OidPredTypeRootBaseConc
  OidPredTypeRootConc oidUTPrefix.
Module oidUTPred : OidUTPred oids oidUTPredType :=
  HBCL_0_1_OidsPred oidUTPredType.
Module oidTTPrefix <: OidPredPrefix.
  Definition HBCL_Oid_Prefix := ("TT")%string.
End oidTTPrefix.
Module oidTTPredType <: OidTTPredType oids :=
  OidPredTypeFuncStemExtTerm oids oidLLibPredType OidPredTypeRootBaseConc
  OidPredTypeRootConc oidTTPrefix.
Module oidTTPred : OidTTPred oids oidTTPredType :=
    HBCL_0_1_OidsPred oidTTPredType.
  Module oidLInstPrefix <: OidPredPrefix.
    Definition HBCL_Oid_Prefix := ("LInst")%string.
  End oidLInstPrefix.
  Module oidLInstPredType <: OidLInstPredType oids :=
    OidPredTypeFuncStem oids oidLLibPredType OidPredTypeRootBaseConc
    oidLInstPrefix OidPredTypeRootConc.
  Module oidLInstPred : OidLInstPred oids oidLInstPredType :=
    HBCL_0_1_OidsPred oidLInstPredType.
  Module oidMemBFPrefix <: OidPredPrefix.
    Definition HBCL_Oid_Prefix := ("MemBF")%string.
  End oidMemBFPrefix.
  Module oidMemBFPredType <: OidMemBFPredType oids :=
    OidPredTypeFuncStemExtTerm oids oidLInstPredType OidPredTypeRootBaseConc
    OidPredTypeRootConc oidMemBFPrefix.
  Module oidMemBFPred : OidMemBFPred oids oidMemBFPredType :=
    HBCL_0_1_OidsPred oidMemBFPredType.
  Module oidMemFBPrefix <: OidPredPrefix.
    Definition HBCL_Oid_Prefix := ("MemFB")%string.
  End oidMemFBPrefix.
  Module oidMemFBPredType <: OidMemFBPredType oids :=
    OidPredTypeFuncStemExtTerm oids oidLInstPredType OidPredTypeRootBaseConc
    OidPredTypeRootConc oidMemFBPrefix.
  Module oidMemFBPred :
    OidMemFBPred oids oidMemFBPredType :=
    HBCL_0_1_OidsPred oidMemFBPredType.
  Definition HBCL_OidUT := oidUTPred.PredOid.
  Definition HBCL_OidTT := oidTTPred.PredOid.
  Definition HBCL_OidLLib := oidLLibPred.PredOid.
  Definition HBCL_OidLInst := oidLInstPred.PredOid.
  Definition HBCL_OidMemBF := oidMemBFPred.PredOid.
  Definition HBCL_OidMemFB := oidMemFBPred.PredOid.
  Module OidMapMod(opt : OidPredType oids)(op : OidsPred oids opt) :=
    FMapWeakList.Make op.PredoidDecidable.
  Module LInstMapModComplete <:
    FMapInterface.WSfun oidLInstPred.PredoidDecidable
  :=
    FMapWeakList.Make oidLInstPred.PredoidDecidable.
  Module LInstMapModRaw <: FMapIfaceRaw oidLInstPred.PredoidDecidable
    with Definition t := fun elt ⇒ list (HBCL_OidLInst × elt)
:=
      LInstMapModComplete.Raw.
  Module LInstMapModPred <: FMapModDatImplPred oidLInstPred.PredoidDecidable
      LInstMapModRaw.
    Definition NoDupType : ∀ elt,
      LInstMapModRaw.t elt → Prop := fun elt ⇒
        SetoidList.NoDupA (@LInstMapModRaw.PX.eqk elt).
  End LInstMapModPred.
Module LInstMapMod <:
    FMapIfaceRF oidLInstPred.PredoidDecidable
    LInstMapModRaw LInstMapModPred with Definition key := HBCL_OidLInst
    with Module Raw := LInstMapModRaw.
  Include LInstMapModComplete.
  Definition Build_t := Build_slist.
End LInstMapMod.
  Module LLibMapModComplete <:
    FMapInterface.WSfun oidLLibPred.PredoidDecidable
  :=
    FMapWeakList.Make oidLLibPred.PredoidDecidable.
  Module LLibMapModRaw <: FMapIfaceRaw oidLLibPred.PredoidDecidable
    with Definition t := fun elt ⇒ list (HBCL_OidLLib × elt)
:=
      LLibMapModComplete.Raw.

  Module LLibMapModPred <: FMapModDatImplPred oidLLibPred.PredoidDecidable
      LLibMapModRaw.

    Definition NoDupType : ∀ elt,
      LLibMapModRaw.t elt → Prop := fun elt ⇒
        SetoidList.NoDupA (@LLibMapModRaw.PX.eqk elt).

  End LLibMapModPred.

Module LLibMapMod <:
    FMapIfaceRF oidLLibPred.PredoidDecidable
    LLibMapModRaw LLibMapModPred with Definition key := HBCL_OidLLib
    with Module Raw := LLibMapModRaw.

  Include LLibMapModComplete.
  Definition Build_t := Build_slist.

End LLibMapMod.

Definition cutAfterFirstOmitDiscrim
  (A : _)(eqbA : A → A → bool)(x : A)(l : list A) :
  (list A × list A)
  :=
  let fix indexOfFirst
    (A : _)(eqbA : A → A → bool)(x : A)(l : list A) : nat :=
    match l with
      | e :: l′ ⇒
        match eqbA e x with
          | true ⇒ O
          | false ⇒ S (indexOfFirst _ eqbA x l′)
        end
      | nil ⇒ O
    end in
    let n := indexOfFirst _ eqbA x l
      in (firstn n l, skipn (n + 1) l).
Implicit Arguments cutAfterFirstOmitDiscrim [A].

  Definition liblessInst : HBCL_OidLInst → Prop.
Admitted.

  Definition splitLib(l : HBCL_OidLLib) :
    HBCL_OidLLib + HBCL_OidLLib × HBCL_OidLLib.
  Defined.

  Definition concatLibInst(l : HBCL_OidLLib)(i : HBCL_OidLInst) :
     HBCL_OidLInst.
  Defined.

  Definition concatInstMemBF(i : sig liblessInst)(m : HBCL_OidMemBF) :
     HBCL_OidMemBF.
  Defined.

  Definition concatInstMemFB(i : sig liblessInst)(m : HBCL_OidMemFB) :
     HBCL_OidMemFB.
  Defined.

  Definition splitLInstOid(i : HBCL_OidLInst) :
    sig liblessInst + (HBCL_OidLLib × HBCL_OidLInst).
Defined.

  Definition splitLiblessLInstOid(i : sig liblessInst) :
    sig liblessInst + (sig liblessInst × sig liblessInst).
Defined.

  Definition instLessMemBF : HBCL_OidMemBF → Prop.
Admitted.

  Definition instLessMemFB : HBCL_OidMemFB → Prop.
Admitted.

  Definition splitMemBFOid(m : HBCL_OidMemBF) :
    sig instLessMemBF + (sig liblessInst × HBCL_OidMemBF).
  Defined.
  Definition splitMemFBOid(m : HBCL_OidMemFB) :
    sig instLessMemFB + (sig liblessInst × HBCL_OidMemFB).
  Defined.
End HBCL_0_1_Oid_S.
```

D.4.3 The bit field type system

Listing D.11: The bit field type system

```
Require Export HBCL.HBCL_0_1.ModSignatures.UTypeSys.
Require Export HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Module HBCL_0_1_L_UTS <: UTypeSys.    Require Import Coq.Structures.Equalities.
  Require Import Coq.Program.Program.
  Require Coq.Lists.SetoidList.
  Require Import Coq.Arith.NatOrderedType.
  Require Import Coq.Arith.Arith_base.
  Require Import Coq.Logic.Eqdep_dec.
  Require Import Coq.Logic.EqdepFacts.
  Require Import Coq.Lists.List.
  Require Import Coq.Lists.SetoidList.
  Require Coq.FSets.FMapFacts.
  Require Import HBCL.Util.ListLemmas.
  Require Import HBCL.Util.sigTypes.
  Import HBCL_0_1_Id_S.
  Module VaridMapWFacts :=
    FMapFacts.WFacts_fun varidPred.PredidDecidable VaridMapMod.
  Module VaridMapWPties :=
    FMapFacts.WProperties_fun varidPred.PredidDecidable VaridMapMod.

  Inductive LBasetype : Set :=
  | BasetypeBool : LBasetype.
  Definition lbt_eqb(bt bt′ : LBasetype) : bool :=
    match bt, bt with BasetypeBool, BasetypeBool ⇒ true end.
  Theorem lbt_eqb_eq(bt bt′ : LBasetype) : lbt_eqb bt bt′ = true ↔ bt = bt′.

  Import HBCL_0_1_Id_S.
  Require Coq.FSets.FMapWeakList.

  Require Import Coq.Arith.Wf_nat.
  Definition Size := nat.
  Inductive LTypeRaw(s : Size) : Type :=
  | LBaseType : LBasetype → s = 1 → LTypeRaw s
  | LTupleType : list (sigT LTypeRaw) → LTypeRaw s
  | LRecordType : VaridMapModRaw.t (sigT LTypeRaw) →
    LTypeRaw s.
  Inductive LTypeRawCeiling(sceil: Size) : sigT LTypeRaw → Prop :=
  | LTypeRawC_intro(raw : sigT LTypeRaw) : projT1 raw < sceil →
    LTypeRawCeiling sceil raw.
  Definition mapSize(sceil : Size)(_ : Varid)
    (dat : sig (LTypeRawCeiling sceil))(s : Size) :=
    match dat with
      | exist raw _ ⇒ s + (projT1 raw)
    end.

  Definition stripCeiling(sceil : Size)(rc : sig (LTypeRawCeiling sceil)) :
    sigT LTypeRaw := ‘rc.
  Program Definition raiseCeiling(s s′ : Size)(prfs : s ≤ s′)
    (rc : sig (LTypeRawCeiling s)): sig (LTypeRawCeiling s′) := rc.
  Obligation 1.
  Notation SCP1 s rc := (projT1 (stripCeiling s rc)).
  Notation SCP2 s rc := (projT2 (stripCeiling s rc)).
  Lemma leAddInvar : ∀ n n′ n″, n = n′ + n″ → n″ ≤ n.
  Implicit Arguments leAddInvar [n n′ n″].
  Notation LTmapOK s :=
    (SetoidList.NoDupA (@VaridMapModRaw.PX.eqk (sig (LTypeRawCeiling s)))).
  Inductive LTypeP(s : Size) : LTypeRaw s → Prop :=
  | LBaseTypeP(bt : LBasetype)(prfs : s = 1) : LTypeP s (LBaseType s bt prfs)
  | TupleTypeP
    (lts : list (sig (LTypeRawCeiling (s)))) : LTypesP (s) lts →
    LTypeP s (LTupleType (s) (List.map (stripCeiling (s)) lts))
  | RecordTypeP(rm : VaridMapModRaw.t (sig (LTypeRawCeiling (s))))
        (mok : (LTmapOK s) rm):
    LRTypesP (s) rm → LTypeP s (LRecordType (s)
     (VaridMapMod.this (VaridMapMod.map (stripCeiling (s))
       (VaridMapMod.Build_slist mok))))

  with LTypesP(s : Size) : list (sig (LTypeRawCeiling s)) → Prop :=
  | LTypes(lts : list (sig (LTypeRawCeiling s))) :
    s = 1 + List.fold_right (listSize s) 0 lts →
    (∀ t, List.In t lts → LTypeP (projT1 (stripCeiling s t))
        (projT2 (stripCeiling s t))) → LTypesP s lts

  with LRTypesP(s : Size) : VaridMapModRaw.t (sig (LTypeRawCeiling s)) →
    Prop :=
  | RTypes(ltr : VaridMapModRaw.t (sig (LTypeRawCeiling s)))
    (mok : (LTmapOK s) ltr) :
    (s = 1 + (VaridMapMod.fold (mapSize s) (VaridMapMod.Build_slist mok) 0)) →
    (∀ v t, VaridMapMod.MapsTo v t (VaridMapMod.Build_slist mok) →
      LTypeP (projT1 (stripCeiling s t)) (projT2 (stripCeiling s t))) →
    LRTypesP s ltr.
  Program Definition EmptyLTypesP(s : Size)(seq : s = 1) : LTypesP s nil :=
    LTypes s nil _ _.
  Obligation 2.
  Definition LTypePS(s : Size) := sig (LTypeP s).
  Definition LTypesPS(s : Size) := sig (LTypesP s).
  Definition LRTypesPS(s : Size) := sig (LRTypesP s).
  Definition BuildBaseTypePS(s : Size)(bt : LBasetype)(seq : s = 1) :=
    exist (LTypeP s) (LBaseType s bt seq) (LBaseTypeP s bt seq).
  Definition BuildLTypesNil(s : Size)(seq : s = 1) :=
    exist (LTypesP s) nil (EmptyLTypesP s seq).
  Definition LRTypesPSRecoverMap(s : Size)(lrtps : LRTypesPS s) :
    VaridMapMod.t (sig (LTypeRawCeiling (s))).
Defined.
  Notation ” “ t ” := (proj2_sig t) (at level 10, t at next level).
  Definition buildLTypePSFromTS(s : Size)(ts : LTypesPS s) : LTypePS s :=
    exist (LTypeP s) (LTupleType s (List.map (stripCeiling s) (‘ts)))
    (TupleTypeP s (‘ts) (“ts)).
  Program Definition LTypesRecoverMok(s : Size)(lrtps : LRTypesPS s) :
    sig (LTmapOK s) := lrtps.
  Obligation 1.
  Definition buildLTypePSFromRT(s : Size)(tr : LRTypesPS s) : LTypePS s :=
    (exist (LTypeP s)
      (LRecordType s (VaridMapModRaw.map (stripCeiling s)(‘tr)))
      (RecordTypeP s (proj1_sig (LTypesRecoverMok s tr))
        (proj2_sig (LTypesRecoverMok s tr)) (“tr))).
Print LTypeP.
 Inductive LTypePSEq(s s′ : Size) : LTypePS s → LTypePS s′ → Prop :=
  | LTE_intro_bt(bt1 bt2 : LBasetype)(seq : s = 1)(seq′: s′ = 1) : bt1 = bt2 →
    LTypePSEq s s′ (BuildBaseTypePS s bt1 seq) (BuildBaseTypePS s′ bt2 seq′)
  | LTE_intro_tt(lts : LTypesPS s)(lts′ : LTypesPS s′) :
    LTypesPSEq s s′ lts lts′ →
    LTypePSEq s s′ (buildLTypePSFromTS s lts) (buildLTypePSFromTS s′ lts′)
  | LTE_intro_rt(ltr : VaridMapModRaw.t (sig (LTypeRawCeiling s)))
    (ltr′ : VaridMapModRaw.t (sig (LTypeRawCeiling s′))) :
    ∀ mok mok′ mok2 mok2′ ltprf ltprf′ sfprf sfprf′,
    LRTypesPSEq s s′ (exist (LRTypesP s) ltr (RTypes s ltr mok sfprf ltprf))
    (exist (LRTypesP s′) ltr′ (RTypes s′ ltr′ mok′ sfprf′ ltprf′)) →
    LTypePSEq s s′ (exist (LTypeP s) (LRecordType s (VaridMapMod.this
      (VaridMapMod.map (stripCeiling s) (VaridMapMod.Build_slist mok))))
      (RecordTypeP s ltr mok (RTypes s ltr mok2 sfprf ltprf)))
    (exist (LTypeP s′) (LRecordType s′ (VaridMapMod.this
      (VaridMapMod.map (stripCeiling s′) (VaridMapMod.Build_slist mok′))))
      (RecordTypeP s′ ltr′ mok′ (RTypes s′ ltr′ mok2′ sfprf′ ltprf′)))

  with LTypesPSEq(s s′ : Size) : LTypesPS s → LTypesPS s′ → Prop :=
  | LTE_intro_tup(ls : LTypesPS s)(ls′ : LTypesPS s′) : s = s′ →
    length (‘ls) = length (‘ls′) →
    (∀ els, List.In els (List.combine (‘ls) (‘ls′)) →
      ∃ si, ∃ si′,
      ∃ pr : ((LTypePS si) × (LTypePS si′)),
        LTypePSEq si si′ (fst pr) (snd pr) ∧
        eq_dep Size LTypeRaw (projT1 (proj1_sig (fst els)))
        (projT2 (proj1_sig (fst els))) si (proj1_sig (fst pr)) ∧
        eq_dep Size LTypeRaw (projT1 (proj1_sig (snd els)))
        (projT2 (proj1_sig (snd els))) si′ (proj1_sig (snd pr)))
    → LTypesPSEq s s′ ls ls′

  with LRTypesPSEq(s s′ : Size) : LRTypesPS s → LRTypesPS s′ → Prop :=
  | LTE_intro_rec(ltr : VaridMapModRaw.t (sig (LTypeRawCeiling s)))
    (ltr′ : VaridMapModRaw.t (sig (LTypeRawCeiling s′))) : ∀
    (mok : (LTmapOK s) ltr) (mok′ : (LTmapOK s′) ltr′)
    (sfprf : (s = 1 + (VaridMapMod.fold (mapSize s)
      (VaridMapMod.Build_slist mok) 0)))
    (sfprf′ : (s′ = 1 + (VaridMapMod.fold (mapSize s′)
      (VaridMapMod.Build_slist mok′) 0)))
    (ltprf : (∀ v t, VaridMapMod.MapsTo v t
      (VaridMapMod.Build_slist mok) →
      LTypeP (projT1 (stripCeiling s t)) (projT2 (stripCeiling s t))))
    (ltprf′ : (∀ v t, VaridMapMod.MapsTo v t
      (VaridMapMod.Build_slist mok′) →
      LTypeP (projT1 (stripCeiling s′ t)) (projT2 (stripCeiling s′ t)))),
    s = s′ →
    (∀ v : Varid, VaridMapMod.In v (VaridMapMod.Build_slist mok) ↔
      (VaridMapMod.In v (VaridMapMod.Build_slist mok′))) →
    (∀ (v : Varid)(el : sig (LTypeRawCeiling s))
      (el′ : sig (LTypeRawCeiling s′)), s = s′ →
      VaridMapMod.MapsTo v el (VaridMapMod.Build_slist mok) →
      VaridMapMod.MapsTo v el′ (VaridMapMod.Build_slist mok′) →
      ∃ si, ∃ si′,
       ∃ pel : (LTypePS si), ∃ pel′ : (LTypePS si′),
         LTypePSEq si si′ pel pel′ ∧
         eq_dep Size LTypeRaw (projT1 (proj1_sig el))
         (projT2 (proj1_sig el)) si (proj1_sig pel) ∧
         eq_dep Size LTypeRaw (projT1 (proj1_sig el′))
         (projT2 (proj1_sig el′)) si′ (proj1_sig pel′)) →
      LRTypesPSEq s s′ (exist (LRTypesP s) ltr (RTypes s ltr mok sfprf ltprf))
     (exist (LRTypesP s′) ltr′ (RTypes s′ ltr′ mok′ sfprf′ ltprf′)).

Implicit Arguments existT [A P].

Definition TypeS := LTypePS.
Definition ProtoT := sigT LTypePS.
Definition ProtoEqTSigT(t1 t2 : ProtoT) :=
  LTypePSEq (projT1 t1) (projT1 t2) (projT2 t1) (projT2 t2).

Definition LTypePSListEqSigT(tl1 tl2 : list (sigT LTypePS)) :=
  SetoidList.eqlistA ProtoEqTSigT tl1 tl2.

Definition LTypePSMapEqSigT(tl1 tl2 : VaridMapMod.t (sigT LTypePS)) :=
  VaridMapMod.Equiv ProtoEqTSigT tl1 tl2.
Infix "=t=" := ProtoEqTSigT (at level 70, no associativity).
Implicit Arguments existT [A P].
Definition extractType(n : nat)(ts : sigT LTypesPS)
  (prf : n < length (‘ (projT2 ts))) : sigT LTypePS.
Definition extractTypeR
  (v : Varid)(tr : sigT LRTypesPS)
    (prf : VaridMapMod.In v (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)))
 : sigT LTypePS.
  Definition sigifyLRaw3(s : Size)
    (rm : sigT LTypeRaw)(prf : LTypeP (projT1 rm)(projT2 rm)) :
    sigT (LTypePS) :=
    existT (projT1 rm) (exist (LTypeP (projT1 rm)) (projT2 rm) prf).
  Fixpoint sigifyListInner(s : Size)(l : (LTypesPS s))
          (dl : list (sig (LTypeRawCeiling (s))))
          (prf : List.incl dl (‘l)) { struct dl } : list (sigT LTypePS).
   Definition sigifyList(s : Size)(l : (LTypesPS s)) :
     list (sigT LTypePS) := sigifyListInner s l (proj1_sig l)
          (List.incl_refl (proj1_sig l)).
Lemma sigifyListLenInnerPrfIrrel : ∀ s ol il il′ p p′,
  il = il′ →
  length (sigifyListInner s ol il p) =
  length (sigifyListInner s ol il′ p′).
Lemma sigifyListLenEq : ∀ ts,
  length (sigifyList (projT1 ts) (projT2 ts)) = length (‘ (projT2 ts)).
Lemma LTCeilingListCombEq : ∀ s (tc tc′ : sig (LTypeRawCeiling s))
  (ltc ltc′ : list ( sig (LTypeRawCeiling s))),
  List.In (tc, tc′) (List.combine ltc ltc′) →
  List.map (stripCeiling s) ltc = List.map (stripCeiling s) ltc′ →
  ‘tc = ‘tc′.

Lemma LTMapsToRawEq : ∀ s s′ (tc : sig (LTypeRawCeiling s))
  (tc′ : sig (LTypeRawCeiling s′))
  (v : Varid)(ltr : VaridMapModRaw.t (sig (LTypeRawCeiling s)))
    (ltr′ : VaridMapModRaw.t (sig (LTypeRawCeiling s′)))
    (mok : (LTmapOK s) ltr) (mok′ : (LTmapOK s′) ltr′), s = s′ →
  VaridMapMod.Raw.map (stripCeiling s) ltr =
  VaridMapMod.Raw.map (stripCeiling s′) ltr′ →
  VaridMapMod.MapsTo v tc (VaridMapMod.Build_slist mok) →
  VaridMapMod.MapsTo v tc′ (VaridMapMod.Build_slist mok′) →
  ‘tc = ‘tc′.
Lemma LTypeMapEqExist : ∀ v s s′ (e : sig (LTypeRawCeiling s))
  (ltr : VaridMapMod.t (sig (LTypeRawCeiling s)))
  (ltr′ : VaridMapMod.t (sig (LTypeRawCeiling s′))),
  VaridMapMod.MapsTo v e ltr → s = s′ →
  VaridMapMod.map (stripCeiling s) ltr =
  VaridMapMod.map (stripCeiling s′) ltr′ →
  ∃ e′ : sig (LTypeRawCeiling s′), VaridMapMod.MapsTo v e′ ltr′.
Lemma LTypeMapEqExistWeak : ∀ v s s′ (e : sig (LTypeRawCeiling s))
  (ltr : VaridMapModRaw.t (sig (LTypeRawCeiling s)))
  (ltr′ : VaridMapModRaw.t (sig (LTypeRawCeiling s′))),
  VaridMapMod.Raw.PX.MapsTo v e ltr → s = s′ →
  VaridMapMod.Raw.map (stripCeiling s) ltr =
  VaridMapMod.Raw.map (stripCeiling s′) ltr′ →
  (LTmapOK s) ltr → (LTmapOK s′) ltr′ →
  ∃ e′ : sig (LTypeRawCeiling s′), VaridMapMod.Raw.PX.MapsTo v e′ ltr′.
Lemma LTypePSPrfIrrel : ∀ s s′ (t : LTypePS s) (t′ : LTypePS s′),
  EqdepFacts.eq_dep Size LTypeRaw s (‘t) s′ (‘t′) →
  LTypePSEq s s′ t t′.

Lemma sigifyListInnerTequivPrfIrrel : ∀ s ol il il′ p p′,
  il = il′ → LTypePSListEqSigT
  (sigifyListInner s ol il p) (sigifyListInner s ol il′ p′).
Lemma existTIdent : ∀ (A : Type) (P : A → Type) (x : sigT P),
x = existT (projT1 x) (projT2 x).
Record NonDepLTPPair : Type := { ndltps : Size;
  ndltpType1 : LTypePS ndltps; ndltpType2 : LTypePS ndltps }.
Definition NonDepLTPPairMeas(p : NonDepLTPPair) := ndltps p.
Definition NonDepLTPPairWF :=
  well_founded_ltof NonDepLTPPair NonDepLTPPairMeas.
Program Definition sigifyLRaw(s : Size)
  (rm : sigT LTypeRaw)(l : (LTypesPS s))(prf : List.In
  rm (List.map (stripCeiling s) l)) : LTypePS (projT1 rm).
Definition sigifyLCeil(s : Size)
  (tc : sig (LTypeRawCeiling s))(l : (LTypesPS s))
  (prf : List.In tc (‘l)) : LTypePS (projT1 (proj1_sig tc)).
Program Definition sigifyLCeilR(s : Size)(v : Varid)
  (rm : sig (LTypeRawCeiling s))(m : (LRTypesPS s))(prf : VaridMapMod.MapsTo
    v rm (VaridMapMod.map (stripCeiling s) (LRTypesPSRecoverMap s m))) :=
  LTypePS (projT1 rm).
Definition sigifyLCeil2R(s : Size)(v : Varid)
  (rm : sig (LTypeRawCeiling s))(m : (LRTypesPS s))(prf : VaridMapMod.MapsTo
    v rm (LRTypesPSRecoverMap s m)) :
  LTypePS (projT1 (proj1_sig rm)).
Definition sigifyLRaw2(s : Size)
  (rm : sigT LTypeRaw)(prf : LTypeP (projT1 rm)(projT2 rm)) :
  LTypePS (projT1 rm) :=
  exist (LTypeP (projT1 rm)) (projT2 rm) prf.

Section SigifyMapLemmasS.
  Variable s : Size.
  Variable l : list (Varid × sig (LTypeRawCeiling s)).
  Variable dl : list (Varid × sig (LTypeRawCeiling s)).
  Hypothesis inclprf : inclA (@VaridMapMod.eq_key_elt _) dl l.
  Hypothesis inprf : ∀ pr : Varid × sig (LTypeRawCeiling s),
    InA (@VaridMapMod.eq_key_elt _) pr l →
    LTypeP (projT1 (‘ (snd pr))) (projT2 (‘ (snd pr))).
  Variable t : Varid × sig (LTypeRawCeiling s).
  Variable ts : list (Varid × sig (LTypeRawCeiling s)).
  Hypothesis H′ : t :: ts = dl.
  Lemma LTypePInclImpl : LTypeP (SCP1 s (snd t)) (SCP2 s (snd t)).
  Lemma LTypeInclInd : inclA (@VaridMapMod.eq_key_elt _) ts l.
  Lemma InAImplInclInd : ∀ pr : Varid × sig (LTypeRawCeiling s),
    InA (@VaridMapMod.eq_key_elt _) pr l →
    LTypeP (projT1 (‘ (snd pr))) (projT2 (‘ (snd pr))).
End SigifyMapLemmasS.
Lemma inclReflLRPS : ∀ s (m : LRTypesPS s),
  SetoidList.inclA (@VaridMapMod.eq_key_elt _) (‘m) (‘m).
Lemma InprfFromLRPS : ∀ s (m : LRTypesPS s) pr,
  SetoidList.InA (@VaridMapMod.eq_key_elt _) pr
  (VaridMapMod.elements (LRTypesPSRecoverMap s m)) →
  LTypeP (projT1 (‘ (snd pr))) (projT2 (‘ (snd pr))).
Fixpoint sigifyMapInner(s : Size)
  (l dl : list (Varid × (sig (LTypeRawCeiling s))))
  (inclprf : SetoidList.inclA (@VaridMapMod.eq_key_elt _ ) dl l)
  (inprf : ∀ pr : (Varid × (sig (LTypeRawCeiling s))),
    SetoidList.InA (@VaridMapMod.eq_key_elt _) pr l →
    LTypeP (projT1 (‘ (snd pr))) (projT2 (‘ (snd pr)))) { struct dl } :
  VaridMapMod.t (sigT LTypePS) :=
   match dl as dl return dl = _ → VaridMapMod.t (sigT LTypePS) with
     | nil ⇒ fun _ ⇒ (VaridMapMod.empty (sigT LTypePS))
     | (cons t ts) ⇒ fun J : (cons t ts) = dl ⇒
        VaridMapMod.add (fst t) (sigifyLRaw3 s (stripCeiling s (snd t))
          (LTypePInclImpl s l dl inclprf inprf t ts J))
        (sigifyMapInner s l ts
          (LTypeInclInd s l dl inclprf t ts J)
          (InAImplInclInd s l dl inclprf inprf))
    end eq_refl.
   Definition sigifyMap(s : Size)(m : (LRTypesPS s)) :
     VaridMapMod.t (sigT LTypePS) :=
     sigifyMapInner s (VaridMapMod.elements (LRTypesPSRecoverMap s m))
       (VaridMapMod.elements (LRTypesPSRecoverMap s m)) (inclReflLRPS s m)
       (InprfFromLRPS s m).


   Infix "=v=" := varidPred.PredidDecidable.eq (at level 70, no associativity).

Lemma sigifyExtractTSEquiv : ∀ s s′ ts ts′ t n n′ lp lprf ncp,
   LTypePSEq s s′ (buildLTypePSFromTS s ts) (buildLTypePSFromTS s′ ts′)
   → n = n′ →
     existT (projT1 (‘t))
       (sigifyLCeil s′ t ts′ (nth_certain_in (n := n) lp ncp)) =t=
     extractType n′ (existT s ts) lprf.

Lemma sigifyExtractTREquiv : ∀ s s′ v v′ t lrt mok sfprf ltprf mp tr inprf,
   LTypePSEq s s′
   (buildLTypePSFromRT s (exist (LRTypesP s) lrt (RTypes s lrt mok sfprf ltprf)))
   (buildLTypePSFromRT s′ tr) →
   v =v= v′ →
   existT (projT1 (‘t)) (sigifyLCeil2R s v t
     (exist (LRTypesP s) lrt (RTypes s lrt mok sfprf ltprf)) mp) =t=
   extractTypeR v′ (existT s′ tr) inprf.

Lemma extractTypeRInvar : ∀ v v′ tr t inprf mprf, v =v= v′ →
   existT (projT1 (‘t)) (sigifyLCeil2R (projT1 tr) v t (projT2 tr) mprf) =t=
     extractTypeR v′ tr inprf.
Print VaridMapMod.Raw.find.
Admitted.

Lemma sigifyInTRIff : ∀ v tr,
   VaridMapMod.In v (sigifyMap (projT1 tr) (projT2 tr)) ↔
     VaridMapMod.In v (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)).

Lemma sigifyMapMapsToInv : ∀ tr v t t′,
   VaridMapMod.MapsTo v t (sigifyMap (projT1 tr) (projT2 tr)) →
   VaridMapMod.MapsTo v t′ (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)) →
   existT (projT1 t) (‘ (projT2 t)) = ‘t′.

   Definition LTypeListEqualWFS(l1 l2 : sigT (LTypesPS))
     (func : sigT LTypePS → sigT LTypePS → bool) : bool :=
     let cmpPair (bin : bool)(p : (sigT LTypePS) × (sigT (LTypePS))) :=
       (bin && func (fst p)(snd p))%bool in
       let lSig1 := sigifyList (projT1 l1)(projT2 l1) in
         let lSig2 := sigifyList (projT1 l2)(projT2 l2) in
           (List.fold_left cmpPair (List.combine lSig1 lSig2) true)%bool.

   Definition LTypeListEqualWFS2(sceil : Size)
     (l1 l2 : list (sig (LTypeRawCeiling sceil)))
     (func : (sig (LTypeRawCeiling sceil)) → (sig (LTypeRawCeiling sceil)) →
       bool) : bool :=
     let cmpPair(bin : bool)
       (p : (sig (LTypeRawCeiling sceil)) × (sig (LTypeRawCeiling sceil))) :=
       (bin && func (fst p)(snd p))%bool in
       ((List.fold_left cmpPair (List.combine l1 l2) true))%bool.

   Definition sigTLTypePSInCeil(sceil : Size)(t : sigT LTypePS) :=
     projT1 t < sceil.

   Definition sigifyLRaw4(s : Size)
     (rm : sigT LTypeRaw)(prft : LTypeP (projT1 rm)(projT2 rm))
     (prfc : projT1 rm < s):
     sig (sigTLTypePSInCeil s) :=
     exist (sigTLTypePSInCeil s)
     (existT (projT1 rm) (exist (LTypeP (projT1 rm)) (projT2 rm) prft))
     prfc.

   Program Definition sigifyLRaw5(s : Size)
     (rmc : sig (LTypeRawCeiling s))
     (prft : LTypeP (projT1 (‘rmc))(projT2 (‘rmc))) :
     sig (sigTLTypePSInCeil s) :=
     sigifyLRaw4 s (‘rmc)(prft) _.
```

```
  Obligation 1.
  Definition sigifyList2(s : Size)(l : (LTypesPS s)) :
   list (sig (sigTLTypePSInCeil s)).
  Definition LTypeListEqualWFS3(sceil : Size)(l1 l2 : (LTypesPS sceil))
     (func : sig (sigTLTypePSInCeil sceil) →
        sig (sigTLTypePSInCeil sceil) → bool) : bool :=
     let cmpPair(bin : bool)
        (p : (sig (sigTLTypePSInCeil sceil)) × (sig (sigTLTypePSInCeil sceil)))
        :=
        (bin && func (fst p)(snd p))%bool in
        let lSig1 := sigifyList2 sceil l1 in
           let lSig2 := sigifyList2 sceil l2 in
              ((List.fold_left cmpPair (List.combine lSig1 lSig2) true))%bool.
  Definition ConvertSTLTPS (s s′ : Size)(t′ : LTypePS s′)
     (prfseq : s = s′) : (LTypePS s).
Defined.
  Implicit Arguments ConvertSTLTPS [s′].
  Definition addCeiling(sceil : Size)(raw : sigT LTypeRaw)
     (prfs : projT1 raw < sceil) : (sig (LTypeRawCeiling sceil)) :=
     exist (LTypeRawCeiling sceil) raw (LTypeRawC_intro sceil raw prfs).
 Definition sizeListRawInner(el : sigT LTypeRaw)(cs : Size) :=
     cs + projT1 el.
  Definition sizeListRaw(l : list (sigT LTypeRaw)) : Size :=

        List.fold_right sizeListRawInner 0 l.
 Lemma listSizeMonot : ∀ (s : Size)(n p : nat)
     (a : sig (LTypeRawCeiling s)),
     listSize s a n + p = listSize s a (n + p).
  Lemma sizeListLeMonot : ∀ (n p : nat)(a : sigT LTypeRaw),
  n ≤ p → n ≤ sizeListRawInner a p.
  Lemma InListImpliesSmaller : ∀ (l : list (sigT LTypeRaw))
     (t : sigT LTypeRaw), List.In t l → projT1 t ≤ sizeListRaw l.
  Definition addCeilingFromList(sceil : Size)(dat : (sigT LTypeRaw))
     (prf : ∃ lts : list (sigT LTypeRaw),
     (∀ t : (sigT LTypeRaw), List.In t lts →
        (LTypeRawCeiling sceil t)) ∧ List.In dat lts ) :
     sig (LTypeRawCeiling sceil).
  Program Fixpoint sigifyToCeiling(sceil : Size)(lts : list (sigT LTypeRaw))
     (prf : ∀ t : (sigT LTypeRaw), List.In t lts →
        (LTypeRawCeiling sceil t)) : list (sig (LTypeRawCeiling sceil)) :=
     match lts with
        | nil ⇒ nil
        | (t :: lts′)%list ⇒
           ((addCeiling sceil t _) :: (sigifyToCeiling sceil lts′ _))%list
     end.
  Obligation 1.
  Obligation 2.
  Lemma sizeListEquiv : ∀(s : Size)(t : sig (LTypeRawCeiling s)),
     sizeListRaw [stripCeiling s t] = listSize s t 0.

  Lemma monotNatRFold : ∀(A : Type)(func : A → nat → nat),
     (∀ (n p : nat)(a : A), func a n + p = func a (n + p)) →
     (∀ (n′ p′ : nat)
        (l : list A), List.fold_right func n′ l + p′ =
        List.fold_right func (n′ + p′) l).
   Implicit Arguments monotNatRFold [A].

  Lemma sizeListEquivMap :
     ∀ (s : Size)(lts : list (sig (LTypeRawCeiling s))),
     sizeListRaw (List.map (stripCeiling s) lts) =
     List.fold_right (listSize s) 0 lts.

  Lemma sigifyInv : ∀ (sceil : Size)(lts : list (sigT LTypeRaw))
```

```
    (prf : ∀ t : (sigT LTypeRaw), List.In t lts →
       (LTypeRawCeiling sceil t)),
    lts = (List.map (stripCeiling sceil) (sigifyToCeiling sceil lts prf)).
  Lemma CeilingSubsetPrf : ∀(sceil : Size)(lts : list (sigT LTypeRaw)),
    (∀ t : (sigT LTypeRaw), List.In t lts →
       (∃ tc : (LTypeRawCeiling sceil t),
          LTypeP (SCP1 sceil (exist (LTypeRawCeiling sceil) t tc))
             (SCP2 sceil (exist (LTypeRawCeiling sceil) t tc)))) →
    (∀ t : (sigT LTypeRaw), List.In t lts →
       (LTypeRawCeiling sceil t)).
  Lemma CeilingInImpl : ∀ (sceil : Size)(lts : list (sigT LTypeRaw))
    (prf : ∀ t : (sigT LTypeRaw), List.In t lts →
       (LTypeRawCeiling sceil t)),
    ∀ rt : sig (LTypeRawCeiling sceil),
       List.In rt (sigifyToCeiling sceil lts prf) → List.In ('rt) lts.
  Lemma CeilingInImpl2 : ∀ (sceil : Size)(raw : sigT LTypeRaw)
    (lts : list (sig (LTypeRawCeiling sceil)))(prfs : projT1 raw < sceil),
    List.In raw (List.map (stripCeiling sceil) lts) →
    ∃ rceil : LTypeRawCeiling sceil raw,
       List.In (exist (LTypeRawCeiling sceil) raw rceil) lts.
    Check proj2_sig a.
  Qed.
  Definition sigifyToCeiling2(sceil : Size)(lts : list (sigT LTypeRaw))
    (prf : (∀ t : (sigT LTypeRaw), List.In t lts →
       (∃ tc : (LTypeRawCeiling sceil t),
          LTypeP (SCP1 sceil (exist (LTypeRawCeiling sceil) t tc))
          (SCP2 sceil (exist (LTypeRawCeiling sceil) t tc)))))
    (prfC : sceil = 1 + sizeListRaw lts) :
    LTypesPS sceil.
  Defined.
  Definition LTypeEqb(ltp : NonDepLTPPair) : bool.
Defined.
  Inductive UBasetype(lt : LBasetype) : Set :=
    UBTBool : lt = BasetypeBool → bool → UBasetype lt.

  Inductive UDataRaw : Type :=
  | UBaseData(blt : LBasetype)(ubt : UBasetype blt) : UDataRaw
  | UTupleData(s : Size)(ts : LTypesPS s) : list UDataRaw → UDataRaw
  | URecordData(s : Size)(tr : LRTypesPS s)
    (rm : VaridMapModRaw.t UDataRaw) :
    UDataRaw.

  Definition LTypePSEqHet(s s′ : Size)(t : LTypePS s)(t′ : LTypePS s′)
    :=
    LTypePSEq s s′ t t′.



  Inductive UDataP : sigT LTypePS →
    UDataRaw → Prop :=
  | UBaseDataP(blt : LBasetype)(t : sigT LTypePS)
    (blu : UBasetype blt) :
    LTypePSEq (projT1 t) 1 (projT2 t) (BuildBaseTypePS 1 blt eq_refl) →
    UDataP t (UBaseData blt blu)
  | UTupleDataP(ts : sigT LTypesPS)(t : sigT LTypePS)
    (lus : list UDataRaw) :
    LTypePSEq (projT1 t)(projT1 ts) (projT2 t)
    (buildLTypePSFromTS (projT1 ts)(projT2 ts)) →
    UTupleP ts lus → UDataP t
    (UTupleData (projT1 ts)(projT2 ts) lus)
  | URecordDataP(tr : sigT LRTypesPS)(t : sigT LTypePS)
   (lur : VaridMapMod.t UDataRaw) :
    LTypePSEq (projT1 t) (projT1 tr) (projT2 t)
    (buildLTypePSFromRT (projT1 tr)(projT2 tr)) →
    URecordP tr (VaridMapMod.this lur) →
    UDataP t
    (URecordData (projT1 tr)(projT2 tr)
      (VaridMapMod.this lur))

  with UTupleP : sigT LTypesPS → list UDataRaw → Prop :=
  | UTuplePIntro (ts : sigT LTypesPS)(lus : list (UDataRaw)) :
    (∃ lus′ : list ((sigT LTypePS) × UDataRaw),
    LTypePSListEqSigT (fst (List.split lus′))
    (sigifyList (projT1 ts) (projT2 ts)) ∧
    (snd (List.split lus′)) = lus ∧
    (∀ pr, List.In pr lus′ → UDataP (fst pr) (snd pr))) →
    UTupleP ts lus




  with URecordP : sigT LRTypesPS →
    VaridMapModRaw.t UDataRaw → Prop :=
  | URecordPIntro (tr : sigT LRTypesPS)
    (lur : VaridMapMod.t UDataRaw) :
    (∃ tr′ : sigT LRTypesPS,
      LRTypesPSEq (projT1 tr)(projT1 tr′)(projT2 tr)(projT2 tr′) ∧
      (∀ v, VaridMapMod.In v
        (LRTypesPSRecoverMap (projT1 tr)(projT2 tr)) →
        ∃ v′, v =v= v′ ∧
        VaridMapMod.In v lur) ∧
      (∀ v u ,
          VaridMapMod.MapsTo v u lur → (∃ v′, v′ =v= v ∧
            (∃ t: (sig (LTypeRawCeiling (projT1 tr))),
              ∃ t′ : sigT LTypePS,

                (eq_dep Size LTypeRaw (projT1 (‘t)) (projT2 (‘t))
                  (projT1 t′) (‘ (projT2 t′))) ∧

                UDataP t′ u ∧
                VaridMapMod.MapsTo v′ t
                (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)))))) →
    URecordP tr (VaridMapMod.this lur).
Lemma LTypePSRefl : ∀ s t, LTypePSEq s s t t.
Lemma LTypePSSym : ∀ s s′ t t′, LTypePSEq s s′ t t′ → LTypePSEq s′ s t′ t.
Lemma LTypePSTrans : ∀ s s′ s″ t t′ t″,
  LTypePSEq s s′ t t′ → LTypePSEq s′ s″ t′ t″ → LTypePSEq s s″ t t″.
Definition ProtoEqT := ProtoEqTSigT.
Lemma ProtoTEqTSigTRefl : ∀ t, ProtoEqTSigT t t.
Lemma ProtoTEqTSigTSym : ∀ t t′, ProtoEqTSigT t t′ → ProtoEqTSigT t′ t.
Lemma ProtoTEqTSigTTrans : ∀ t t′ t″,
  ProtoEqTSigT t t′ → ProtoEqTSigT t′ t″ → ProtoEqTSigT t t″.
  Add Relation (ProtoT) (ProtoEqTSigT)
  reflexivity proved by (@ProtoTEqTSigTRefl)
  symmetry proved by (@ProtoTEqTSigTSym)
    transitivity proved by (@ProtoTEqTSigTTrans)
      as ProtoTEqTSigT_rel.
Lemma UDataConvertP(t1 t2 : ProtoT)(u : UDataRaw)
  (p : (UDataP t1 u))(teq : t1 =t= t2) : (UDataP t2 u).
Implicit Arguments UDataConvertP [t1 t2 u].
Definition UDataConvert(t1 t2 : ProtoT)
  (u : sig (UDataP t1))(teq : t1 =t= t2) : sig (UDataP t2) :=
  exist (UDataP t2) (‘u) (UDataConvertP (“u) teq).

  Definition UDataPST(t : sigT LTypePS) := sig (UDataP t).
  Definition UTuplePST(t : sigT LTypesPS) := sig (UTupleP t).
  Definition URecordPST(t : sigT LRTypesPS) := sig (URecordP t).
  Theorem UDataInhabited : ∀ t, inhabited (UDataPST t).
  Definition UDataPS := sigTD UDataP.
  Definition UTuplePS := sigTD UTupleP.
  Definition URecordPS := sigTD URecordP.

  Definition DataR := UDataRaw.
  Definition DataP := UDataP.
  Definition ProtoU := UDataPST.

  Definition LTypeSConv(s s′ : Size)(t : LTypePS s′)(prfs : s = s′) : LTypePS s.
  Defined.

  Definition ProtoEqbTSigT(t1 t2 : ProtoT) :=
    match Nat_as_UBE.eqb (projT1 t1) (projT1 t2) as deq return
      Nat_as_UBE.eqb (projT1 t1) (projT1 t2) = deq → bool with
      | true ⇒ fun H : (beq_nat (projT1 t1) (projT1 t2) = true) ⇒
        (LTypeEqb (Build_NonDepLTPPair (projT1 t1) (projT2 t1)
          (LTypeSConv (projT1 t1) (projT1 t2) (projT2 t2)
            (proj1 (Nat_as_UBE.eqb_eq (projT1 t1) (projT1 t2)) H))))
      | false ⇒ fun _ ⇒ false
    end eq_refl.
Definition LTypesPSEqSigT(ts ts′ : sigT LTypesPS) :=
  LTypesPSEq (projT1 ts) (projT1 ts′) (projT2 ts) (projT2 ts′).
Definition LRTypesPSEqSigT(tr tr′ : sigT LRTypesPS) :=
  LRTypesPSEq (projT1 tr) (projT1 tr′) (projT2 tr) (projT2 tr′).
Lemma UTupleConvertP(ts1 ts2 : sigT LTypesPS)(ul : list UDataRaw)
  (p : (UTupleP ts1 ul))(teq : LTypesPSEqSigT ts1 ts2) : (UTupleP ts2 ul).
Lemma URecordConvertP(tr1 tr2 : sigT LRTypesPS)
  (ur : VaridMapModRaw.t UDataRaw)
  (p : (URecordP tr1 ur))(teq : LRTypesPSEqSigT tr1 tr2) : (URecordP tr2 ur).
Definition buildProtoTFromSigTS(ts : sigT LTypesPS) :=
    (existT (projT1 ts)
      (buildLTypePSFromTS (projT1 ts) (projT2 ts))).
Definition buildProtoTFromSigTR(tr : sigT LRTypesPS) :=
    (existT (projT1 tr)
      (buildLTypePSFromRT (projT1 tr) (projT2 tr))).
Lemma LTypesPSInjEq : ∀ ts ts′, (buildProtoTFromSigTS ts) =t=
   (buildProtoTFromSigTS ts′) → LTypesPSEqSigT ts ts′.
Lemma LRTypesPSInjEq : ∀ tr tr′, (buildProtoTFromSigTR tr) =t=
   (buildProtoTFromSigTR tr′) → LRTypesPSEqSigT tr tr′.
  Theorem ProtoTSigT_eqb_eq : ∀ t1 t2 : ProtoT,
    ProtoEqbTSigT t1 t2 = true ↔ ProtoEqTSigT t1 t2.
  Definition BTBoolSP1(lbt : LBasetype) : LTypePS 1 :=
    exist (LTypeP 1) (LBaseType 1 lbt eq_refl) (LBaseTypeP 1 lbt eq_refl).
  Definition BTBoolSPT(lbt : LBasetype) : ProtoT :=
    existT 1 (BTBoolSP1 lbt).
  Definition buildLTypePSFromTSSigT(ts : sigT LTypesPS) : sigT LTypePS :=
    let t := buildLTypePSFromTS (projT1 ts) (projT2 ts)
      in existT (projT1 ts) t.
  Definition genTupleTFromTS(s : Size)(ts : LTypesPS s) :=
    existT (P := LTypePS) s (exist (LTypeP s)
      (LTupleType s (List.map (stripCeiling s) (‘ts)))
      (TupleTypeP s (‘ts) (“ts))).
  Definition genRecordTFromTR(s : Size)(tr : LRTypesPS s) :=
    existT (P := LTypePS) s (exist (LTypeP s)
      (LRecordType s
        (VaridMapModRaw.map (stripCeiling s)(‘tr)))
      (RecordTypeP s (proj1_sig (LTypesRecoverMok s tr))
        (proj2_sig (LTypesRecoverMok s tr)) (“tr))).
  Definition LTypePSEqHetEx(s s′ : Size)(t : LTypePS s)(t′ : LTypePS s′) :=
    LTypePSEqHet s s′ t t′ .
  Module LTypeTypMod <: Typ with Definition t := ProtoT.
    Definition t := ProtoT.
  End LTypeTypMod.
  Module LTypeHasEqMod <: HasEq LTypeTypMod.
    Definition eq := ProtoEqTSigT.
  End LTypeHasEqMod.
  Module LTypeEqMod := LTypeTypMod <+ LTypeHasEqMod.
  Module LTypeEqNotationMod <: EqNotation LTypeEqMod.
    Include EqNotation LTypeEqMod.
  End LTypeEqNotationMod.
  Module LTypeEq'Mod := LTypeEqMod <+ LTypeEqNotationMod.
  Module LTypeHasEqBoolMod <: HasEqBool LTypeEq'Mod.
    Definition eqb := ProtoEqbTSigT.
    Definition eqb_eq := ProtoTSigT_eqb_eq.
  End LTypeHasEqBoolMod.
  Module LTypeHasEqDecMod := HasEqBool2Dec LTypeEqMod LTypeHasEqBoolMod.
  Module LTypeIsEqMod <: IsEq LTypeEqMod.
  Instance eq_equiv : Equivalence LTypeEqMod.eq := ProtoTEqTSigT_rel.
  End LTypeIsEqMod.
  Module LTypeDecidableTypeMod <: Equalities.DecidableType
    with Definition t := ProtoT
      := LTypeEqMod <+ LTypeIsEqMod <+ LTypeHasEqDecMod.

End HBCL_0_1_L_UTS.
```

D.4.4 The untimed OID type system functor

Listing D.12: The untimed OID type system functor

```
Require Export HBCL.HBCL_0_1.ModSignatures.UTypeSys.
Require Export HBCL.HBCL_0_1.ModSignatures.Oids.
Require Export HBCL.HBCL_0_1.ModSignatures.UTypeSysOid.
Require Import HBCL.Util.sigTypes.

Module HBCL_0_1_L_UTSOid (Import uts : UTypeSys)(opm : OidPreds) <:
  UTypeSysOid uts opm.

  Inductive TOid(o : opm.HBCL_OidUT)
    (t : ProtoT ) : Type :=
    MakeOidT : TOid o t.

  Definition T := TOid.

  Definition TEq(ut1 ut2 : sigTD T) :=
    opm.oidUTPred.PredoidDecidable.eq (projTD1 ut1) (projTD1 ut2) ∧
    ProtoEqT (projTD2 ut1) (projTD2 ut2).

  Definition VOid := fun (o : opm.HBCL_OidUT)(t : ProtoT)(to : T o t)
    (u : ProtoU t) ⇒ Type.

  Definition V := VOid.

End HBCL_0_1_L_UTSOid.
```

D.4.5 The harmonic type system functor

Listing D.13: The harmonic type system functor

```
Module HTypeSys_F (Import opm : OidPreds)
  (UTSparam : UTypeSys)(UTSOidParam : UTypeSysOid UTSparam opm) <:
  HTypeSys UTSparam opm UTSOidParam.

  Inductive TimedTLocal :
    HBCL_OidTT → Freq → sigTD UTSOidParam.T → Type :=
  | TType : ∀ (ou : HBCL_OidUT)(ot : HBCL_OidTT)
    (f : Freq)(ut : sigTD UTSOidParam.T),
    TimedTLocal ot f ut.
  Definition TimedT := TimedTLocal.
  Inductive TimedTEqLoc(tt1 tt2 : sigTT TimedT) : Prop :=
    TimeTLocalEq_Intro :
    oidTTPred.PredoidDecidable.eq (projTT1 tt1) (projTT1 tt2) →
    Freq_eq (projTT2 tt1) (projTT2 tt2) →
    UTSOidParam.TEq (projTT3 tt1) (projTT3 tt2) →
    TimedTEqLoc tt1 tt2.
  Definition TimedTEq := TimedTEqLoc.
  Inductive TimedVLocal(ttimed : sigTT TimedT)(ttime : TTime (projTT2 ttimed)) :
    Type :=
    | MakeTimedV(u : option (UTSparam.ProtoU (projTD2 (projTT3 ttimed)))) :

        TimedVLocal ttimed ttime.
  Definition TimedV := TimedVLocal.
  Definition TimedTF(f : Freq) := fun o u ⇒ TimedT o f u.
  Definition TimedTFEq (tt1 tt2 : sigTT TimedTF) : Prop.
Admitted.
  Definition TimedVF(f : Freq) := sigTD (TimedTF f) → TTime f → Type.
End HTypeSys_F.
```

D.4.6 The expression language type classes

Listing D.14: The untimed box type

```
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.ModSignatures.Ids.
Require Import HBCL.HBCL_0_1.ModSignatures.Oids.
Require Import HBCL.HBCL_0_1.ModSignatures.UTypeSys.
Require Import HBCL.HBCL_0_1.ModSignatures.UTypeSysOid.
Require Import HBCL.HBCL_0_1.ModSignatures.UCost.
Require Import HBCL.HBCL_0_1.ModSignatures.UBox.
Module SF_UBoxEmptyEnc(Import ipm : IdPreds)(Import opm : OidPreds)
  (Import UTSparam : UTypeSys)(Import UTSOidParam : UTypeSysOid UTSparam opm)
  (Import uc : UCost UTSparam) <: UBox ipm opm UTSparam UTSOidParam uc.
  Definition Encoding := Empty_set.
  Implicit Arguments UPot [T uraw u CTDT CTDTP cb].
  Definition InpOutpTypes(CTDT : Type)
      (CTDTP : ProtoT → CTDT → Prop) := ipm.VaridMapMod.t
    ({t : UTSparam.ProtoT &
      {o : opm.HBCL_OidUT & UTSOidParam.T o t} &
      sig (CTDTP t)} × nat × nat).
  Definition UDataPSTMatchesInpOutpTypes(CTDT : Type)
    (CTDTP : ProtoT → CTDT → Prop)
    (inpTypes : InpOutpTypes CTDT CTDTP)
    (udat : ipm.VaridMapMod.t (sigT UDataPST)) : Prop :=
    ipm.VaridMapMod.Equiv ProtoEqT
    (ipm.VaridMapMod.map (projT1 (P := UDataPST)) udat)
    (ipm.VaridMapMod.map
      (fun inpOutpType ⇒ (sigTypes.projT1sigT2
        (P := fun t ⇒
          {o : opm.HBCL_OidUT & UTSOidParam.T o t})
        (Q := fun t ⇒ sig (CTDTP t)
      )) (fst (fst inpOutpType))) inpTypes).
  Implicit Arguments existT [A P].
Check Size.
Check DataR.
Check DataP.
  Record UExprLang := {
    CTDT : Type;
    CTDTP : ProtoT → CTDT → Prop;
    costB : CostBase TypeS CTDT CTDTP DataR DataP;
    AST : Set;
    parse : Encoding → AST;
    sso : InpOutpTypes CTDT CTDTP → InpOutpTypes CTDT CTDTP →
      Type;
    compile (itypes otypes : InpOutpTypes CTDT CTDTP) :
      option (sso itypes otypes);
    reduce (itypes otypes : InpOutpTypes CTDT CTDTP) :
      sso itypes otypes →
      sig (UDataPSTMatchesInpOutpTypes CTDT CTDTP itypes) →
      sig (UDataPSTMatchesInpOutpTypes CTDT CTDTP otypes)
  }.
End SF_UBoxEmptyEnc.
```

D.4.7 The harmonic box functor

Listing D.15: The harmonic box functor

```
Require Import Coq.QArith.QArith_base.
Require Import Coq.NArith.BinNat.
Require Import Coq.NArith.BinPos.
Require Import Coq.ZArith.ZArith_base.
Require Import Coq.Program.Program.
Require Import HBCL.Util.Freq.
Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.sigTypes.
Require Export HBCL.HBCL_0_1.ModSignatures.HBox.

Module MData (ipm : IdPreds)(Import opm : OidPreds)(Import uts : UTypeSys)
  (Import UTSOidParam : UTypeSysOid uts opm)
  (Import HTSparam : HTypeSys uts opm UTSOidParam) :
  MDataType ipm opm uts UTSOidParam HTSparam.

  Record MDatBoxFreqEltBase : Type :=
    { MDBFE_Base_Oid : oids.Oid;
      MDBFE_Base_Freq : Freq;
      MDBFE_Base_timt : sigTT HTSparam.TimedT;
      MDBFE_Base_TTFL : TTFL;
      MDBFE_Base_minSize : N;
      MDBFE_Base_maxSize : positive
    }.

  Definition MDatBoxElt := MDatBoxFreqEltBase.

  Inductive MemDatMode : Set := ReadEnabled | WriteEnabled.

  Definition memDatModeTimeRel(m : MemDatMode) : Z :=
    match m with
      | ReadEnabled ⇒ (-1)%Z
      | WriteEnabled ⇒ (1)%Z
    end.

  Definition DatListElType := (sigTD (TimedV )).

  Definition MemDatListRaw :=
    list DatListElType.

  Implicit Arguments existTD [A B P].
  Implicit Arguments existT [A P].

  Inductive MemDatListPred(mode : MemDatMode)
    (mdfe: MDatBoxFreqEltBase )
    (baseTime : TTime (MDBFE_Base_Freq mdfe))
    (ttime : TTime (projTT2 (MDBFE_Base_timt mdfe))) :
```

```
    MemDatListRaw → Prop :=
  | MemDatListBasePred
    (tv : (TimedV (MDBFE_Base_timt mdfe) ttime)) :
      TTseq ttime baseTime →
      MemDatListPred mode mdfe baseTime ttime
      (cons (existTD (MDBFE_Base_timt mdfe) ttime tv) nil)
  | MemDatInd(prevLastTime : TTime (projTT2 (MDBFE_Base_timt mdfe)))
    (tv : (TimedV (MDBFE_Base_timt mdfe) ttime))
    (mdlr' : MemDatListRaw ) :
    MemDatListPred mode mdfe baseTime prevLastTime mdlr' →
    ( (getTimeZ ttime) =
      (getTimeZ prevLastTime) + memDatModeTimeRel mode)%Z →
    ( (1 + (getTimeZ ttime) - (getTimeZ baseTime)))%Z
    = Zpos (MDBFE_Base_maxSize mdfe) →
    MemDatListPred mode mdfe baseTime ttime
      (cons (existTD (MDBFE_Base_timt mdfe) ttime tv) mdlr').

  Definition MemDatTime(mode : MemDatMode)
    (mdfe: MDatBoxFreqEltBase )
    (baseTime : TTime (MDBFE_Base_Freq mdfe))
    (ttime : TTime (projTT2 (MDBFE_Base_timt mdfe))) :=
    sig (MemDatListPred mode mdfe baseTime ttime).

  Definition MDatTimeElt :=

    MemDatListRaw.

End MData.

  Module SF_MemModBox(ipm : IdPreds)(opm : OidPreds)
  (uts : UTypeSys)
  (uc : UCost uts)(UTSOidParam : UTypeSysOid uts opm)
  (ubox : UBox ipm opm uts UTSOidParam uc)
  (HTSparam : HTypeSys uts opm UTSOidParam)
  (mdi : MDataType ipm opm uts UTSOidParam HTSparam)
    (ott : OidPredType opm.oids)
    (ot : OidsPred opm.oids ott)

    <:
    MemModBox ipm opm uts UTSOidParam HTSparam ott ot mdi.

    Module otd := ot.

    Module otm : FMapInterface.WSfun(otd.PredoidDecidable) :=
      FMapWeakList.Make otd.PredoidDecidable.

    Module otmWPties :=
      FMapFacts.WProperties_fun ot.PredoidDecidable otm.


    Import mdi.

    Definition MDatTimeMapRaw :=
      otm.t ( MDatTimeElt ).

  Definition MDatBoxTimeMapPred(mode : MemDatMode)
    (freqm : otm.t MDatBoxElt)(f : Freq)(t : TTime f)
    (mdm : MDatTimeMapRaw) : Prop.
Admitted.

  Definition MDatBoxTime(mode : MemDatMode)
    (freqm : otm.t MDatBoxElt)(f : Freq)(t : TTime f) :=
    sig (MDatBoxTimeMapPred mode freqm f t).

  End SF_MemModBox.

Module SF_HBox(ipm : IdPreds)(opm : OidPreds)
  (Import uts : UTypeSys)
  (Import uc : UCost uts)(UTSOidParam : UTypeSysOid uts opm)
  (Import ubox : UBox ipm opm uts UTSOidParam uc)
  (Import HTSparam : HTypeSys uts opm UTSOidParam) <:
  HBox ipm opm uts uc UTSOidParam ubox HTSparam.

  Module BoxTypeIdMapMod :
    FMapInterface.WSfun(ipm.boxidPred.PredidDecidable) :=
    FMapWeakList.Make ipm.boxidPred.PredidDecidable.
```

```
Module ipmModVaridLCM := MapLCM ipm.varidPred.PredidDecidable ipm.VaridMapMod.
Module ipmModBoxidLCM :=
  MapLCM ipm.boxidPred.PredidDecidable BoxTypeIdMapMod.
Module MDataInst <: MDataType ipm opm uts UTSOidParam HTSparam :=
  MData ipm opm uts UTSOidParam HTSparam.
Module InMemModBox <: MemModBox ipm opm uts UTSOidParam HTSparam
  opm.oidMemFBPredType opm.oidMemFBPred
    MDataInst
   :=
  SF_MemModBox ipm opm uts uc UTSOidParam ubox HTSparam MDataInst
  opm.oidMemFBPredType opm.oidMemFBPred.
Module OutMemModBox <: MemModBox ipm opm uts UTSOidParam HTSparam
  opm.oidMemBFPredType opm.oidMemBFPred MDataInst :=
  SF_MemModBox ipm opm uts uc UTSOidParam ubox HTSparam MDataInst
  opm.oidMemBFPredType opm.oidMemBFPred.
Import MDataInst.
Definition InMapVaridConvertPred
  (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
  (f : Freq)
  (ivm : InMemModBox.otm.t ipm.Varid)
  (vm : sig (vpred f))
  (im : InMemModBox.otm.t MDataInst.MDatBoxElt)
  := ∃ v,
∃ m, ipm.VaridMapMod.MapsTo v m (proj1_sig vm) →
    ∃ i, InMemModBox.otm.MapsTo i v ivm ∧
      InMemModBox.otm.MapsTo i m im.
  Definition InTypePredConvert
    (uexprlang : ubox.UExprLang)
    (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
    (f : Freq)
    (im : InMemModBox.otm.t MDataInst.MDatBoxElt)
    (utypes : InpOutpTypes _ _)
    (inpred : ∀(f : Freq)
      (freqm : sig (vpred f ))
      (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop)
    (ivm : InMemModBox.otm.t ipm.Varid) :=
    ∃ vm, InMapVaridConvertPred vpred f ivm vm im ∧
      inpred f vm utypes.
Definition OutMapVaridConvertPred
  (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
  (f : Freq)
  (ovm : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
  (om : OutMemModBox.otm.t MDataInst.MDatBoxElt)
(vm : sig (vpred f)) :=
  ∀ v, ∃ m, ipm.VaridMapMod.MapsTo v m (proj1_sig vm) →
    ∃ o, ipm.VaridMapMod.MapsTo v o ovm ∧
      OutMemModBox.otm.MapsTo o m om.
  Definition OutTypePredConvert
    (uexprlang : ubox.UExprLang)
    (vpred : Freq → ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop)
    (f : Freq)
    (om : OutMemModBox.otm.t MDataInst.MDatBoxElt)
    (utypes : InpOutpTypes _ (ubox.CTDTP uexprlang))
    (outpred : ∀(f : Freq)
      (freqm : sig (vpred f ))
      (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop)
    (ovm : ipm.VaridMapMod.t opm.HBCL_OidMemBF) :=
    ∃ vm, OutMapVaridConvertPred vpred f ovm om vm ∧
      outpred f vm utypes.
Record HBoxAbs := {
  uexprlang : ubox.UExprLang;
  boxfreqcorrectin : Freq →
    ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop;
  boxfreqcorrectout : Freq →
```

```
      ipm.VaridMapMod.t MDataInst.MDatBoxElt → Prop;
    IOtypePredIn : ∀(f : Freq)
      (freqm : sig (boxfreqcorrectin f ))
      (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop;
    IOtypePredOut : ∀(f : Freq)
      (freqm : sig (boxfreqcorrectout f ))
      (tco : InpOutpTypes _ (ubox.CTDTP uexprlang)), Prop;
    convertInp : ∀
      (f : Freq)(tf : TTime f)
      (memvarmap : InMemModBox.otm.t ipm.Varid)
      (ttypes : sig
        (fun tmap ⇒
          ∃ vm,
            InMapVaridConvertPred boxfreqcorrectin f memvarmap vm tmap))
      (utypes : InpOutpTypes _ _),
      InTypePredConvert uexprlang boxfreqcorrectin f (proj1_sig ttypes) utypes
      IOtypePredIn memvarmap →
      InMemModBox.MDatBoxTime MDataInst.ReadEnabled
      (proj1_sig ttypes) f tf →
      option (sig (ubox.UDataPSTMatchesInpOutpTypes _ _ utypes));
    convertOutp : ∀
      (f : Freq)(tf : TTime f)
      (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
      (ttypes : sig
        (fun tmap ⇒ ∃ vm,
            OutMapVaridConvertPred boxfreqcorrectout f varmemmap tmap vm))
      (utypes : InpOutpTypes _ (ubox.CTDTP uexprlang)),
      OutTypePredConvert uexprlang boxfreqcorrectout f (proj1_sig ttypes)
      utypes IOtypePredOut varmemmap →
      sig (ubox.UDataPSTMatchesInpOutpTypes _ _ utypes) →
      OutMemModBox.MDatBoxTime MDataInst.WriteEnabled (proj1_sig ttypes) f tf
  }.

    Inductive HBoxSSORawI : Type :=
    HBoxSSORaw_make
    (hBoxType : HBoxAbs)
    (itypes otypes : InpOutpTypes _ (ubox.CTDTP (uexprlang hBoxType)))
    (usso : ubox.sso (uexprlang hBoxType) itypes otypes) :

      HBoxSSORawI .
  Definition HBoxSSORaw := HBoxSSORawI.

  Module VaridMapWPties :=
    FMapFacts.WProperties_fun ipm.varidPred.PredidDecidable ipm.VaridMapMod.
  Inductive HBoxSSOPredI(f fi fo : Freq)
    (ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
    (ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
:
    HBoxSSORaw → Prop :=
  | HBoxSSO_intro
    (hBoxType : HBoxAbs)
    (itypes otypes : InpOutpTypes _ (ubox.CTDTP (uexprlang hBoxType)))
    (usso : ubox.sso (uexprlang hBoxType) itypes otypes)

    (ttmfInPred : boxfreqcorrectin hBoxType fi ttmfIn)
    (ttmfOutPred : boxfreqcorrectout hBoxType fo ttmfOut) :
    FreqIsLCM fi fo f →
    IOtypePredIn hBoxType fi (exist _ _ ttmfInPred) itypes →
    IOtypePredOut hBoxType fo (exist _ _ ttmfOutPred) otypes →
    VaridMapWPties.Disjoint itypes otypes →
    HBoxSSOPredI f fi fo ttmfIn ttmfOut
    (HBoxSSORaw_make hBoxType itypes otypes usso ).
  Definition HBoxSSOPred : ∀ (f fi fo : Freq)

    (ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
    (ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
,
    HBoxSSORaw → Prop := HBoxSSOPredI.
  Definition HBoxSSO(f fi fo : Freq)

    (ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
    (ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
    :=
    sig (HBoxSSOPred f fi fo ttmfIn ttmfOut).
  Definition HBoxStepPred(f fmi fmo : Freq)
    (t : TTime f )
    (ti : TTime fmi)(to : TTime fmo)
    (ttmfIn : _ )
    (ttmfOut : _ )
    (memvarmap : InMemModBox.otm.t ipm.Varid)
    (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
    (ttmfIn′ : _)
    (ttmfOut′ : _)
:
    HBoxSSO f fmi fmo ( ttmfIn) ( ttmfOut) →
    InMemModBox.MDatBoxTime MDataInst.ReadEnabled ( ttmfIn′)
    fmi ti →
    OutMemModBox.MDatBoxTime MDataInst.WriteEnabled ( ttmfOut′)
    fmo to →
    Prop.
Admitted.
  Module InMemBoxWPties :=
    FMapFacts.WProperties_fun opm.oidMemFBPred.PredoidDecidable
    InMemModBox.otm.
  Module OutMemBoxWPties :=
    FMapFacts.WProperties_fun opm.oidMemBFPred.PredoidDecidable
    OutMemModBox.otm.
      Definition mapVaridMemToOtmIn(vm : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
        (memvarmap : InMemModBox.otm.t ipm.Varid) :
        InMemModBox.otm.t MDataInst.MDatBoxElt.
      Defined.
      Definition mapVaridMemToOtmOut(vm : ipm.VaridMapMod.t MDataInst.MDatBoxElt)
        (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF) :
        OutMemModBox.otm.t MDataInst.MDatBoxElt.
      Defined.
      Lemma upcastBucketsDataPNOT_ENOUGH_ARGS : ∀
        (mtype : MDatBoxElt) e,
        DataP (projTD2 (projTT3 (MDBFE_Base_timt mtype))) e.
      Fixpoint upcastBucketsToTime(mtype : MDatBoxElt)
        (ul′ : list (option DataR))
        (tv : TTime (projTT2 (MDataInst.MDBFE_Base_timt mtype)))
        {struct ul′} : list
        (sigTD HTSparam.TimedV).
      Defined.
      Lemma genBlankBucketTrainFDivTTimeConvNOT_ENOUGH_ARGS :
        ∀ mspec fmo,
        FreqDivide (projTT2 (MDBFE_Base_timt mspec)) fmo.
      Definition genBlankBucketTrain(mspec : MDatBoxElt)
        (fmo : Freq)(to : TTime fmo) : list (sigTD HTSparam.TimedV).
      Defined.
  Definition genBlankOut(ttmfOut : OutMemModBox.otm.t MDatBoxElt)
    (fmo : Freq)(to : TTime fmo) :
    sig (OutMemModBox.MDatBoxTimeMapPred WriteEnabled ttmfOut fmo to).
  Defined.
  Definition HBoxStep(f fmi fmo : Freq)
    (t : TTime f )
    (ti : TTime fmi)(to : TTime fmo)
    (ttmfIn : _ )
    (ttmfOut : _ )
    (memvarmap : InMemModBox.otm.t ipm.Varid)
    (varmemmap : ipm.VaridMapMod.t opm.HBCL_OidMemBF)
    (ttmfIn' : _ )
    (ttmfOut' : _ )
    (hbox : HBoxSSO f fmi fmo ttmfIn ttmfOut)
    (inp : InMemModBox.MDatBoxTime MDataInst.ReadEnabled (ttmfIn') fmi ti) :
    sig (HBoxStepPred f fmi fmo t ti to
      ttmfIn ttmfOut memvarmap varmemmap ttmfIn' ttmfOut'
      hbox inp).
Defined.

  Record HBoxSSONonDep : Type := {
    HBoxSSONonDep_f : Freq;
    HBoxSSONonDep_fi : Freq;
    HBoxSSONonDep_fo : Freq;

    HBoxSSONonDep_ttmfIn : ipm.VaridMapMod.t MDataInst.MDatBoxElt
 ;
    HBoxSSONonDep_ttmfOut : ipm.VaridMapMod.t MDataInst.MDatBoxElt
 ;
    HBoxSSONonDep_HBoxSSO : HBoxSSO HBoxSSONonDep_f HBoxSSONonDep_fi
      HBoxSSONonDep_fo HBoxSSONonDep_ttmfIn
      HBoxSSONonDep_ttmfOut
  }.
End SF_HBox.
```

D.4.8 The coordination language functor

Listing D.16: The coordination language functor

```
Require Import Coq.Arith.Bool_nat.
Require Import Coq.ZArith.BinInt.
Require Import Coq.ZArith.Zminmax.
Require Import Coq.ZArith.Zbool.
Require Coq.FSets.FMapInterface.
Require Coq.FSets.FMapWeakList.
Require Import Coq.Program.Program.
Require Import HBCL.Util.Freq.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.ModSignatures.Ids.
Require Import HBCL.HBCL_0_1.ModSignatures.Oids.
Require Import HBCL.HBCL_0_1.ModSignatures.UTypeSys.
Require Import HBCL.HBCL_0_1.ModSignatures.UTypeSysOid.
Require Import HBCL.HBCL_0_1.ModSignatures.UCost.
Require Import HBCL.HBCL_0_1.ModSignatures.UBox.
Require Import HBCL.HBCL_0_1.ModSignatures.HBox.
Require Export HBCL.HBCL_0_1.ModSignatures.Coord.


Module SF_Coord
  (ipm : IdPreds)(opm : OidPreds)
  (Import uts : UTypeSys)
  (Import uc : UCost uts)(UTSOidParam : UTypeSysOid uts opm)
  (ubox : UBox ipm opm uts UTSOidParam uc)
  (Import HTSparam : HTypeSys uts opm UTSOidParam)
  (Import hbox : HBox ipm opm uts uc UTSOidParam ubox HTSparam) <:
  Coord ipm opm uts uc UTSOidParam ubox HTSparam hbox.
  Import ipm.
  Import opm.
  Import uts.
  Import UTSOidParam.
  Definition CoordAST := unit.
  Definition parse : ubox.Encoding → CoordAST := fun _ ⇒ tt.

  Module LInstOidMapMod := FMapWeakList.Make oidLInstPred.PredoidDecidable.
  Module MDataTypeInst <:
    MemDataTypeInstType
    ipm opm uts
    UTSOidParam HTSparam MDataInst :=
    MemDataTypeInst ipm opm uts
    UTSOidParam HTSparam MDataInst.
  Module SF_MemModInst(ott : OidPredType opm.oids)
    (ot : OidsPred opm.oids ott)
    (memBoxes : MemModBox ipm opm uts

    UTSOidParam HTSparam ott ot MDataInst
      ) <:
    MemModInst ipm opm uts
    UTSOidParam HTSparam ott ot MDataInst MDataTypeInst memBoxes.

    Definition MDatFreqMapRaw := memBoxes.otm.t (MDataTypeInst.MDatFreqElt).

    Definition RawFreqMapPred(f : Freq)(mf : MDatFreqMapRaw) : Prop.
Admitted.
    Definition MDatFreqMap(f : Freq) := sig (RawFreqMapPred f).

    Definition MDatFreqMapIOPred(f : Freq)(mdf : MDatFreqMap f) :
    Prop.
Admitted.

    Definition MDatFreqMapIO(f : Freq) := sig (MDatFreqMapIOPred f).

    Definition MDatFreqMapElt_seq : MDataTypeInst.MDatFreqElt →
      MDataTypeInst.MDatFreqElt → Prop.
Admitted.
    Definition MDatFreqMapEltOpt_seq(o1 : option (MDataTypeInst.MDatFreqElt))
      (o2 : option (MDataTypeInst.MDatFreqElt )) :=
      match o1, o2 with
        | Some mdf1, Some mdf2 ⇒ MDatFreqMapElt_seq mdf1 mdf2
        | _, _ ⇒ False
      end.

    Definition MDatMapFreqTimePred : ∀ f : Freq,
      ∀ mf :
      MDatFreqMap f, memBoxes.MDatTimeMapRaw → Prop.
Admitted.

    Definition MDatMapTime(f : Freq)
      (mf : MDatFreqMap f) := sig (MDatMapFreqTimePred f mf).
Print MDataTypeInst.MDatFreqElt.
Print MDataInst.MDatBoxFreqEltBase.
Print MDataInst.MDatTimeElt.

    Definition MDatMapModeReadPred(f : Freq)
     (t : TTime f)(mf : MDatFreqMap f)(mt : MDatMapTime f mf) :=
     (∀ oid : (sig ott.Pred), memBoxes.otm.In oid (proj1_sig mt)) ∧
     ∀ (oid : (sig ott.Pred))(tme : MDataInst.MDatTimeElt ),
       memBoxes.otm.MapsTo oid tme (proj1_sig mt) →
       ∃ mfe, memBoxes.otm.MapsTo oid mfe (‘mf) ∧
         ∃ t′, ∃ t″, TTseq t t′ ∧ TTseq t t″ ∧
       MDataInst.MemDatListPred MDataInst.ReadEnabled
       (MDataTypeInst.MDFE_boxMemDat mfe) t′ t″ tme.

    Definition MDatMapModeWritePred(f : Freq)
     (t : TTime f)(mf : MDatFreqMap f)(mt : MDatMapTime f mf) :=
     (∀ oid : (sig ott.Pred), memBoxes.otm.In oid (proj1_sig mt)) ∧
     ∀ (oid : (sig ott.Pred))(tme : MDataInst.MDatTimeElt ),
        memBoxes.otm.MapsTo oid tme (proj1_sig mt) →
        ∃ mfe, memBoxes.otm.MapsTo oid mfe ('mf) ∧
          ∃ t', ∃ t'', TTseq t t' ∧ TTseq t t'' ∧
        MDataInst.MemDatListPred MDataInst.WriteEnabled
        (MDataTypeInst.MDFE_boxMemDat mfe) t' t'' tme.
End SF_MemModInst.
Module InMemModInst : MemModInst ipm opm uts

   UTSOidParam HTSparam
   oidMemFBPredType oidMemFBPred MDataInst MDataTypeInst InMemModBox :=
   SF_MemModInst oidMemFBPredType oidMemFBPred InMemModBox.
Module OutMemModInst : MemModInst ipm opm uts

   UTSOidParam HTSparam
   oidMemBFPredType oidMemBFPred MDataInst MDataTypeInst OutMemModBox :=
   SF_MemModInst oidMemBFPredType oidMemBFPred OutMemModBox.


Record LInstSignatureRaw : Type := {
   InstSigFreqMemIn : Freq;
   InstSigFreqMemOut : Freq;
   InstSigInputMems : InMemModInst.MDatFreqMapIO InstSigFreqMemIn;
   InstSigOutputMems : OutMemModInst.MDatFreqMapIO InstSigFreqMemOut
}.
Definition LInstSignature(f : Freq) :=
   { lisr : LInstSignatureRaw |
      FreqDivide (InstSigFreqMemIn lisr) f ∧
      FreqDivide (InstSigFreqMemOut lisr) f
   }.
Inductive LInstSSORaw : Type :=
   LInstSSORaw_make(fmi fmo fmil fmol fmin fmon fl fn f : Freq) :

   InMemModInst.MDatFreqMap fmil →
   OutMemModInst.MDatFreqMap fmol →
   InMemModInst.MDatFreqMap fmin →
   OutMemModInst.MDatFreqMap fmon →



   BoxTypeIdMapMod.t
   (HBoxSSONonDep × (InMemModBox.otm.t Varid)
      × (VaridMapMod.t opm.HBCL_OidMemBF)) →
   InMemModBox.otm.t (HBCL_OidMemFB) →
   OutMemModBox.otm.t (HBCL_OidMemBF) →
   VaridMapMod.t (HBCL_OidMemBF × HBCL_OidMemFB) →

   LInstMapMod.Raw.t
      ( LInstSSORaw ) →
   LInstMapMod.t HBCL_OidLInst →
LInstMapMod.t
((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw) →

   LLibMapMod.Raw.t (LibSSORaw ) →
LInstMapMod.t TTFL →

LLibMapMod.t ( LInstMapMod.t LInstSignatureRaw) →

LInstSSORaw

with LibSSORaw : Type :=
   LLibSSORaw_make :

   VaridMapMod.t (sigTD T) →
   VaridMapMod.t (sigTT TimedT) →
   LInstMapMod.Raw.t (LInstSSORaw ) →
LInstMapMod.t
```

```
((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw) →

   LLibMapMod.Raw.t (LibSSORaw ) →
LLibMapMod.t ( LInstMapMod.t LInstSignatureRaw) →

   LibSSORaw .

Definition HBoxMapPredF(f : Freq)(h : BoxTypeIdMapMod.t
   (HBoxSSONonDep × (InMemModBox.otm.t ipm.Varid)
      × (ipm.VaridMapMod.t opm.HBCL_OidMemBF))) :=
ipmModBoxidLCM.FreqIsLCMMapFunc _ HBoxSSONonDep_f f
(BoxTypeIdMapMod.map (fun m ⇒ fst (fst m)) h).

Definition FMapLCMQualVarid(f : Freq) := sig (ipmModVaridLCM.FreqIsLCMMap f).

Module opmModLInstLCM :=
   MapLCM oidLInstPred.PredoidDecidable opm.LInstMapMod.

Definition FMapLCMQualLInst(f : Freq) := sig (opmModLInstLCM.FreqIsLCMMap f).

Module opmModLLibLCM :=
   MapLCM oidLLibPred.PredoidDecidable opm.LLibMapMod.

Definition FMapLCMQualLLib(f : Freq) := sig (opmModLLibLCM.FreqIsLCMMap f).

Definition HBoxLCMQual(f : Freq) := sig (HBoxMapPredF f).

Inductive LInstSSOPred :
   ∀ f, LInstSignature f → LInstMapMod.t LInstSignatureRaw →
      LInstSSORaw
   → Prop :=
   LInstSSO_intro

   (f fmi fmo fmil fmol fmin fmon fl fn f : Freq)

   (hboxSSO : sigT HBoxLCMQual )
   (obsMap : InMemModBox.otm.t (HBCL_OidMemFB) )
   (manifMap : OutMemModBox.otm.t
      (HBCL_OidMemBF) )
   (fifoMap : VaridMapMod.t (HBCL_OidMemBF × HBCL_OidMemFB))
   (instNestMap : LInstMapMod.t (LInstSSORaw))

   (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)

   (instTypeMap : LInstMapMod.t HBCL_OidLInst)
   (instTShiftMap : LInstMapMod.t TTFL)
(instSigDatMap : LInstMapMod.t
 ((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw))

   (libMap : LLibMapMod.t LibSSORaw)
   (libmapsigmap : LLibMapMod.t
      ( LInstMapMod.t LInstSignatureRaw))
   (instMemIn : InMemModInst.MDatFreqMapIO fmi)
   (instMemOut : OutMemModInst.MDatFreqMapIO fmo)
   (instMemInLoc : InMemModInst.MDatFreqMap fmil)
   (instMemOutLoc : OutMemModInst.MDatFreqMap fmol)
   (instMemInNest : InMemModInst.MDatFreqMap fmin)
   (instMemOutNest : OutMemModInst.MDatFreqMap fmon)
   (fmnest : sigT FMapLCMQualLInst)
   (fmtref : sigT FMapLCMQualLInst):

   FreqIsLCMList f (cons fmi (cons fmo (cons (projT1 hboxSSO)
      (cons (projT1 fmnest) (cons (projT1 fmtref) nil))))) →

   ((∀ v, LInstMapMod.In v (proj1_sig (projT2 fmnest)) ↔
      LInstMapMod.In v instNestMap) →
      (∀ v,
         ∀ lir, LInstMapMod.MapsTo v lir instNestMap →
            ∀ f', LInstMapMod.MapsTo v f' (proj1_sig (projT2 fmnest)) →
               ∃ fmi', ∃ fmo',
               ∃ inmems, ∃ outmems, ∃ instTypeScopeMap',
                  ∃ fsmatch,
                LInstSSOPred (f′) (exist (fun lisr ⇒
                   FreqDivide (InstSigFreqMemIn lisr) f′ ∧
                   FreqDivide (InstSigFreqMemOut lisr) f′)
                (Build_LInstSignatureRaw fmi′ fmo′ inmems outmems) fsmatch)
                instTypeScopeMap′ ( lir))
    ) →




    ∀ lisp,
    LInstSSOPred (f)
    (exist _ (Build_LInstSignatureRaw fmi fmo instMemIn instMemOut) lisp)
    instTypeScopeMap (LInstSSORaw_make fmi fmo fmil fmol
       fmin fmon fl fn f
       (instMemInLoc) (instMemOutLoc) (instMemInNest) (instMemOutNest)
       (proj1_sig (projT2 hboxSSO)) obsMap manifMap fifoMap
       (LInstMapMod.this instNestMap) instTypeMap
       instSigDatMap
       (LLibMapMod.this libMap)
    instTShiftMap libmapsigmap)

  with LLibSSOPred : LInstMapMod.t LInstSignatureRaw → LibSSORaw → Prop :=
    LLibSSO_intro
    (libsig : LInstMapMod.t LInstSignatureRaw)
    (libenv : LInstMapMod.Raw.t LInstSignatureRaw)
    (utypes : VaridMapMod.t (sigTD T))
    (ttypes : VaridMapMod.t (sigTT TimedT))
    (instMap : LInstMapMod.t LInstSSORaw)
(instSigDatMap : LInstMapMod.t
  ((sigT LInstSignature) × LInstMapMod.t LInstSignatureRaw))
    (fminst : sigT FMapLCMQualLInst)
    (libNestMap : LLibMapMod.t LibSSORaw)
    (libmapsigmap :
       LLibMapMod.t ( LInstMapMod.t LInstSignatureRaw)) :



((∀ v, LInstMapMod.In v (proj1_sig (projT2 fminst)) ↔
       LInstMapMod.In v instMap) →
       (∀ v,
          ∀ lir, LInstMapMod.MapsTo v lir instMap →
             ∀ f′, LInstMapMod.MapsTo v f′ (proj1_sig (projT2 fminst)) →
                ∃ fmi′, ∃ fmo′,
                ∃ inmems, ∃ outmems, ∃ instTypeScopeMap,
                   ∃ lisp,
                LInstSSOPred (f′)
                (exist _ (Build_LInstSignatureRaw fmi′ fmo′ inmems outmems) lisp)
                instTypeScopeMap lir)
    )
    →

  ((∀ v, VaridMapMod.In v ttypes → VaridMapMod.In v utypes) →
  ∀ v t u, VaridMapMod.MapsTo v t ttypes →
  VaridMapMod.MapsTo v u utypes → UTSOidParam.TEq u (projTT3 t)) →
    LLibSSOPred libsig (LLibSSORaw_make utypes ttypes
       (LInstMapMod.this instMap) instSigDatMap
       (LLibMapMod.this libNestMap) libmapsigmap).

  Definition LInstSSO(f : Freq)(linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
   :=
    sig (LInstSSOPred (f)
  linstsig instTypeScopeMap).

  Definition LLibSSO(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw) :=
    sig (LLibSSOPred instTypeScopeMap).
```

```
Inductive LissoLibPred : ∀
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw),
  LLibMapMod.t (sigT LLibSSO) → Prop :=.

Definition LissoLibDep(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw) :=
  sig (LissoLibPred instTypeScopeMap).

Definition LibClosRaw :=
  list ((LInstMapMod.t LInstSignatureRaw) × (sigT LissoLibDep)).

Inductive LibClosPred :
  LInstMapMod.t LInstSignatureRaw → LibClosRaw → Prop :=.

Module LInstMapModWPties :=
  FMapFacts.WProperties_fun opm.oidLInstPred.PredoidDecidable
  LInstMapMod.

Module LLibMapModWPties :=
  FMapFacts.WProperties_fun opm.oidLLibPred.PredoidDecidable
  LLibMapMod.

Definition LibClos(instTypeScopeMap : LInstMapMod.t LInstSignatureRaw) :=
  sig (LibClosPred instTypeScopeMap).

Definition Lisso(f : Freq)(linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissoLibMap : LibClos instTypeScopeMap) :=
  LInstSSO f linstsig instTypeScopeMap.

Definition StatSemObj := sigTQ Lisso.

Module Type MemsEnabledSpec.

  Parameter imemtrel : ∀ fmi, (TTime fmi) → (TTime fmi).
  Parameter omemtrel : ∀ fmo, (TTime fmo) → (TTime fmo).
  Parameter imemrwpred : ∀ (fmi : Freq)(tmi : TTime fmi)
     (mfi : InMemModInst.MDatFreqMap fmi),
     InMemModInst.MDatMapTime fmi mfi → Prop.
  Parameter omemrwpred : ∀ (fmo : Freq)(tmo : TTime fmo)
     (mfo : OutMemModInst.MDatFreqMap fmo),
     OutMemModInst.MDatMapTime fmo mfo → Prop.

End MemsEnabledSpec.

Module Type MemsEnabledModType(Import mes : MemsEnabledSpec).

  Parameter MemsEnabledPred : ∀ (f fmi fmo: Freq)
    (t : TTime f )(tmi : TTime fmi)(tmo : TTime fmo)
    (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
    (mti : InMemModInst.MDatMapTime fmi mfi)
    (mto : OutMemModInst.MDatMapTime fmo mfo), Prop.

End MemsEnabledModType.

Module MemsEnabledMod(Import mes : MemsEnabledSpec) : MemsEnabledModType mes.

  Implicit Arguments imemtrel [fmi].
  Implicit Arguments omemtrel [fmo].

  Definition MemsEnabledPred(f fmi fmo: Freq)
    (t : TTime f )(tmi : TTime fmi)(tmo : TTime fmo)
    (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
    (mti : InMemModInst.MDatMapTime fmi mfi)
    (mto : OutMemModInst.MDatMapTime fmo mfo) :=
     TTseq ( t) tmi ∧ TTseq ( t) tmo ∧
     imemrwpred fmi (imemtrel tmi) mfi mti ∧
     omemrwpred fmo (omemtrel tmo) mfo mto.

End MemsEnabledMod.

Module BoxesMemsEnabledSpec : MemsEnabledSpec.
  Definition imemtrel := tId.
  Definition omemtrel := tId.
  Definition imemrwpred := InMemModInst.MDatMapModeReadPred.
  Definition omemrwpred := OutMemModInst.MDatMapModeReadPred.
End BoxesMemsEnabledSpec.

Module MemBFMemsEnabledSpec : MemsEnabledSpec.
  Definition imemtrel := tId.
  Definition omemtrel := tNext.
  Definition imemrwpred := InMemModInst.MDatMapModeReadPred.
  Definition omemrwpred := OutMemModInst.MDatMapModeWritePred.
End MemBFMemsEnabledSpec.

Module FIFOsMemsEnabledSpec : MemsEnabledSpec.
  Definition imemtrel := tId.
  Definition omemtrel := tNext.
  Definition imemrwpred := InMemModInst.MDatMapModeReadPred.
  Definition omemrwpred := OutMemModInst.MDatMapModeReadPred.
End FIFOsMemsEnabledSpec.

Module MemFBMemsEnabledSpec : MemsEnabledSpec.
  Definition imemtrel := tNext.
  Definition omemtrel := tNext.
  Definition imemrwpred := InMemModInst.MDatMapModeWritePred.
  Definition omemrwpred := OutMemModInst.MDatMapModeReadPred.
End MemFBMemsEnabledSpec.

Module BoxesMemsEnabled : MemsEnabledModType BoxesMemsEnabledSpec
  := MemsEnabledMod BoxesMemsEnabledSpec.
Module MemBFMemsEnabled : MemsEnabledModType MemBFMemsEnabledSpec
  := MemsEnabledMod MemBFMemsEnabledSpec.
Module FIFOsMemsEnabled : MemsEnabledModType FIFOsMemsEnabledSpec
  := MemsEnabledMod FIFOsMemsEnabledSpec.
Module MemFBMemsEnabled : MemsEnabledModType MemFBMemsEnabledSpec
  := MemsEnabledMod MemFBMemsEnabledSpec.

Definition isNextTimeStep(f : Freq)(t t' : TTime f) :=
    TTseq (tNext f t) t'.

Lemma TTseq_refl : ∀ (f : Freq)(t : TTime f), TTseq t t.

Lemma tNextIsNextTimeStep : ∀ f t, isNextTimeStep f t (tNext f t).

Module Type StepModType(mespre mespost : MemsEnabledSpec).

  Declare Module SMemEnabledModPre : MemsEnabledModType mespre.
  Declare Module SMemEnabledModPost : MemsEnabledModType mespost.

  Parameter StepSSO : ∀ (f fmi fmo : Freq)
    (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo),
    Type.

  Parameter StepPred : ∀ (f fmi fmo : Freq)
    (t t' : TTime f )
    (ti ti' : TTime fmi)(to to' : TTime fmo)
    (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo),
    StepSSO f fmi fmo mfi mfo →
    isNextTimeStep f t t' →
    { mts |
    (SMemEnabledModPre.MemsEnabledPred f fmi fmo t ti to mfi mfo
      (fst mts) (snd mts)) } →
    { mts |
      (SMemEnabledModPost.MemsEnabledPred f fmi fmo t' ti' to' mfi mfo
      (fst mts) (snd mts)) } →
    Prop.

End StepModType.

Module Type StepMod(mespre mespost : MemsEnabledSpec)
  (stepmodtype : StepModType mespre mespost
   ).

  Parameter StepFunc : ∀ (f fmi fmo : Freq)
    (t t' : TTime f )
    (ti ti' : TTime fmi)(to to' : TTime fmo)
    (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
    (stepSSO : stepmodtype.StepSSO f fmi fmo mfi mfo)
    (nexttimeprf : isNextTimeStep f t t')
    (instate : { mts |
      (stepmodtype.SMemEnabledModPre.MemsEnabledPred f fmi fmo t ti to
        mfi mfo (fst mts) (snd mts)) }),
    sig (stepmodtype.StepPred f fmi fmo t t' ti ti' to to' mfi mfo stepSSO
      nexttimeprf instate).

End StepMod.
```

```
  Module BoxesStep .

    Inductive StepSSOPred(f fmi fmo : Freq)
      (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
      (ndmap : BoxTypeIdMapMod.t
         (HBoxSSONonDep × (InMemModBox.otm.t ipm.Varid)
      × (ipm.VaridMapMod.t opm.HBCL_OidMemBF))) : Prop :=
    | StepSSO_intro :

      StepSSOPred f fmi fmo mfi mfo ndmap.
    Definition StepSSO(f fmi fmo : Freq)
      (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
      := sig (StepSSOPred f fmi fmo mfi mfo).
    Definition StepPred(f fmi fmo : Freq)
      (t t′ : TTime f )
      (ti ti′ : TTime fmi)(to to′ : TTime fmo)
      (mfi : InMemModInst.MDatFreqMap fmi)
      (mfo : OutMemModInst.MDatFreqMap fmo) :
      StepSSO f fmi fmo mfi mfo →
      isNextTimeStep f t t′ →
      sig (InMemModInst.MDatMapModeReadPred _ ti mfi) →

      sig (OutMemModInst.MDatMapModeWritePred _ to mfo) →



      Prop.
Admitted.
  End BoxesStep.
  Lemma inclReflBoxid : ∀ elt m,
      SetoidList.inclA (@BoxTypeIdMapMod.eq_key_elt elt) m m.
  Module InMemBoxWPties :=
    FMapFacts.WProperties_fun opm.oidMemFBPred.PredoidDecidable
    InMemModBox.otm.
  Module OutMemBoxWPties :=
    FMapFacts.WProperties_fun opm.oidMemBFPred.PredoidDecidable
    OutMemModBox.otm.

  Inductive CoordStateRaw : Type :=
  | CoordStateRaw_make :
    InMemModBox.MDatTimeMapRaw → OutMemModBox.MDatTimeMapRaw
    →
    LInstMapModRaw.t ( CoordStateRaw) → CoordStateRaw.
Print Lisso.

  Inductive CoordStateStaticPred(f : Freq)

    (linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)

    (lisso : Lisso f linstsig instTypeScopeMap lissolib)

  :
    CoordStateRaw → Prop :=
  | CoordStatePred_intro(f′ fl fn fmi′ fmo′ : Freq)
    (mfi′ : InMemModInst.MDatFreqMap fmi′)
    (mfo′ : OutMemModInst.MDatFreqMap fmo′)
    (mfiTrace′ : InMemModInst.MDatMapTime fmi′ mfi′)
    (mfoTrace′ : OutMemModInst.MDatMapTime fmo′ mfo′)
    (nestCSR : LInstMapMod.t ( CoordStateRaw)) :

    CoordStateStaticPred f linstsig instTypeScopeMap
    lissolib lisso
    (CoordStateRaw_make
      (‘mfiTrace′) (‘mfoTrace′)
      (LInstMapMod.this nestCSR)).

 Inductive CoordStateBoxesEnabled(f : Freq)
  (t : TTime f)(linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=


 with CoordStateMemBFEnabled(f : Freq)
  (t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=


 with CoordStateInnerFIFOsEnabled(f : Freq)
  (t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=


 with CoordStateInnerMemFBEnabled(f : Freq)
  (t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  sig (CoordStateStaticPred f linstsig instTypeScopeMap lissolib lisso) →
  Prop :=.

  Definition CoordStateOuterFIFOsEnabled := CoordStateBoxesEnabled.

  Definition CoordStateOuterMemFBEnabled := CoordStateInnerMemFBEnabled.

 Record CSTempCorrectND
 : Type := {
   CSTempCorrectND_f : Freq;
   CSTempCorrectND_t : TTime CSTempCorrectND_f;
   CSTempCorrectND_linstsig : LInstSignature CSTempCorrectND_f;
   CSTempCorrectND_instTypeScopeMap : LInstMapMod.t LInstSignatureRaw;
   CSTempCorrectND_lissolib : LibClos CSTempCorrectND_instTypeScopeMap;
   CSTempCorrectND_lisso : Lisso CSTempCorrectND_f
     CSTempCorrectND_linstsig
     CSTempCorrectND_instTypeScopeMap
     CSTempCorrectND_lissolib;
   CSTempCorrectND_CS :
     sig (CoordStateStaticPred CSTempCorrectND_f
       CSTempCorrectND_linstsig
     CSTempCorrectND_instTypeScopeMap CSTempCorrectND_lissolib
     CSTempCorrectND_lisso)

 }.
 Definition CSTempCorrectND_seq :
 CSTempCorrectND →
 CSTempCorrectND → Prop.
Admitted.

Check InMemModInst.MDatMapModeReadPred.
 CoInductive InputStream(f : Freq)(mfi : InMemModInst.MDatFreqMapIO f)
```

```
    (t : TTime f) : Type :=
    | InputStreamFinal :

      InputStream f mfi t
    | InputStreamInd(tnext : TTime f) :
      TTseq (tNext f t) tnext →
      sig (InMemModInst.MDatMapModeReadPred f t (‘mfi)) →
      InputStream f mfi tnext → InputStream f mfi t.

  CoInductive OutputStream(f : Freq)(mfo : OutMemModInst.MDatFreqMapIO f)
    (t : TTime f) : Type :=
  | OutputStreamFinal :

    OutputStream f mfo t
  | OutputStreamInd(tnext : TTime f) :
    TTseq (tNext f t) tnext →
    sig (OutMemModInst.MDatMapFreqTimePred f (‘mfo)) →
    OutputStream f mfo tnext → OutputStream f mfo t.

  Definition LissoNestMatch(f : Freq)(linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
    (lisso : Lisso f linstsig instTypeScopeMap lissolib)
    (nestCSR : LInstMapMod.t LInstSSORaw) : Prop.
Admitted.

  Definition InstMapMatchSSO(f fn : Freq)(linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
    (lisso : Lisso f linstsig instTypeScopeMap lissolib)
    (csTemporalPred : ∀ f : Freq,
                          TTime f →
                          ∀ (linstsig : LInstSignature f)
                            (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
                            (lissolib : LibClos instTypeScopeMap)
                            (lisso : Lisso f linstsig instTypeScopeMap lissolib),
                          sig
                            (CoordStateStaticPred f linstsig instTypeScopeMap
                              lissolib lisso) → Prop)
    (nestCSR : LInstMapMod.t (
      (CSTempCorrectND) )) : Prop.
Admitted.

  Inductive InstBoxesNestPred
    (f fn : Freq)(t : TTime f)(linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
    (lisso : Lisso f linstsig instTypeScopeMap lissolib)

    (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso
      CoordStateInnerFIFOsEnabled ))
    (nestCSR′ : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso
      CoordStateOuterFIFOsEnabled ))
    (traceEnab : ∀ (f : Freq)(t : TTime f)
      (linstsig : LInstSignature f)
      (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
      (lissolib : LibClos instTypeScopeMap)
      (lisso : Lisso f linstsig instTypeScopeMap lissolib)
      (cstate :
        sig (CoordStateBoxesEnabled
          f t linstsig instTypeScopeMap lissolib lisso))
      ,
      Type) : Prop :=.

    CoInductive TraceBoxesEnab(f : Freq)(t : TTime f)
    (linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
```

```
   (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
   ∀

   (cstate : sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap
      lissolib lisso))

,
   Type :=
| TraceBoxesStep (fl fn : Freq)(fmti fmto : Freq)
   (tfl tfl′ : TTime fl)
   (t′ : TTime f)(ti′ : TTime (InstSigFreqMemIn (‘linstsig)))
   (tti tti′ : TTime fmti)(tto tto′ : TTime fmto)
   (mfti : InMemModInst.MDatFreqMap fmti)
   (mfto : OutMemModInst.MDatFreqMap fmto)
   (bssso : BoxesStep.StepSSO fl fmti fmto mfti mfto)
   (mftiState : InMemModInst.MDatMapTime fmti mfti)
   (mftoState : OutMemModInst.MDatMapTime fmto mfto)
   (mftiState′ : InMemModInst.MDatMapTime fmti mfti)
   (mftoState′ : OutMemModInst.MDatMapTime fmto mfto)
   (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso
      CoordStateInnerFIFOsEnabled))
   (nestCSR′ : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso
      CoordStateOuterFIFOsEnabled))

   (prfnxtim : isNextTimeStep fl tfl tfl′)
   (bsteppre :
      InMemModInst.MDatMapModeReadPred _ tti mfti mftiState)

   (bsteppost :
      OutMemModInst.MDatMapModeWritePred _ tto mfto mftoState)

   (cstate :
      sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap lissolib lisso))
   (cstateNext :
      sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap lissolib lisso))
   :

   BoxesStep.StepPred fl fmti fmto tfl tfl′ tti tti′ tto tto′
   mfti mfto bssso prfnxtim
   (exist _ _ bsteppre)
   (exist _ _ bsteppost)

   →
   (InstBoxesNestPred f fn t linstsig instTypeScopeMap lissolib lisso
nestCSR nestCSR′ TraceBoxesEnab) →
   TraceMemBFEnab f t linstsig instTypeScopeMap lissolib lisso cstateNext
   →
   TraceBoxesEnab f t linstsig instTypeScopeMap lissolib lisso cstate


   with TraceMemBFEnab(f : Freq)(t : TTime f)
   (linstsig : LInstSignature f)
   (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
   (lissolib : LibClos instTypeScopeMap)
   (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
   ∀
   (cstate : sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
   ,
   Type :=
| TraceMemBFStep

   (cstate :
      sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap lissolib lisso))
   (cstateNext :
        sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
           lisso)) :
     TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso cstateNext
     →
     TraceMemBFEnab f t linstsig instTypeScopeMap lissolib lisso cstate


  with TraceFIFOsEnab(f : Freq)(t : TTime f)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
  ∀
     (cstate :
        sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
           lisso))
     ,
     Type
     :=
  | TraceFIFOsStep
     (cstate :
        sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
           lisso))
     (cstateNext :
        sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
           lisso))
     :
        TraceMemFBEnab f t linstsig instTypeScopeMap lissolib lisso cstateNext →
     TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso cstate
     | TraceFIFOsFinal (cstate :
        sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap lissolib
           lisso))
     : OutMemModInst.MDatMapTime _ (‘ (InstSigOutputMems (‘linstsig))) →
        TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso cstate


     with TraceMemFBEnab(f : Freq)(t : TTime f)
     (linstsig : LInstSignature f)
     (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
     (lissolib : LibClos instTypeScopeMap)
     (lisso : Lisso f linstsig instTypeScopeMap lissolib) :
     ∀
     (cstate :
        sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
           lisso))
     ,
     Type :=
     | TraceMemFBStepSkipIO
     (cstate :
        sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
           lisso))
     (cstateNext :
        sig (CoordStateBoxesEnabled f (tNext f t) linstsig instTypeScopeMap
           lissolib lisso))
     :
     TraceBoxesEnab f (tNext _ t) linstsig instTypeScopeMap lissolib lisso
     cstateNext →
     TraceMemFBEnab f t linstsig instTypeScopeMap lissolib lisso cstate.



End SF_Coord.
```

D.5 Coordination language interpreter implementation (main functions)

D.5.1 Super-step

Listing D.17: The FIFO-step wrapper

```
Definition traceFIFOsGenNFP(t : TTime f)
  (currState :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
    (‘ (InstSigOutputMems (‘linstsig))))
  (mti : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
    (‘ (InstSigInputMems (‘linstsig))))
  (traceMemFBGenFunc : ∀ csInnerMemFBEnabled mtiNest,
    TraceMemFBEnab f t linstsig instTypeScopeMap
    lissolib lisso csInnerMemFBEnabled)
  : TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso currState :=
  let (csNext, mtiNest) :=
    FIFOsStepCoordImpl t mti currState mtoNest in
    TraceFIFOsStep f t linstsig instTypeScopeMap lissolib lisso
    currState csNext
    (traceMemFBGenFunc csNext mtiNest).
```

Listing D.18: The FIFO-step worker

```
Definition FIFOsStepCoordImpl(t : TTime f)
  (mti : InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig))))

  (cstate :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
    (‘ (InstSigOutputMems (‘linstsig))))
  :
  ((sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap lissolib
    lisso)) × InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig))))%type.
refine (
  let fix FIFOsStepCoordImplInner(f′ : _)(t′ : TTime f′)(linstsig′ : _)
    (instTypeScopeMap′ : _)(lissolib′ : _)(lisso′ : _)
    (mti′ : InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig′))))
    (cstate′ :
      sig (CoordStateInnerFIFOsEnabled f′ t′ linstsig′ instTypeScopeMap′
        lissolib′ lisso′))(cstate′Acc : Acc CoordStateRecSizeLT (‘ (‘cstate′)))
    { struct cstate′Acc } :
    ((sig (CoordStateInnerMemFBEnabled f′ t′ linstsig′ instTypeScopeMap′
      lissolib′ lisso′)) ×
    InMemModInst.MDatMapTime _ (‘ (InstSigInputMems (‘linstsig)))
    )%type :=


    match cstate′ as cstate′ return _ = cstate′ → _ with
      | exist csstatic cstp ⇒ fun J : cstate′ = exist _ csstatic cstp ⇒
        match csstatic as csstatic return _ = csstatic → _ with
          | exist csr cssp ⇒ fun J0 ⇒
            match csr as csr return _ = csr → _ with
              | CoordStateRaw_make mti″ mto′ csn
                ⇒
                fun J1 ⇒
```

```
match (‘lisso’) with
  LInstSSORaw_make fmi’ fmo’ fmil fmol fmin fmon
  fl fn f’’’
  mfi’ mfo’ mfin mfon boxmap obs manif fifos
  nestlinsts typeinsts instsigdat libinsts
  insttshiftmap libmapsigmap ⇒


  let mtiNestResolved :=

    exist _ (inMemBoxMapKeys (resolveObsInstQual obs)
      (‘ (mti’))) _
    in



    let localUpdatedMti :
      InMemModInst.MDatMapTime fmil mfi’ :=
      let localMtiChanges :=
        FilterInputMapLocal _ _ mtiNestResolved
        in
        mtiPrepend _ _ (‘ (ttimeConv fmil f’ t’ _))
        _ _ (exist (InMemModInst.MDatMapFreqTimePred _
          mfi’)
        mti’’ _) localMtiChanges
        in


        let (newLocMap, newNestMap) :=
          let nestedUpdatedMti :=
            let nestedMtiChanges :=
              FilterInputMapNested _ _ (‘linstsig’)
              mtiNestResolved
              in
              (exist (InMemModInst.MDatMapFreqTimePred _
                mfin) (‘nestedMtiChanges) _)
              in

              processFIFOs f’ t’ fmil fmin
              mfi’ mfin localUpdatedMti nestedUpdatedMti
              linstsig’ instTypeScopeMap’ lissolib’
              lisso’ cstate’
              (VaridMapMod.elements fifos)
              in


              let newNestedCoordStates :
                LInstMapMod.t CoordStateRaw :=
                let csnmok :
                  LInstMapModPred.NoDupType _ csn := _
                  in
                  (LInstMapMod.Build_t _ _ csnmok)
                  in


                  let newCSR := CoordStateRaw_make
                    (InMemBoxWPties.update
                      (‘localUpdatedMti) (‘newLocMap))
                    mto’
                    (LInstMapMod.this
                      newNestedCoordStates)
                    in


                    let csStaticStrong := exist
                      (CoordStateStaticPred f’
                                      linstsig′ instTypeScopeMap′
                                      lissolib′ lisso′) newCSR _
                                    in
                                    let newUpdatedNestData :=
                                      exist _
                                      (InMemBoxWPties.update (‘mti)
                                        (‘newNestMap)) _
                                      in
                                      (exist
                                        (CoordStateInnerMemFBEnabled
                                          f′ t′ linstsig′
                                          instTypeScopeMap′ lissolib′
                                          lisso′) csStaticStrong _,
                                        newUpdatedNestData
                                      )


                  end
              end eq_refl
          end eq_refl
      end eq_refl


      in
      FIFOsStepCoordImplInner f t linstsig instTypeScopeMap lissolib
      lisso mti cstate _
  ).
Defined.
```

Listing D.19: The FIFO-box memory step wrapper

```
Definition traceMemFBGenNFP(t : TTime f)
  (currState :
    sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
    (‘ (InstSigInputMems (‘linstsig))))
  (traceBoxesGenFunc : ∀
    csBoxesEnabled mtiNest, TraceBoxesEnab f (tNext t)
    linstsig instTypeScopeMap lissolib lisso csBoxesEnabled) :
  TraceMemFBEnab f t linstsig instTypeScopeMap lissolib lisso currState :=
  let csNext := MemFBStepOuterCoordImpl t currState
    in
    TraceMemFBStepSkipIO f t linstsig instTypeScopeMap lissolib lisso
    currState csNext (traceBoxesGenFunc csNext mtiNest).
```

Listing D.20: The FIFO-box memory step worker function

```
    Definition MemFBStepOuterCoordImpl
      (cstate :
        sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap
          lissolib lisso)) :
      sig (CoordStateBoxesEnabled f (tNext _ t) linstsig instTypeScopeMap
        lissolib lisso).
    refine (

      let fix MemFBStepOuterCoordImplInner(f′ : _)(t′ : TTime f′)(linstsig′ : _)
        (instTypeScopeMap′ : _)(lissolib′ : _)(lisso′ : _)
        (cstate′ :
```

```
  sig (CoordStateInnerMemFBEnabled f′ t′ linstsig′ instTypeScopeMap′
     lissolib′ lisso′))(cstate′Acc : Acc CoordStateRecSizeLT (‘ (‘cstate′)))
{ struct cstate′Acc } :
sig (CoordStateBoxesEnabled f′ (tNext _ t′) linstsig′ instTypeScopeMap′
   lissolib′ lisso′) :=
match cstate as cstate return _ = cstate → _ with
   | exist csstatic cstp ⇒ fun J : cstate = exist _ csstatic cstp ⇒
      match csstatic as csstatic return _ = csstatic → _ with
         | exist csr cssp ⇒ fun J0 ⇒
            match csr as csr return _ = csr → _ with
               | CoordStateRaw_make
                  mti mto csn ⇒
                  fun J1 ⇒



                     let mtiMemFreqMapActiveSubset :=
                        match (‘lisso′) with
                           LInstSSORaw_make
                           fmi′ fmo′ fmil fmol fmin fmon fl fn f′′′
                           mfi′ mfo′ mfin mfon boxmap obs manif fifos
                           nestlinsts typeinsts instsigdat libinsts insttshiftmap
                           libmapsigmap ⇒
                           InMemBoxWPties.filter
                           (fun _ e ⇒ MemFBEnabledBool e t) (‘mfi′)
                        end
                        in
                        let mtoMemFreqMapActiveSubset :=
                           match (‘lisso′) with
                              LInstSSORaw_make fmi′ fmo′ fmil fmol fmin fmon
                              fl fn f′′′
                              mfi′ mfo′ mfin mfon boxmap obs manif fifos
                              nestlinsts typeinsts instsigdat libinsts insttshiftmap
                              libmapsigmap ⇒
                              OutMemBoxWPties.filter
                              (fun _ e ⇒ MemBFEnabledBool e t) (‘mfo′)
                           end
                           in



                           let mtiMemTimeMapActiveSubset := InMemBoxWPties.filter
                              (fun k _ ⇒ InMemModBox.otm.mem k mtiMemFreqMapActiveSubset)
                              (mti)
                              in
                              let mtoMemTimeMapActiveSubset := OutMemBoxWPties.filter
                                 (fun k _ ⇒ OutMemModBox.otm.mem k mtoMemFreqMapActiveSubset)
                                 (mto)
                                 in



                                 let mtiMemsExecuted :=
                                    InMemModBox.otm.map executeMem mtiMemTimeMapActiveSubset
                                    in
                                    let mtoMemsExecuted :=
                                       OutMemModBox.otm.map executeMem mtoMemTimeMapActiveSubset
                                       in



                                       let mtiRawNew :=
                                          InMemBoxWPties.update (mti) mtiMemsExecuted
                                          in
                                          let mtoRawNew :=
                                             OutMemBoxWPties.update (mto) mtoMemsExecuted
                                             in
                                             let newCoordStateRaw :=
                                               CoordStateRaw_make mtiRawNew mtoRawNew csn
                                               in
                                               let newCoordStateRawStrongStatic :
                                                 sig (CoordStateStaticPred f′ linstsig′
                                                   instTypeScopeMap′ lissolib′ lisso′) :=
                                                 exist _ newCoordStateRaw
                                                 (depCoordStateStaticPredProcessNestedCSMapNOT_ENOUGH_ARGS _ _
_ _ _ _)

                                                 in
                                                 exist (CoordStateBoxesEnabled f′
                                                   (tNext _ t′)
                                                   linstsig′ instTypeScopeMap′ lissolib′
                                                   lisso′) newCoordStateRawStrongStatic
                                                 (depCoordStateOuterFIFOsEnabledProcessNestedCSMapNOT_ENOUGH_ARGS
_ _ _ _ _ _ _)

                          end eq_refl
                       end eq_refl
                    end eq_refl
                    in MemFBStepOuterCoordImplInner f t linstsig instTypeScopeMap lissolib
                    lisso cstate _
                 ).
      Defined.
```

Listing D.21: The box step invocation function

```
Definition traceBoxesGenNFP(t : TTime f)
  (currState :
    sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
    (‘ (InstSigInputMems (‘linstsig))))
  (traceNestBoxesGenFunc : ∀ fn
    (csFIFOsEnabled :
      sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
        lisso CoordStateInnerFIFOsEnabled))
    (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
      (‘ (InstSigInputMems (‘linstsig)))),
    (sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
      lisso CoordStateOuterFIFOsEnabled) ×
    sig (mtoExternalNestPred (InstSigFreqMemOut (‘linstsig))
      ((InstSigOutputMems (‘linstsig))))
    )%type)
  (traceMemBFGenFunc : ∀
    csMemBFEnabled
    (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
      (‘ (InstSigOutputMems (‘linstsig)))),
    TraceMemBFEnab f t linstsig instTypeScopeMap lissolib
    lisso csMemBFEnabled
  )
  :
  (TraceBoxesEnab f t linstsig instTypeScopeMap lissolib lisso currState
  )%type.
    [Function body omitted]
refine(
  match lisso as lisso′ return
    (TraceBoxesEnab f t linstsig instTypeScopeMap lissolib
      lisso currState)%type with
    | exist lissoraw lissopred ⇒
```

```
match lissoraw with
  | LInstSSORaw_make fmi fmo fmil fmol fmin fmon
      fl fn f' mfi mfo mfin mfon
      hboxmap obs manif fifos nestedlinsts libinsts linstsigmap
      libmap insttshiftmap libmapsigs ⇒

      let fLocalDivPrf : FreqDivide fl f := _
        in let instsigInDivPrf :
          FreqDivide (InstSigFreqMemIn (`linstsig)) f := _
          in let fmiDivPrf : FreqDivide fmil f := _
            in let fmoDivPrf : FreqDivide fmol f := _
              in let prfnxtim : isNextTimeStep fl (` (ttimeConv fl f t fLocalDivPrf))
                (tNext (` (ttimeConv fl f t fLocalDivPrf))) :=
                tNextIsNextTimeStep _ _


                in let boxesStepPred : BoxesStep.StepSSOPred
                  fl fmil fmol mfi mfo hboxmap := _


                  in let lissoLeibLoc : FreqMemsInLissoLeib
                    fl fmil fmol mfi mfo
                    (` lisso) := _ in
                    let lissoLeibNest :
                      FreqNestInLissoLeib fn
                      (` lisso) := _ in


                      let nb := (traceNestBoxesGenFunc f
                        (currStateNestBoxes t currState f)) mtiNest
                      in


                      let tfl := (` (ttimeConv fl f t fLocalDivPrf)) in
                        let tfl' := (tNext (` (ttimeConv fl f t fLocalDivPrf))) in
                          let tti := (` (ttimeConv fmil f t fmiDivPrf)) in
                            let tti' := (` (ttimeConv fmil f (tNext t) fmiDivPrf)) in
                              let tto := (` (ttimeConv fmol f t fmoDivPrf)) in
                                let tto' := (` (ttimeConv fmol f (tNext t) fmoDivPrf)) in


                                  let mti :=
                                    (mtiCurrStateMti
                                      fl fmil fmol t mfi mfo currState lissoLeibLoc)
                                    in let mto :=
                                      (mtiCurrStateMto
                                        fl fmil fmol t mfi mfo currState lissoLeibLoc) in
                                      let nestCSR := (currStateNestBoxes t currState fn)


                                        in let outp :=
                                          BoxesStepImpl.StepFunc fl fmil fmol
                                          tfl tfl' tti tti' tto tto'
                                          mfi mfo (exist _ _ boxesStepPred)
                                          (tNextIsNextTimeStep _ _) (exist _ _
                                            (boxesPre fmil tti mfi mti))
                                          in


                                          let outpPrepended :=
                                            mtoPrepend fmol fmol (` (ttimeConv fmol f t _))
                                            mfo mfo mto (` (`outp))
                                            in


                                            let nestCSR' := (exist _ (` (fst nb)) _) in
```

```
                                                  let cstateNext := (buildCurrStateMemBF fl fn fmil
                                                    fmol t tfl' tti' tto' mfi mfo mti outpPrepended
                                                    nestCSR') in


                                                  let traceBF :=
                                                    let mtoResolved :=
                                                      exist _ (outMemBoxMapKeys _ (resolveManifInstQualRev manif)

                                                        (' (' (snd nb)))) _
                                                      in
                                                      traceMemBFGenFunc cstateNext (' (snd nb))
                                                      in let bsteppre := _ in let bsteppost := _ in

                                                        TraceBoxesStep
                                                        f t linstsig instTypeScopeMap lissolib lisso fl fn
                                                        fmil fmol tfl tfl' t
                                                        (' (ttimeConv (InstSigFreqMemIn ('linstsig)) f t
                                                          instsigInDivPrf)) tti tti' tto tto' mfi mfo
                                                        (exist _ _ boxesStepPred) mti mto mti outpPrepended
                                                        nestCSR nestCSR' prfnxtim bsteppre bsteppost currState cstateNext
_ _ traceBF

      end
  end
).
Defined.
```

Listing D.22: The box step worker function

```
Module BoxesStepImpl .
  Section StepOneBoxS.
    Variables f fmi fmo : Freq.
    Variable t : TTime f.
    Variable ti : TTime fmi.
    Variable to : TTime fmo.
    Variable mfi : InMemModInst.MDatFreqMap fmi.
    Variable mfo : OutMemModInst.MDatFreqMap fmo.
    Variable stepSSO : BoxesStep.StepSSO f fmi fmo mfi mfo.
    Variable instate : sig (InMemModInst.MDatMapModeReadPred _ ti mfi).
    Variable hbx : boxidPred.PredidDecidable.t ×
      (HBoxSSONonDep × (InMemModBox.otm.t ipm.Varid)
        × (ipm.VaridMapMod.t opm.HBCL_OidMemBF)).
    Hypothesis inprf : SetoidList.InA (@BoxTypeIdMapMod.eq_key_elt _) hbx
      (BoxTypeIdMapMod.elements ('stepSSO)).
    Definition stepOneBox : OutMemModBox.MDatTimeMapRaw.
    refine (

      match (snd hbx) as hbxt return _ = hbxt → _ with
        | (Build_HBoxSSONonDep hbf hbfi hbfo hbinmem hboutmem hbsso,
          inmemmap, outmemmap) ⇒
        fun J ⇒

          match hbsso as hbsso return _ = hbsso → _ with
            | exist hbraw hbpred ⇒ fun J1 ⇒


              let thb := ' (ttimeConv hbf _ t _)
                in
                let thfi := ' (ttimeConv hbfi _ ti _)
                  in
```

```
                    let thfo := ‘ (ttimeConv hbfo _ to _)
                      in


                      let inp :
                        InMemModBox.MDatBoxTime MDataInst.ReadEnabled
                        (InMemModBox.otm.map MDataTypeInst.MDFE_boxMemDat (‘mfi)) hbfi thfi :=
                        exist _ (‘ ( (‘instate))) _
                        in


                        let hboxresult := HBoxStep hbf _ _ thb thfi
                          thfo

                          hbinmem hboutmem inmemmap outmemmap
                          (InMemModBox.otm.map (MDataTypeInst.MDFE_boxMemDat) (‘mfi))
                          (OutMemModBox.otm.map (MDataTypeInst.MDFE_boxMemDat) (‘mfo))
                          hbsso inp
                          in (‘ (‘ hboxresult))

                end eq_refl

            end eq_refl
        ).
    Defined.

End StepOneBoxS.

Definition StepFunc(f fmi fmo : Freq)
    (t t′ : TTime f  CoordPhase )
    (ti ti′ : TTime fmi)(to to′ : TTime fmo)
    (mfi : InMemModInst.MDatFreqMap fmi)(mfo : OutMemModInst.MDatFreqMap fmo)
    (stepSSO : BoxesStep.StepSSO f fmi fmo mfi mfo)
    (nexttimeprf : isNextTimeStep f t t′)
    (instate : sig (InMemModInst.MDatMapModeReadPred _ ti mfi) ) :
    sig (BoxesStep.StepPred
      f fmi fmo t t′ ti ti′ to to′ mfi mfo stepSSO nexttimeprf instate).
refine (

    let fix processBoxesMapAsList
      (dl : list (ipm.boxidPred.PredidDecidable.t ×
        (HBoxSSONonDep × InMemModBox.otm.t Varid × VaridMapMod.t HBCL_OidMemBF)))
      (inclprf : SetoidList.inclA (@BoxTypeIdMapMod.eq_key_elt _ ) dl
        (BoxTypeIdMapMod.elements
          ( (‘ (stepSSO))))
      ) { struct dl }
      : OutMemModBox.MDatTimeMapRaw :=
      match dl as dl return _ = dl → _ with
        | nil ⇒ fun _ ⇒ OutMemModBox.otm.empty _
        | (hbx :: dl′)%list ⇒ fun J : dl = (hbx :: dl′)%list ⇒
          let
            accumMap : OutMemModBox.MDatTimeMapRaw :=
            processBoxesMapAsList dl′ _
            in
            OutMemBoxWPties.update
            (accumMap)
            (stepOneBox _ _ _ t ti to mfi mfo instate hbx)
      end eq_refl
      in


      let outputSubsetRaw : OutMemModInst.MDatMapTime fmo mfo := exist _
        (processBoxesMapAsList (BoxTypeIdMapMod.elements
          (
            (‘ (stepSSO))))

        (inclReflBoxid _ _)) _
      in


      let outputSubset := exist _ outputSubsetRaw _ in
        exist (BoxesStep.StepPred
          f fmi fmo t t′ ti ti′ to to′ mfi mfo stepSSO nexttimeprf instate)
        outputSubset _

).
Defined.
End BoxesStepImpl.
```

Listing D.23: The box-FIFO memory step wrapper

```
Definition traceMemBFGenNFP(t : TTime f)
  (currState :
    sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
    (‘ (InstSigOutputMems (‘linstsig))))
  (traceFIFOsGenFunc : ∀ csInnerFIFOsEnabled
    (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (‘linstsig))
      (‘ (InstSigOutputMems (‘linstsig)))),
    TraceFIFOsEnab f t linstsig instTypeScopeMap
    lissolib lisso
    csInnerFIFOsEnabled
  )
  : TraceMemBFEnab f t linstsig instTypeScopeMap lissolib lisso currState.
refine (
  let csNext := MemBFStepCoordImpl t currState
    in
    let mtoNest′ :=
      let mtoNest′Raw := OutMemBoxWPties.update (‘mtoNest)
        (‘ (coordStateMtoExternalFromInner _ _ _ _ _ _ csNext))
        in exist _ mtoNest′Raw _ in
        TraceMemBFStep f t linstsig instTypeScopeMap lissolib lisso
        currState csNext
        (traceFIFOsGenFunc csNext mtoNest′
        )
).
Defined.
```

Listing D.24: The box-FIFO memory step worker function

```
    Definition MemBFStepCoordImpl
      (cstate : sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap
        lissolib lisso)) :
      sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap
        lissolib lisso).
    refine (
      let fix MemBFStepCoordImplInner(f′ : _)(t′ : TTime f′)(linstsig′ : _)
        (instTypeScopeMap′ : _)(lissolib′ : _)(lisso′ : _)
        (cstate′ :
          sig (CoordStateMemBFEnabled f′ t′ linstsig′ instTypeScopeMap′
            lissolib′ lisso′))(cstate′Acc : Acc CoordStateRecSizeLT (‘ (‘cstate′)))
        { struct cstate′Acc } :
        sig (CoordStateInnerFIFOsEnabled f′ t′ linstsig′ instTypeScopeMap′
          lissolib′ lisso′)
        :=
    match cstate as cstate return _ = cstate → _ with
      | exist csstatic cstp ⇒ fun J : cstate = exist _ csstatic cstp ⇒
        match csstatic as csstatic return _ = csstatic → _ with
          | exist csr cssp ⇒ fun J0 ⇒
            match csr as csr return _ = csr → _ with
              | CoordStateRaw_make
                mti mto csn ⇒
                fun J1 ⇒
                  let mtiMemFreqMapActiveSubset :=
                    match ('lisso') with
                      LInstSSORaw_make fmi' fmo' fmil fmol fmin fmon
                      fl fn f''
                      mfi' mfo' mfin mfon boxmap obs manif fifos
                      nestlinsts typeinsts instsigdat libinsts insttshiftmap
                      libmapsigmap ⇒
                      InMemBoxWPties.filter
                      (fun _ e ⇒ MemFBEnabledBool e t) ('mfi')
                    end
                    in
                    let mtiMemTimeMapActiveSubset := InMemBoxWPties.filter
                      (fun k _ ⇒ InMemModBox.otm.mem k mtiMemFreqMapActiveSubset)
                      (mti)
                      in
                      let mtoMemFreqMapActiveSubset :=
                        match ('lisso') with
                          LInstSSORaw_make fmi' fmo' fmil fmol fmin fmon
                          fl fn f''
                          mfi' mfo' mfin mfon boxmap obs manif fifos
                          nestlinsts typeinsts instsigdat libinsts insttshiftmap
                          libmapsigmap ⇒
                          OutMemBoxWPties.filter
                          (fun _ e ⇒ MemBFEnabledBool e t) ('mfo')
                        end
                        in
                        let mtoMemTimeMapActiveSubset := OutMemBoxWPties.filter
                          (fun k _ ⇒ OutMemModBox.otm.mem k mtoMemFreqMapActiveSubset)
                          (mto)

                          in let mtiMemsExecuted :=
                            InMemModBox.otm.map executeMem mtiMemTimeMapActiveSubset
                            in let mtoMemsExecuted :=
                              OutMemModBox.otm.map executeMem mtoMemTimeMapActiveSubset

                              in let mtiRawNew :=
                                InMemBoxWPties.update (mti) mtiMemsExecuted
                                in let mtoRawNew :=
                                  OutMemBoxWPties.update (mto) mtoMemsExecuted
                                  in

                                  let newCoordStateRaw :=
                                    CoordStateRaw_make mtiRawNew mtoRawNew

                          csn
  in
  let newCoordStateRawStrongStatic :
    sig (CoordStateStaticPred f' linstsig'
      instTypeScopeMap' lissolib' lisso') :=
    exist _ newCoordStateRaw
    (depCoordStateStaticPredProcessNestedCSMapNOT_ENOUGH_ARGS _ _ _ _ _ _)
    in exist (CoordStateInnerFIFOsEnabled f' t'
      linstsig' instTypeScopeMap' lissolib'
      lisso') newCoordStateRawStrongStatic
    (depCoordStateInnerFIFOsEnabledProcessNestedCSMapNOT_ENOUGH_ARGS _ _ _ _ _ _ _)
            end eq_refl
        end eq_refl
    end eq_refl
    in MemBFStepCoordImplInner f t linstsig instTypeScopeMap lissolib lisso cstate
```

```
      _
  ).
  Defined.
```

Listing D.25: The nested box step

```
Definition traceNestBoxesGenNFP(fn : Freq)(t : TTime f)
  (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
    lisso CoordStateInnerFIFOsEnabled))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (‘linstsig))
    (‘ (InstSigInputMems (‘linstsig)))) :
  (sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
    lisso CoordStateOuterFIFOsEnabled) ×
  sig (mtoExternalNestPred (InstSigFreqMemOut (‘linstsig))
    ((InstSigOutputMems (‘linstsig))))
  )%type.
refine(

  let fix processNestedInst
    (l dl : list (HBCL_OidLInst × (CSTempCorrectND × TTFL)))
    (inclprf : SetoidList.inclA (@LInstMapMod.eq_key_elt _ ) dl l)
    (inprf : boxesEnabNestPred t l)
    (mapsofar : LInstMapMod.t CSTempCorrectND)
    (outdatsofar : OutMemModBox.MDatTimeMapRaw)
    : ((LInstMapMod.t CSTempCorrectND) × OutMemModBox.MDatTimeMapRaw)%type :=
    match dl as dl return _ = dl → _ with
      | nil ⇒ fun _ ⇒ (mapsofar, outdatsofar)
      | cons (v, (cs, ttfl)) m′ ⇒ fun J : dl = cons (v, (cs, ttfl)) m′ ⇒


        let mnew := processNestedInst
              l m′ (inclNest _ _ _ _ _ _ inclprf J)
inprf mapsofar outdatsofar in


let csnew :=
  match TTseqb _ _ (ttimeTTFLAdjust f (tPrev _ t) ttfl)
    (CSTempCorrectND_t cs)
    as ttseqb return
      _ = ttseqb → _ with


    | true ⇒ fun J0 : TTseqb _ _
      (ttimeTTFLAdjust f (tPrev _ t) ttfl)
      (CSTempCorrectND_t cs) = true ⇒
      let csbp := (inprf (v, (cs, ttfl)) _ _)
        in


        let mtiNest″ :=
          FilterInputMapNestedInst v _ _
          (‘ (CSTempCorrectND_linstsig cs)) mtiNest
          in


          let fTrace :=
            (traceFIFOsGenFunc
              (CSTempCorrectND_f cs)
              (CSTempCorrectND_linstsig cs)
              (CSTempCorrectND_instTypeScopeMap cs)
              (CSTempCorrectND_lissolib cs)
              (CSTempCorrectND_lisso cs)
              ( (CSTempCorrectND_t cs))
            (exist (CoordStateInnerFIFOsEnabled (CSTempCorrectND_f cs)
              ( (CSTempCorrectND_t cs))
              (CSTempCorrectND_linstsig cs)
              (CSTempCorrectND_instTypeScopeMap cs)
              (CSTempCorrectND_lissolib cs)
              (CSTempCorrectND_lisso cs))
            (CSTempCorrectND_CS cs) csbp)
          )
          mtiNest''
          (exist _ (OutMemModBox.otm.empty _) _) in


          let newcs := (processSubInstFinite
            cs (inprf (v, (cs, ttfl))
              (inprfNest _ _ _ _ _ _ inclprf J)
              (ttseqCorr v cs ttfl t J0 ))
            fTrace) in



          ({| CSTempCorrectND_f := CSTempCorrectND_f cs;
            CSTempCorrectND_t :=
            (starvationLongstop cs);
            CSTempCorrectND_linstsig :=
            CSTempCorrectND_linstsig cs;
            CSTempCorrectND_instTypeScopeMap :=
            CSTempCorrectND_instTypeScopeMap cs;
            CSTempCorrectND_lisso := CSTempCorrectND_lisso cs;
            CSTempCorrectND_CS := (` (fst newcs))
          |}, (snd newcs))


  | false ⇒ fun _ ⇒
    (cs, exist _ (OutMemModBox.otm.empty _) _)
end eq_refl


in let vLiblessStrong := exist liblessInst v _
  in let extKeysLifted := outMemBoxMapKeys
    (concatInstMemBF vLiblessStrong) (` (snd csnew))
    in


    ((LInstMapMod.add v (fst csnew) (fst mnew)),
      (OutMemBoxWPties.update (snd mnew) extKeysLifted)
    )
  end eq_refl


  in let processRaw :=
    let linstttflmap :=
      match (`lisso) with
        LInstSSORaw_make _ _ _ _ _ _ _ _ _ _ _ _ _
        _ _ _ _ _ _ _ _ ttflinstmap _ ⇒ ttflinstmap
      end in
      let linstttflmapOK := _ in
        processNestedInst
        (LInstMapMod.elements (combineLInstMapModMaps (`nestCSR) linstttflmap
          linstttflmapOK))
        (LInstMapMod.elements (combineLInstMapModMaps (`nestCSR) linstttflmap
          linstttflmapOK))
        (inclRefl _ _)
        (coordStateMapBoxesNest t linstttflmap fn (`nestCSR) (``nestCSR) _)
        (LInstMapMod.empty _)
        (OutMemModBox.otm.empty _)
        in
```

```
                let processPred : InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib lisso
                  CoordStateOuterFIFOsEnabled (fst processRaw) := _ in


                  let mtoExtTimeStrong := exist _ (snd processRaw) _ in
                    let mtoExtStrong := exist (mtoExternalNestPred (InstSigFreqMemOut ('linstsig))
                      (InstSigOutputMems ('linstsig))) mtoExtTimeStrong _ in
                    (exist _ (fst processRaw) processPred, mtoExtStrong)

).
Defined.
```

Listing D.26: The coordination language cofixpoint super-step

```
CoFixpoint traceBoxesGen(f : Freq)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib)
  (t : TTime f)
  (ti : TTime (InstSigFreqMemIn ('linstsig)))
  (ttieqi : TTseq t ti)
  (is : InputStream (InstSigFreqMemIn ('linstsig))
    ( (InstSigInputMems ('linstsig))) ti)
  (tteq : nextCeil (proj1 (''linstsig)) t ti)(currState :
    sig (CoordStateBoxesEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn ('linstsig))
    (' (InstSigInputMems ('linstsig))))
  : TraceBoxesEnab f t linstsig instTypeScopeMap lissolib lisso currState :=
  traceBoxesGenNFP f linstsig instTypeScopeMap lissolib lisso t currState mtiNest (fun fn nestCSR mtiNest' ⇒
          traceNestBoxesGenFix f linstsig instTypeScopeMap lissolib lisso t fn
          nestCSR mtiNest'
          (WFPNestCSRWF _ _ _ _ _ _ nestCSR) )
      (traceMemBFGen f linstsig instTypeScopeMap lissolib lisso t ti ttieqi
        is tteq)

with traceMemBFGen(f : Freq)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib)
  (t : TTime f)(ti : TTime (InstSigFreqMemIn ('linstsig)))
  (ttieqi : TTseq t ti)
  (is : InputStream (InstSigFreqMemIn ('linstsig))
    ( (InstSigInputMems ('linstsig))) ti)
  (tteq : nextCeil (proj1 (''linstsig)) t ti)
  (currState :
    sig (CoordStateMemBFEnabled f t linstsig instTypeScopeMap lissolib
      lisso))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut ('linstsig))
    (' (InstSigOutputMems ('linstsig))))
  : TraceMemBFEnab f t linstsig instTypeScopeMap lissolib lisso currState :=
  traceMemBFGenNFP f linstsig instTypeScopeMap lissolib lisso t currState mtoNest
      (traceFIFOsGen f linstsig instTypeScopeMap lissolib lisso t ti ttieqi
        is tteq)

with traceFIFOsGen(f : Freq)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib)
  (t : TTime f)(ti : TTime (InstSigFreqMemIn (`linstsig)))
  (ttieqi : TTseq t ti)
  (is : InputStream (InstSigFreqMemIn (`linstsig))
    ( (InstSigInputMems (`linstsig))) ti)
  (tteq : nextCeil (proj1 (``linstsig)) t ti)
  (currState :
    sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut (`linstsig))
    (` (InstSigOutputMems (`linstsig))))
  : TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso currState :=
  match is with
    | InputStreamFinal ⇒ TraceFIFOsFinal f t linstsig instTypeScopeMap
      lissolib lisso currState mtoNest
    | InputStreamInd tinext tinxtprf mtii isnext ⇒
      traceFIFOsGenNFP f linstsig instTypeScopeMap lissolib lisso
      t currState mtoNest (`mtii)
      (traceMemFBGen f linstsig instTypeScopeMap lissolib lisso t
        tinext (TTSeqNextNOT_ENOUGH_ARGS _ _ _ _)
        isnext (nextCeil3NOT_ENOUGH_ARGS _ _ _ _ _))
  end

with traceMemFBGen(f : Freq)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib)
  (t : TTime f)(ti : TTime (InstSigFreqMemIn (`linstsig)))
  (ttieqi : TTseq (tNext _ t) ti)
  (is : InputStream (InstSigFreqMemIn (`linstsig))
    ( (InstSigInputMems (`linstsig))) ti)
  (tteq : nextCeil (proj1 (``linstsig)) (tNext _ t) ti)
  (currState :
    sig (CoordStateInnerMemFBEnabled f t linstsig instTypeScopeMap
      lissolib lisso))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (`linstsig))
    (` (InstSigInputMems (`linstsig))))
  : TraceMemFBEnab f t linstsig instTypeScopeMap lissolib lisso currState :=
  (traceMemFBGenNFP f linstsig instTypeScopeMap lissolib lisso
    t currState mtiNest
    (traceBoxesGen f linstsig instTypeScopeMap lissolib lisso (tNext _ t)
      ti ttieqi is
      tteq)).
```

Listing D.27: The coordination language nesting fixpoint super-step

```
Fixpoint traceNestBoxesGenFix(f : Freq)
  (linstsig : LInstSignature f)
  (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
  (lissolib : LibClos instTypeScopeMap)
  (lisso : Lisso f linstsig instTypeScopeMap lissolib)
  (t : TTime f)(fn : Freq)
  (nestCSR : sig (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib
    lisso CoordStateInnerFIFOsEnabled))
  (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn (`linstsig))
    (` (InstSigInputMems (`linstsig))))
  (nestCSRAcc : Acc CoordStateNestMapSizeExistsLT
    (LInstMapMod.map (fun e ⇒ (` (CSTempCorrectND_CS e))) (`nestCSR)))

  { struct nestCSRAcc } :
  (sig
    (InstMapMatchSSO f fn linstsig instTypeScopeMap lissolib lisso
      CoordStateOuterFIFOsEnabled) ×
    sig
    (mtoExternalNestPred (InstSigFreqMemOut ('linstsig))
      (InstSigOutputMems ('linstsig))))%type :=
  let fix
    traceFIFOsGenFix(f : Freq)
    (linstsig : LInstSignature f)
    (instTypeScopeMap : LInstMapMod.t LInstSignatureRaw)
    (lissolib : LibClos instTypeScopeMap)
    (lisso : Lisso f linstsig instTypeScopeMap lissolib)
    (t tbase : TTime f)
    (currState :
      sig (CoordStateInnerFIFOsEnabled f t linstsig instTypeScopeMap
        lissolib lisso))
    (mti : InMemModInst.MDatMapTime (InstSigFreqMemIn ('linstsig))
      (' (InstSigInputMems ('linstsig))))
    (mtoNest : OutMemModInst.MDatMapTime (InstSigFreqMemOut ('linstsig))
      (' (InstSigOutputMems ('linstsig))))
    (wfp : AccTraceFix f t linstsig)
    { struct wfp }
    : TraceFIFOsEnab f t linstsig instTypeScopeMap lissolib lisso currState :=

    let traceMemBFGenFix(t' : TTime f)
      (currState :
        sig (CoordStateMemBFEnabled f t' linstsig instTypeScopeMap lissolib
          lisso))
      (mtoNest' : OutMemModInst.MDatMapTime (InstSigFreqMemOut ('linstsig))
        (' (InstSigOutputMems ('linstsig))))
      (wfp : AccTraceFix f t' linstsig)
      : TraceMemBFEnab f t' linstsig instTypeScopeMap lissolib lisso currState :=

      let mti : InMemModInst.MDatMapTime (InstSigFreqMemIn ('linstsig))
        (' (InstSigInputMems ('linstsig))) :=
        (exist _ (InMemModBox.otm.empty _)
          (MDatTimeStrongNOT_ENOUGH_ARGS _ _ _)) in
        traceMemBFGenNFP f linstsig instTypeScopeMap lissolib lisso
        t' currState mtoNest'
        (fun currState mtoNest'' ⇒
          traceFIFOsGenFix f linstsig instTypeScopeMap lissolib lisso
          t' tbase currState mti mtoNest'' wfp)

        in let
          traceBoxesGenFix(t' : TTime f)
          (currState :
            sig (CoordStateBoxesEnabled f t' linstsig instTypeScopeMap lissolib
              lisso))
          (mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn ('linstsig))
            (' (InstSigInputMems ('linstsig))))
          (wfp : AccTraceFix f t' linstsig)
          : TraceBoxesEnab f t' linstsig instTypeScopeMap lissolib lisso
          currState :=
traceBoxesGenNFP f linstsig instTypeScopeMap lissolib lisso
t' currState mtiNest (fun fn nestCSR mtiNest' ⇒
  traceNestBoxesGenFix
  f linstsig instTypeScopeMap lissolib lisso t' fn nestCSR mtiNest'
  (WFPNestCSRAccInvNOT_ENOUGH_ARGS _ _ _ _ _ _ _ _ _ _ _ _ _ _
    nestCSRAcc)


)
(fun currState' mtoNest' ⇒ traceMemBFGenFix t' currState'
  mtoNest' wfp)

in let traceMemFBGenFix(t' : TTime f)(currState :
  sig (CoordStateInnerMemFBEnabled f t' linstsig instTypeScopeMap
    lissolib lisso))
(mtiNest : InMemModInst.MDatMapTime (InstSigFreqMemIn ('linstsig))
  (‘ (InstSigInputMems (‘linstsig))))
(wfp : AccTraceFix f t′ linstsig)
: TraceMemFBEnab f t′ linstsig instTypeScopeMap lissolib lisso
currState :=
traceMemFBGenNFP f linstsig instTypeScopeMap lissolib lisso
t′ currState mtiNest
(fun currState mtiNest ⇒
  traceBoxesGenFix (tNext _ t′) currState mtiNest
  (WFPNOT_ENOUGH_ARGS _ _ _ _ _ _ wfp) )

in
if TTleb _ _ (tNext _ tbase) t
  then
    TraceFIFOsFinal f t linstsig instTypeScopeMap lissolib lisso
    currState mtoNest
  else
    traceFIFOsGenNFP f linstsig instTypeScopeMap lissolib lisso
    t currState mtoNest mti
    (fun currState mtiNest ⇒
      traceMemFBGenFix t currState mtiNest wfp)
    in

    traceNestBoxesGenNFP f linstsig instTypeScopeMap lissolib lisso
    (fun f′ linstsig′ instTypeScopeMap′ lissolib′ lisso′ t′
      currState′ mti′ mtoNest′ ⇒
      traceFIFOsGenFix f′ linstsig′ instTypeScopeMap′ lissolib′ lisso′
      t′ t′ currState′ mti′ mtoNest′
      (WFPNestNOT_ENOUGH_ARGS f f′ t t′ linstsig linstsig′
        (FreqDivNOT_ENOUGH_ARGS _ _) (AccTraceFixWF _ _ _))
    )
    fn t nestCSR mtiNest.
```

D.6 Expression language

Listing D.28: The expression language

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Wellfounded.Inverse_Image.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.ExprSSO.

Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.redFunc.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.RedAppBranch.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.RedPattBranch.

Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UTypeSysOid_SB.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UBoxEmptyEnc_SB.
Import HBCL_0_1_L_UBoxEmtpy.
Import HBCL_0_1_L_UTS.
Local Open Scope program_scope.

Section exprLangS.
  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).
  Context '{ICostDT : CostDT CTDT CTDTP}.
  Context '{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context '{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.
  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.
  Hypothesis ICostDRecTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDRecT)) = LRTypesPSEqSigT.
  Variable minC : ∀ t : ProtoT, sig (CTDTP t).
  Hypothesis minCMin : ∀ t u, CT_PD_interp(CostBase := ICostDT)
    t (' (minC t)) u (" (minC t)) = 0.
  Definition Encoding := Empty_set.
  Definition AST : Set := unit.
  Definition parse : Encoding → AST.
  Defined.

  Definition OidTypeIOMatch(map : InpOutpTypes _ CTDTP)(t : ProtoT) :=
  ∃ t', t =t= t' ∧
    ∃ mc : HBCL_0_1_Id_S.VaridMapMod.t (sigT LTypeRaw),
      mc = HBCL_0_1_Id_S.VaridMapMod.map (fun t ⇒
        existT (projT1 (sigTypes.projT1sigT2 (fst (fst t))))
        (' (projT2 (sigTypes.projT1sigT2 (fst (fst t)))))) map ∧
      ∃ lp : LTypeP (projT1 t')
        (LRecordType (projT1 t') (HBCL_0_1_Id_S.VaridMapMod.this mc)),
        (projT2 t' = exist _ _ lp).

  Inductive sso (uboxInp uboxOutp : InpOutpTypes _ CTDTP) : Type :=
  | Make_sso(t t' : ProtoT)(c : sig (CTDTP t))(c' : sig (CTDTP t'))
    (v : HBCL_0_1_Id_S.Varid)(r : Rsso ICostDT)
    (wc : sigWssoClos CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC
      minC r) :
    funcExtractionPred CTDT CTDTP r t t' c c' v →
    OidTypeIOMatch uboxInp t' →
    OidTypeIOMatch uboxOutp t →
    sso uboxInp uboxOutp.
  Definition compile(itypes otypes : InpOutpTypes _ CTDTP) :
    option (sso itypes otypes) := None.

  Section RedFitS.
    Variables itypes otypes : InpOutpTypes _ CTDTP.
    Variable ssoTypeInst : sso itypes otypes.
    Variable inDat : sig (UDataPSTMatchesInpOutpTypes _ CTDTP itypes).

    Definition redFuncInstance :=
      redFunc.reduce CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
      CTDTP_REC ICostDTInProtoT ICostDTupTInSigTTsEq ICostDRecTInSigTTsEq
      minC minCMin.
    Definition redFuncInstanceC :=
      redFunc.reduce' CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
      CTDTP_REC ICostDTInProtoT ICostDTupTInSigTTsEq ICostDRecTInSigTTsEq
      minC minCMin.
    Definition crunchFunc :=
      extractFuncFromEnv CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
      CTDTP_REC minC minCMin.

    Definition redAppInst := redApp CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
      CTDTP_REC ICostDTInProtoT minC minCMin redFuncInstanceC.

    Lemma LTypeCeilInLRTNOT_ENOUGH_ARGS : ∀ (t : ProtoT)
      (e : {t : ProtoT & {o : HBCL_0_1_Oid_S.HBCL_OidUT & HBCL_0_1_L_UTSOid.T o t}
      & sig (CTDTP t)}),
   LTypeRawCeiling (projT1 t)
     (existT (projT1 (projT1sigT2 e)) (‘ (projT2 (projT1sigT2 e)))).

    Lemma LRTypesLRTNOT_ENOUGH_ARGS : ∀ (t : ProtoT) lrtraw,
      LRTypesP (projT1 t) lrtraw.

    Definition LRTypesPSFromIOP(iotypes : _)(t : _) :
      OidTypeIOMatch iotypes t →
      LRTypesPS (projT1 t).
    Defined.
Check LTypePS.
Implicit Arguments UPot [].
Check UPot LTypePS DataR DataP CTDT CTDTP ICostDT.

Check ProtoT.
    Definition UDatToUPotMin(u : sigT UDataPST) :
      UPot LTypePS DataR DataP CTDT CTDTP ICostDT
      (projT1 u) (minC (projT1 u)).
    Defined.

    Definition UDatToFssoRaw(r′ : Rsso ICostDT)(u : sigT UDataPST) :
      Fsso _ _ _ _ _ _ (ICostDT := ICostDT) (ICostDTupT := ICostDTupT)
      (ICostDRecT := ICostDRecT) minC r′
      (TssoGenDataT (existT (projT1 u) (minC _))).
    Defined.

    Program Definition updateWclos(r r′ : Rsso ICostDT)
      (wc : sigWssoClos CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC
        minC r)
      (w′ : WssoCpltSI CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC
        r′) :
      sigWssoClos CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC
      minC (VaridMapWPties.update r r′) :=
      (r, existT r′ (w′)) :: (‘wc).
    Obligation 1.
Admitted.

    Program Definition IOtypesConvertFromR
      (iotypes : InpOutpTypes _ CTDTP)
      (t : ProtoT) (iop : OidTypeIOMatch iotypes t)
      (iodat : sig (UDataPSTMatchesInpOutpTypes _ _ iotypes)) :
      UDataPST t :=
      URecordData (projT1 t) (LRTypesPSFromIOP iotypes t iop)
      (HBCL_0_1_Id_S.VaridMapMod.this
        (HBCL_0_1_Id_S.VaridMapMod.map (fun u ⇒ (‘ (projT2 u))) (‘iodat))).
    Obligation 1.
    Admitted.

    Lemma URIncl : ∀ (t : HBCL_0_1_Id_S.Varid × UDataRaw) ts dl l, t
      :: ts = dl →
      SetoidList.inclA (@HBCL_0_1_Id_S.VaridMapMod.eq_key_elt _) dl l →
      SetoidList.inclA (@HBCL_0_1_Id_S.VaridMapMod.eq_key_elt _) ts l.

    Lemma URIn : ∀ p ps dl urm, p :: ps = dl →
      SetoidList.inclA (@HBCL_0_1_Id_S.VaridMapMod.eq_key_elt _) dl
      (HBCL_0_1_Id_S.VaridMapMod.elements (elt:=UDataRaw) urm) →
      HBCL_0_1_Id_S.VaridMapMod.In (fst p) urm.

    Fixpoint sigifyURInner
        (dl : list (HBCL_0_1_Id_S.Varid × UDataRaw))
        (urm : HBCL_0_1_Id_S.VaridMapMod.t UDataRaw)
        (tr : sigT LRTypesPS)
        (urp : URecordP tr (HBCL_0_1_Id_S.VaridMapMod.this urm))
        (inclprf : SetoidList.inclA
          (@HBCL_0_1_Id_S.VaridMapMod.eq_key_elt _ ) dl
```

```
            (HBCL_0_1_Id_S.VaridMapMod.elements urm))
          (inprf : ∀ v, HBCL_0_1_Id_S.VaridMapMod.In v urm →
            HBCL_0_1_Id_S.VaridMapMod.In v
            (LRTypesPSRecoverMap (projT1 tr) (projT2 tr))) { struct dl } :
          HBCL_0_1_Id_S.VaridMapMod.t (sigT UDataPST) :=
          match dl as dl return dl = _ →
            HBCL_0_1_Id_S.VaridMapMod.t (sigT UDataPST) with
            | nil ⇒ fun _ ⇒ (HBCL_0_1_Id_S.VaridMapMod.empty (sigT UDataPST))
            | (cons p ps) ⇒ fun J : (cons p ps) = dl ⇒
              HBCL_0_1_Id_S.VaridMapMod.add (fst p)
              (existT (extractTypeR (fst p) tr (inprf (fst p) _ ))
                (extractDatR (fst p) tr (HBCL_0_1_Id_S.VaridMapMod.this urm)
                  urp (inprf (fst p) (URIn p ps dl urm J inclprf))))
              (sigifyURInner ps urm tr urp (URIncl p ps dl
                (HBCL_0_1_Id_S.VaridMapMod.elements (elt:=UDataRaw) urm)
                J inclprf) inprf)
          end eq_refl.
      Definition sigifyUR(tr : sigT LRTypesPS)
        (urm : HBCL_0_1_Id_S.VaridMapMod.Raw.t UDataRaw)(urp : URecordP tr
          ( urm)) :
        HBCL_0_1_Id_S.VaridMapMod.t (sigT UDataPST).
      Defined.
      Definition IOtypesConvertToR
        (iotypes : InpOutpTypes _ CTDTP)
        (t : ProtoT) (iop : OidTypeIOMatch iotypes t)
        (iodat : UDataPST t) :
        sig (UDataPSTMatchesInpOutpTypes _ _ iotypes).
      Defined.
      Definition computeFunc : sig (UDataPSTMatchesInpOutpTypes _ _ otypes).
Defined.

  End RedFitS.
End exprLangS.
Local  Close Scope program_scope.
Definition CTDTtriv := nat.
Inductive CTDTPtriv(t : HBCL_0_1_L_UTS.ProtoT)(c : CTDTtriv) : Prop :=
  CTDTPtriv_intro : c ≥ 0 → CTDTPtriv t c.
Definition triv_interp(t : HBCL_0_1_L_UTS.ProtoT)(cost : CTDTtriv) :
  ∀ u : sig (HBCL_0_1_L_UTS.DataP t),
    CTDTPtriv t cost → nat := fun _ _ ⇒ cost + 1.
Lemma triv_interp_prf : ∀ (t : sigT HBCL_0_1_L_UTS.TypeS) (cd : CTDTriv)
      (u : sig (HBCL_0_1_L_UTS.DataP t)) (p : CTDTPtriv t cd),
    triv_interp t cd u p ≥ 1.
Definition triv_max(t : HBCL_0_1_L_UTS.ProtoT)(c :sig (CTDTPtriv t)) : nat :=
  proj1_sig c + 1.
Lemma triv_maxCost_prf :
    ∀ t : {t'' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t'')},
    (∀ u : sig (HBCL_0_1_L_UTS.DataP (projT1 t)),
     triv_interp (projT1 t) (‘ (projT2 t))%prg u (proj2_sig (projT2 t)) ≤
     triv_max (projT1 t) (projT2 t)) ∧
    (∃ v : sig (HBCL_0_1_L_UTS.DataP (projT1 t)),
       triv_interp (projT1 t) (‘ (projT2 t))%prg v (proj2_sig (projT2 t)) =
       triv_max (projT1 t) (projT2 t)).
Lemma triv_UPredSeq : ∀ (ur : HBCL_0_1_L_UTS.DataR)
  (t t' : sigT HBCL_0_1_L_UTS.TypeS),
    HBCL_0_1_L_UTS.ProtoEqT t t' →
    HBCL_0_1_L_UTS.DataP t ur → HBCL_0_1_L_UTS.DataP t' ur.
Definition triv_eqbrel : sigT HBCL_0_1_L_UTS.TypeS →
  sigT HBCL_0_1_L_UTS.TypeS → bool.
Defined.
Lemma triv_eqb_eqrel :
  ∀ x y : sigT HBCL_0_1_L_UTS.TypeS,
   triv_eqbrel x y = true ↔ HBCL_0_1_L_UTS.ProtoEqT x y.
Definition triv_eq_struct : sigT (fun t ⇒ sig (CTDTPtriv t)) →
      sigT (fun t ⇒ sig (CTDTPtriv t)) → Prop.
Admitted.
Definition triv_eqb_struct : sigT (fun t ⇒ sig (CTDTPtriv t)) →
      sigT (fun t ⇒ sig (CTDTPtriv t)) → bool.
Defined.
Lemma triv_eqb_eq_struct : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
   triv_eqb_struct x y = true ↔ triv_eq_struct x y.
Lemma triv_eq_struct_c : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
   triv_eq_struct x y → HBCL_0_1_L_UTS.ProtoEqT (projT1 x) (projT1 y).
Lemma triv_eq_struct_Equiv : RelationClasses.Equivalence triv_eq_struct.
Definition triv_eq_pot : sigT (fun t ⇒ sig (CTDTPtriv t)) →
  sigT (fun t ⇒ sig (CTDTPtriv t)) → Prop.
Admitted.
Definition triv_eqb_pot : sigT (fun t ⇒ sig (CTDTPtriv t)) →
      sigT (fun t ⇒ sig (CTDTPtriv t)) → bool.
Defined.
Lemma triv_eqb_eq_pot : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
   triv_eqb_pot x y = true ↔ triv_eq_pot x y.
Lemma triv_eq_pot_c : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
   triv_eq_pot x y → HBCL_0_1_L_UTS.ProtoEqT (projT1 x) (projT1 y).
Instance triv_eq_pot_Equiv : RelationClasses.Equivalence triv_eq_pot.
Admitted.
Lemma triv_eq_pot_eq : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
   triv_eq_pot x y ↔
   (∀ (u : sig (HBCL_0_1_L_UTS.DataP (projT1 x)))
       (v : sig (HBCL_0_1_L_UTS.DataP (projT1 y))),
    triv_interp (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) =
    triv_interp (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).
Definition triv_costFuncConvertRespP :
  ∀ (t : {t : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t)}) t',
    HBCL_0_1_L_UTS.ProtoEqT (projT1 t) t' →
    {t : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t)}.
Defined.
Lemma triv_costFuncRespPCorrect :
  ∀ (t : {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')})
      (t' : sigT HBCL_0_1_L_UTS.TypeS)
      (teq : HBCL_0_1_L_UTS.ProtoEqT (projT1 t) t'),
   triv_eq_pot t (triv_costFuncConvertRespP t t' teq).
Lemma triv_max_tcequiv : ∀ (t :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')})
      (t' : {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')}),
   triv_eq_pot t t' →
   triv_max (projT1 t) (projT2 t) = triv_max (projT1 t') (projT2 t').
Definition triv_le_pot : sigT (fun t ⇒ sig (CTDTPtriv t)) →
  sigT (fun t ⇒ sig (CTDTPtriv t)) → Prop.
Admitted.
Definition triv_leb_pot : sigT (fun t ⇒ sig (CTDTPtriv t)) →
  sigT (fun t ⇒ sig (CTDTPtriv t)) → bool.
Defined.
Lemma triv_leb_le_pot : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
   triv_leb_pot x y = true ↔ triv_le_pot x y.
Lemma triv_le_pot_c : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
    triv_le_pot x y → HBCL_0_1_L_UTS.ProtoEqT (projT1 x) (projT1 y).
Instance triv_le_pot_Equivalence : RelationClasses.Equivalence triv_le_pot.
Admitted.
Instance triv_le_pot_PreOrder : RelationClasses.PreOrder triv_le_pot.
Instance CT_PD_T_le_pot_PartialOrder :
  RelationClasses.PartialOrder triv_eq_pot triv_le_pot.
Admitted.
Lemma triv_le_pot_eq : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
    triv_le_pot x y ↔
    (∀ (u : sig (HBCL_0_1_L_UTS.DataP (projT1 x)))
         (v : sig (HBCL_0_1_L_UTS.DataP (projT1 y))),
      triv_interp (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) ≤
      triv_interp (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).
Definition triv_lt_pot : sigT (fun t ⇒ sig (CTDTPtriv t)) →
  sigT (fun t ⇒ sig (CTDTPtriv t)) → Prop.
Admitted.
Definition triv_ltb_pot : sigT (fun t ⇒ sig (CTDTPtriv t)) →
  sigT (fun t ⇒ sig (CTDTPtriv t)) → bool.
Defined.
Lemma triv_ltb_lt_pot : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
    triv_ltb_pot x y = true ↔ triv_lt_pot x y.
Lemma triv_lt_pot_c : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
    triv_lt_pot x y → HBCL_0_1_L_UTS.ProtoEqT (projT1 x) (projT1 y).
Instance triv_lt_pot_StrOrd : RelationClasses.StrictOrder triv_lt_pot.
Admitted.
Lemma triv_lt_pot_eq : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
    triv_lt_pot x y ↔
    (∀ (u : sig (HBCL_0_1_L_UTS.DataP (projT1 x)))
         (v : sig (HBCL_0_1_L_UTS.DataP (projT1 y))),
      triv_interp (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) <
      triv_interp (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).
Lemma triv_disj : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.TypeS & sig (CTDTPtriv t')},
    (triv_lt_pot x y ∨ triv_eq_pot x y) ∧
    ¬ (triv_lt_pot x y ∧ triv_eq_pot x y) ↔ triv_le_pot x y.
Instance instBTSCostBase : HBCL_0_1_L_UTSCost.CostBase
  (T := HBCL_0_1_L_UTS.TypeS) CTDTtriv CTDTPtriv HBCL_0_1_L_UTS.DataR
  HBCL_0_1_L_UTS.DataP := {
    CT_PD_interp := triv_interp;
    CT_PD_interp_prf := triv_interp_prf;
    CT_PD_max := triv_max;
    maxCost_prf := triv_maxCost_prf;
    CT_PD_T_eqrel := HBCL_0_1_L_UTS.ProtoEqT;
    CT_PD_UPredSeq := triv_UPredSeq;
    CT_PD_T_eqbrel := triv_eqbrel;
    CT_PD_T_eqb_eqrel := triv_eqb_eqrel;
    CT_PD_T_eqrel_Equiv := HBCL_0_1_L_UTS.ProtoTEqTSigT_rel;
    CT_PD_T_eq_struct := triv_eq_struct;
    CT_PD_T_eqb_struct := triv_eqb_struct;
    CT_PD_T_eqb_eq_struct := triv_eqb_eq_struct;
    CT_PD_T_eq_struct_c := triv_eq_struct_c;
    CT_PD_T_eq_struct_Equiv := triv_eq_struct_Equiv;
    CT_PD_T_eq_pot := triv_eq_pot;
    CT_PD_T_eqb_pot := triv_eqb_pot;
    CT_PD_T_eqb_eq_pot := triv_eqb_eq_pot;
    CT_PD_T_eq_pot_c := triv_eq_pot_c;
    CT_PD_T_eq_pot_eq := triv_eq_pot_eq;
    CT_PD_T_costFuncConvertRespP := triv_costFuncConvertRespP;
    CT_PD_T_costFuncRespPCorrect := triv_costFuncRespPCorrect;
    CT_PD_max_tcequiv := triv_max_tcequiv;
    CT_PD_T_le_pot := triv_le_pot;
    CT_PD_T_leb_pot := triv_leb_pot;
    CT_PD_T_leb_le_pot := triv_leb_le_pot;
    CT_PD_T_le_pot_c := triv_le_pot_c;
    CT_PD_T_le_pot_eq := triv_le_pot_eq;
    CT_PD_T_lt_pot := triv_lt_pot;
    CT_PD_T_ltb_pot := triv_ltb_pot;
    CT_PD_T_ltb_lt_pot := triv_ltb_lt_pot;
    CT_PD_T_lt_pot_c := triv_lt_pot_c;
    CT_PD_T_lt_pot_eq := triv_lt_pot_eq;
    CT_PD_disj := triv_disj
  }.
Definition CTDTtrivT := nat.

Inductive CTDTPtrivT(t : sigT HBCL_0_1_L_UTS.LTypesPS)(c : CTDTtrivT) : Prop :=
  CTDTPtrivT_intro : c > 0 → CTDTPtrivT t c.

Definition triv_interpT(t : sigT HBCL_0_1_L_UTS.LTypesPS)(cost : CTDTtrivT) :
  ∀ u : sig (HBCL_0_1_L_UTS.UTupleP t),
    CTDTPtrivT t cost → nat := fun _ _ ⇒ cost.

Lemma triv_interp_prfT :
  ∀ (t : sigT HBCL_0_1_L_UTS.LTypesPS) (cd : CTDTtrivT)
      (u : sig (HBCL_0_1_L_UTS.UTupleP t)) (p : CTDTPtrivT t cd),
    triv_interpT t cd u p ≥ 1.

Definition triv_maxT(t : sigT HBCL_0_1_L_UTS.LTypesPS)
  (c :sig (CTDTPtrivT t)) : nat :=
  proj1_sig c.
Lemma triv_maxCost_prfT :
    ∀ t : {t'' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t'')},
    (∀ u : sig (HBCL_0_1_L_UTS.UTupleP (projT1 t)),
     triv_interpT (projT1 t) (‘ (projT2 t))%prg u (proj2_sig (projT2 t)) ≤
     triv_maxT (projT1 t) (projT2 t)) ∧
    (∃ v : sig (HBCL_0_1_L_UTS.UTupleP (projT1 t)),
       triv_interpT (projT1 t) (‘ (projT2 t))%prg v (proj2_sig (projT2 t)) =
       triv_maxT (projT1 t) (projT2 t)).

Lemma triv_UPredSeqT : ∀ (ur : list HBCL_0_1_L_UTS.DataR)
  (t t' : sigT HBCL_0_1_L_UTS.LTypesPS),
    HBCL_0_1_L_UTS.LTypesPSEqSigT t t' →
    HBCL_0_1_L_UTS.UTupleP t ur → HBCL_0_1_L_UTS.UTupleP t' ur.

Definition triv_eqbrelT : sigT HBCL_0_1_L_UTS.LTypesPS →
  sigT HBCL_0_1_L_UTS.LTypesPS → bool.
Defined.

Lemma triv_eqb_eqrelT :
  ∀ x y : sigT HBCL_0_1_L_UTS.LTypesPS,
    triv_eqbrelT x y = true ↔ HBCL_0_1_L_UTS.LTypesPSEqSigT x y.

Lemma triv_eqrel_EquivT :
  RelationClasses.Equivalence HBCL_0_1_L_UTS.LTypesPSEqSigT.

Definition triv_eq_structT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
      sigT (fun t ⇒ sig (CTDTPtrivT t)) → Prop.
Admitted.

Definition triv_eqb_structT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
      sigT (fun t ⇒ sig (CTDTPtrivT t)) → bool.
Defined.

Lemma triv_eqb_eq_structT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
    triv_eqb_structT x y = true ↔ triv_eq_structT x y.

Lemma triv_eq_struct_cT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
    triv_eq_structT x y → HBCL_0_1_L_UTS.LTypesPSEqSigT (projT1 x) (projT1 y).

Lemma triv_eq_struct_EquivT : RelationClasses.Equivalence triv_eq_structT.

Definition triv_eq_potT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
  sigT (fun t ⇒ sig (CTDTPtrivT t)) → Prop.
Admitted.
```

```
Definition triv_eqb_potT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
        sigT (fun t ⇒ sig (CTDTPtrivT t)) → bool.
Defined.
Lemma triv_eqb_eq_potT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
   triv_eqb_potT x y = true ↔ triv_eq_potT x y.
Lemma triv_eq_pot_cT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
   triv_eq_potT x y → HBCL_0_1_L_UTS.LTypesPSEqSigT (projT1 x) (projT1 y).
Instance triv_eq_pot_EquivT : RelationClasses.Equivalence triv_eq_potT.
Admitted.
Lemma triv_eq_pot_eqT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
   triv_eq_potT x y ↔
   (∀ (u : sig (HBCL_0_1_L_UTS.UTupleP (projT1 x)))
      (v : sig (HBCL_0_1_L_UTS.UTupleP (projT1 y))),
    triv_interpT (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) =
    triv_interpT (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).
Definition triv_costFuncConvertRespPT :
  ∀ (t : {t : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t)}) t',
    HBCL_0_1_L_UTS.LTypesPSEqSigT (projT1 t) t' →
    {t : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t)}.
Defined.
Lemma triv_costFuncRespPCorrectT :
  ∀ (t : {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')})
     (t' : sigT HBCL_0_1_L_UTS.LTypesPS)
     (teq : HBCL_0_1_L_UTS.LTypesPSEqSigT (projT1 t) t'),
   triv_eq_potT t (triv_costFuncConvertRespPT t t' teq).
Lemma triv_max_tcequivT : ∀ (t :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')})
     (t' : {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')}),
   triv_eq_potT t t' →
   triv_maxT (projT1 t) (projT2 t) = triv_maxT (projT1 t') (projT2 t').
Definition triv_le_potT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
  sigT (fun t ⇒ sig (CTDTPtrivT t)) → Prop.
Admitted.
Definition triv_leb_potT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
  sigT (fun t ⇒ sig (CTDTPtrivT t)) → bool.
Defined.
Lemma triv_leb_le_potT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
   triv_leb_potT x y = true ↔ triv_le_potT x y.
Lemma triv_le_pot_cT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
   triv_le_potT x y → HBCL_0_1_L_UTS.LTypesPSEqSigT (projT1 x) (projT1 y).
Instance triv_le_pot_EquivalenceT : RelationClasses.Equivalence triv_le_potT.
Admitted.
Instance triv_le_pot_PreOrderT : RelationClasses.PreOrder triv_le_potT.
Instance CT_PD_T_le_pot_PartialOrderT :
  RelationClasses.PartialOrder triv_eq_potT triv_le_potT.
Admitted.
Lemma triv_le_pot_eqT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
   triv_le_potT x y ↔
   (∀ (u : sig (HBCL_0_1_L_UTS.UTupleP (projT1 x)))
      (v : sig (HBCL_0_1_L_UTS.UTupleP (projT1 y))),
    triv_interpT (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) ≤
    triv_interpT (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).
Definition triv_lt_potT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
  sigT (fun t ⇒ sig (CTDTPtrivT t)) → Prop.
Admitted.
Definition triv_ltb_potT : sigT (fun t ⇒ sig (CTDTPtrivT t)) →
```

```
  sigT (fun t ⇒ sig (CTDTPtrivT t)) → bool.
Defined.

Lemma triv_ltb_lt_potT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
    triv_ltb_potT x y = true ↔ triv_lt_potT x y.

Lemma triv_lt_pot_cT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
    triv_lt_potT x y → HBCL_0_1_L_UTS.LTypesPSEqSigT (projT1 x) (projT1 y).

Instance triv_lt_pot_StrOrdT : RelationClasses.StrictOrder triv_lt_potT.
Admitted.

Lemma triv_lt_pot_eqT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
    triv_lt_potT x y ↔
    (∀ (u : sig (HBCL_0_1_L_UTS.UTupleP (projT1 x)))
         (v : sig (HBCL_0_1_L_UTS.UTupleP (projT1 y))),
      triv_interpT (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) <
      triv_interpT (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).

Lemma triv_disjT : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LTypesPS & sig (CTDTPtrivT t')},
    (triv_lt_potT x y ∨ triv_eq_potT x y) ∧
    ¬ (triv_lt_potT x y ∧ triv_eq_potT x y) ↔ triv_le_potT x y.

Instance instBTSCostTuple : HBCL_0_1_L_UTSCost.CostBase
  (T := HBCL_0_1_L_UTS.LTypesPS) CTDTtrivT CTDTPtrivT
  (list HBCL_0_1_L_UTS.DataR)
  HBCL_0_1_L_UTS.UTupleP := {
    CT_PD_interp := triv_interpT;
    CT_PD_interp_prf := triv_interp_prfT;
    CT_PD_max := triv_maxT;
    maxCost_prf := triv_maxCost_prfT;
    CT_PD_T_eqrel := HBCL_0_1_L_UTS.LTypesPSEqSigT;
    CT_PD_UPredSeq := triv_UPredSeqT;
    CT_PD_T_eqbrel := triv_eqbrelT;
    CT_PD_T_eqb_eqrel := triv_eqb_eqrelT;
    CT_PD_T_eqrel_Equiv := triv_eqrel_EquivT;
    CT_PD_T_eq_struct := triv_eq_structT;
    CT_PD_T_eqb_struct := triv_eqb_structT;
    CT_PD_T_eqb_eq_struct := triv_eqb_eq_structT;
    CT_PD_T_eq_struct_c := triv_eq_struct_cT;
    CT_PD_T_eq_struct_Equiv := triv_eq_struct_EquivT;
    CT_PD_T_eq_pot := triv_eq_potT;
    CT_PD_T_eqb_pot := triv_eqb_potT;
    CT_PD_T_eqb_eq_pot := triv_eqb_eq_potT;
    CT_PD_T_eq_pot_c := triv_eq_pot_cT;
    CT_PD_T_eq_pot_eq := triv_eq_pot_eqT;
    CT_PD_T_costFuncConvertRespP := triv_costFuncConvertRespPT;
    CT_PD_T_costFuncRespPCorrect := triv_costFuncRespPCorrectT;
    CT_PD_max_tcequiv := triv_max_tcequivT;
    CT_PD_T_le_pot := triv_le_potT;
    CT_PD_T_leb_pot := triv_leb_potT;
    CT_PD_T_leb_le_pot := triv_leb_le_potT;
    CT_PD_T_le_pot_c := triv_le_pot_cT;
    CT_PD_T_le_pot_eq := triv_le_pot_eqT;
    CT_PD_T_lt_pot := triv_lt_potT;
    CT_PD_T_ltb_pot := triv_ltb_potT;
    CT_PD_T_ltb_lt_pot := triv_ltb_lt_potT;
    CT_PD_T_lt_pot_c := triv_lt_pot_cT;
    CT_PD_T_lt_pot_eq := triv_lt_pot_eqT;
    CT_PD_disj := triv_disjT
  }.

Definition CTDTtrivR := nat.

Inductive CTDTPtrivR(t : sigT HBCL_0_1_L_UTS.LRTypesPS)(c : CTDTtrivR) : Prop :=
  CTDTPtrivR_intro : c > 0 → CTDTPtrivR t c.

Definition triv_interpR(t : sigT HBCL_0_1_L_UTS.LRTypesPS)(cost : CTDTtrivR) :
  ∀ u : sig (HBCL_0_1_L_UTS.URecordP t),
    CTDTPtrivR t cost → nat := fun _ _ ⇒ cost.
Lemma triv_interp_prfR :
  ∀ (t : sigT HBCL_0_1_L_UTS.LRTypesPS) (cd : CTDTtrivR)
      (u : sig (HBCL_0_1_L_UTS.URecordP t)) (p : CTDTPtrivR t cd),
    triv_interpR t cd u p ≥ 1.
Definition triv_maxR(t : sigT HBCL_0_1_L_UTS.LRTypesPS)
  (c :sig (CTDTPtrivR t)) : nat :=
  proj1_sig c.
Lemma triv_maxCost_prfR :
    ∀ t : {t'' : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t'')},
    (∀ u : sig (HBCL_0_1_L_UTS.URecordP (projT1 t)),
      triv_interpR (projT1 t) (' (projT2 t))%prg u (proj2_sig (projT2 t)) ≤
      triv_maxR (projT1 t) (projT2 t)) ∧
    (∃ v : sig (HBCL_0_1_L_UTS.URecordP (projT1 t)),
        triv_interpR (projT1 t) (' (projT2 t))%prg v (proj2_sig (projT2 t)) =
        triv_maxR (projT1 t) (projT2 t)).
Lemma triv_UPredSeqR : ∀ (ur : HBCL_0_1_Id_S.VaridMapMod.Raw.t
  HBCL_0_1_L_UTS.DataR)
  (t t' : sigT HBCL_0_1_L_UTS.LRTypesPS),
    HBCL_0_1_L_UTS.LRTypesPSEqSigT t t' →
    HBCL_0_1_L_UTS.URecordP t ur →
    HBCL_0_1_L_UTS.URecordP t' ur.

Definition triv_eqbrelR : sigT HBCL_0_1_L_UTS.LRTypesPS →
  sigT HBCL_0_1_L_UTS.LRTypesPS → bool.
Defined.
Lemma triv_eqb_eqrelR :
  ∀ x y : sigT HBCL_0_1_L_UTS.LRTypesPS,
    triv_eqbrelR x y = true ↔ HBCL_0_1_L_UTS.LRTypesPSEqSigT x y.
Lemma triv_eqrel_EquivR :
  RelationClasses.Equivalence HBCL_0_1_L_UTS.LRTypesPSEqSigT.
Definition triv_eq_structR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
        sigT (fun t ⇒ sig (CTDTPtrivR t)) → Prop.
Admitted.
Definition triv_eqb_structR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
        sigT (fun t ⇒ sig (CTDTPtrivR t)) → bool.
Defined.
Lemma triv_eqb_eq_structR : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t')},
    triv_eqb_structR x y = true ↔ triv_eq_structR x y.
Lemma triv_eq_struct_cR : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t')},
    triv_eq_structR x y → HBCL_0_1_L_UTS.LRTypesPSEqSigT (projT1 x) (projT1 y).
Lemma triv_eq_struct_EquivR : RelationClasses.Equivalence triv_eq_structR.
Definition triv_eq_potR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
  sigT (fun t ⇒ sig (CTDTPtrivR t)) → Prop.
Admitted.
Definition triv_eqb_potR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
        sigT (fun t ⇒ sig (CTDTPtrivR t)) → bool.
Defined.
Lemma triv_eqb_eq_potR : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t')},
    triv_eqb_potR x y = true ↔ triv_eq_potR x y.
Lemma triv_eq_pot_cR : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t')},
    triv_eq_potR x y → HBCL_0_1_L_UTS.LRTypesPSEqSigT (projT1 x) (projT1 y).
Instance triv_eq_pot_EquivR : RelationClasses.Equivalence triv_eq_potR.
Admitted.
Lemma triv_eq_pot_eqR : ∀ x y :
  {t' : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t')},
    triv_eq_potR x y ↔
    (∀ (u : sig (HBCL_0_1_L_UTS.URecordP (projT1 x)))
        (v : sig (HBCL_0_1_L_UTS.URecordP (projT1 y))),
      triv_interpR (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) =
      triv_interpR (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).

Definition triv_costFuncConvertRespPR :
  ∀ (t : {t : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t)}) t′,
      HBCL_0_1_L_UTS.LRTypesPSEqSigT (projT1 t) t′ →
      {t : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t)}.
Defined.

Lemma triv_costFuncRespPCorrectR :
  ∀ (t : {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)})
      (t′ : sigT HBCL_0_1_L_UTS.LRTypesPS)
      (teq : HBCL_0_1_L_UTS.LRTypesPSEqSigT (projT1 t) t′),
    triv_eq_potR t (triv_costFuncConvertRespPR t t′ teq).

Lemma triv_max_tcequivR : ∀ (t :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)})
      (t′ : {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)}),
    triv_eq_potR t t′ →
    triv_maxR (projT1 t) (projT2 t) = triv_maxR (projT1 t′) (projT2 t′).

Definition triv_le_potR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
  sigT (fun t ⇒ sig (CTDTPtrivR t)) → Prop.
Admitted.

Definition triv_leb_potR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
  sigT (fun t ⇒ sig (CTDTPtrivR t)) → bool.
Defined.

Lemma triv_leb_le_potR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
    triv_leb_potR x y = true ↔ triv_le_potR x y.

Lemma triv_le_pot_cR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
    triv_le_potR x y → HBCL_0_1_L_UTS.LRTypesPSEqSigT (projT1 x) (projT1 y).

Instance triv_le_pot_EquivalenceR : RelationClasses.Equivalence triv_le_potR.
Admitted.

Instance triv_le_pot_PreOrderR : RelationClasses.PreOrder triv_le_potR.

Instance CT_PD_T_le_pot_PartialOrderR :
  RelationClasses.PartialOrder triv_eq_potR triv_le_potR.
Admitted.

Lemma triv_le_pot_eqR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
    triv_le_potR x y ↔
    (∀ (u : sig (HBCL_0_1_L_UTS.URecordP (projT1 x)))
        (v : sig (HBCL_0_1_L_UTS.URecordP (projT1 y))),
      triv_interpR (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) ≤
      triv_interpR (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).

Definition triv_lt_potR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
  sigT (fun t ⇒ sig (CTDTPtrivR t)) → Prop.
Admitted.

Definition triv_ltb_potR : sigT (fun t ⇒ sig (CTDTPtrivR t)) →
  sigT (fun t ⇒ sig (CTDTPtrivR t)) → bool.
Defined.

Lemma triv_ltb_lt_potR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
    triv_ltb_potR x y = true ↔ triv_lt_potR x y.

Lemma triv_lt_pot_cR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
    triv_lt_potR x y → HBCL_0_1_L_UTS.LRTypesPSEqSigT (projT1 x) (projT1 y).

Instance triv_lt_pot_StrOrdR : RelationClasses.StrictOrder triv_lt_potR.
Admitted.

Lemma triv_lt_pot_eqR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
    triv_lt_potR x y ↔
    (∀ (u : sig (HBCL_0_1_L_UTS.URecordP (projT1 x)))
        (v : sig (HBCL_0_1_L_UTS.URecordP (projT1 y))),
    triv_interpR (projT1 x) (‘ (projT2 x))%prg u (proj2_sig (projT2 x)) <
    triv_interpR (projT1 y) (‘ (projT2 y))%prg v (proj2_sig (projT2 y))).
Lemma triv_disjR : ∀ x y :
  {t′ : sigT HBCL_0_1_L_UTS.LRTypesPS & sig (CTDTPtrivR t′)},
   (triv_lt_potR x y ∨ triv_eq_potR x y) ∧
   ¬ (triv_lt_potR x y ∧ triv_eq_potR x y) ↔ triv_le_potR x y.
Instance instBTSCostRecord : HBCL_0_1_L_UTSCost.CostBase
  (T := HBCL_0_1_L_UTS.LRTypesPS) CTDTtrivR CTDTPtrivR
  (HBCL_0_1_Id_S.VaridMapMod.Raw.t HBCL_0_1_L_UTS.DataR)

  HBCL_0_1_L_UTS.URecordP
 := {
    CT_PD_interp := triv_interpR;
    CT_PD_interp_prf := triv_interp_prfR;
    CT_PD_max := triv_maxR;
    maxCost_prf := triv_maxCost_prfR;
    CT_PD_T_eqrel := HBCL_0_1_L_UTS.LRTypesPSEqSigT;
    CT_PD_UPredSeq := triv_UPredSeqR;
    CT_PD_T_eqbrel := triv_eqbrelR;
    CT_PD_T_eqb_eqrel := triv_eqb_eqrelR;
    CT_PD_T_eqrel_Equiv := triv_eqrel_EquivR;
    CT_PD_T_eq_struct := triv_eq_structR;
    CT_PD_T_eqb_struct := triv_eqb_structR;
    CT_PD_T_eqb_eq_struct := triv_eqb_eq_structR;
    CT_PD_T_eq_struct_c := triv_eq_struct_cR;
    CT_PD_T_eq_struct_Equiv := triv_eq_struct_EquivR;
    CT_PD_T_eq_pot := triv_eq_potR;
    CT_PD_T_eqb_pot := triv_eqb_potR;
    CT_PD_T_eqb_eq_pot := triv_eqb_eq_potR;
    CT_PD_T_eq_pot_c := triv_eq_pot_cR;
    CT_PD_T_eq_pot_eq := triv_eq_pot_eqR;
    CT_PD_T_costFuncConvertRespP := triv_costFuncConvertRespPR;
    CT_PD_T_costFuncRespPCorrect := triv_costFuncRespPCorrectR;
    CT_PD_max_tcequiv := triv_max_tcequivR;
    CT_PD_T_le_pot := triv_le_potR;
    CT_PD_T_leb_pot := triv_leb_potR;
    CT_PD_T_leb_le_pot := triv_leb_le_potR;
    CT_PD_T_le_pot_c := triv_le_pot_cR;
    CT_PD_T_le_pot_eq := triv_le_pot_eqR;
    CT_PD_T_lt_pot := triv_lt_potR;
    CT_PD_T_ltb_pot := triv_ltb_potR;
    CT_PD_T_ltb_lt_pot := triv_ltb_lt_potR;
    CT_PD_T_lt_pot_c := triv_lt_pot_cR;
    CT_PD_T_lt_pot_eq := triv_lt_pot_eqR;
    CT_PD_disj := triv_disjR
  }.
Program Definition minCtriv :
  ∀ t : HBCL_0_1_L_UTS.ProtoT, sig (CTDTPtriv t) :=
    fun _ ⇒ 0.
Obligation 1.
Lemma TEqEquiv:
  HBCL_0_1_L_UTSCost.CT_PD_T_eqrel = HBCL_0_1_L_UTS.ProtoEqT.
Lemma TSEqEquiv:
  HBCL_0_1_L_UTSCost.CT_PD_T_eqrel = HBCL_0_1_L_UTS.LTypesPSEqSigT.
Lemma TREqEquiv:
HBCL_0_1_L_UTSCost.CT_PD_T_eqrel = HBCL_0_1_L_UTS.LRTypesPSEqSigT.
Check computeFunc.
Lemma minCMinTriv : ∀ t u,
  HBCL_0_1_L_UTSCost.CT_PD_interp(CostBase := instBTSCostBase)
    t (proj1_sig (minCtriv t)) u (proj2_sig (minCtriv t)) = 0.
Require Import Coq.Strings.String.
Print sso.

Definition UBitLang : HBCL_0_1_L_UBoxEmtpy.UExprLang := {|
  HBCL_0_1_L_UBoxEmtpy.CTDT := CTDTtriv;
  HBCL_0_1_L_UBoxEmtpy.CTDTP := CTDTPtriv;
  HBCL_0_1_L_UBoxEmtpy.costB := instBTSCostBase;
  HBCL_0_1_L_UBoxEmtpy.AST := AST;
  HBCL_0_1_L_UBoxEmtpy.parse := parse;
  HBCL_0_1_L_UBoxEmtpy.sso := sso _ CTDTtrivT CTDTtrivR _
  CTDTPtrivT CTDTPtrivR (ICostDTupT := instBTSCostTuple)
  (ICostDRecT := instBTSCostRecord) minCtriv;
  HBCL_0_1_L_UBoxEmtpy.compile := compile _ _ _ _ _ _ _ ;

  HBCL_0_1_L_UBoxEmtpy.reduce := computeFunc _ _ _ _ _ _ (ICostDT := instBTSCostBase) TEqEquiv TSEqEquiv TREqEquiv
_ minCMinTriv

|}.
```

Listing D.29: The expression language expression type

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Arith.Lt.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Wellfounded.Inverse_Image.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.

Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.

Local Open Scope program_scope.

Module VaridMapWFacts :=
  FMapFacts.WFacts_fun varidPred.PredidDecidable VaridMapMod.
Module VaridMapWPties :=
  FMapFacts.WProperties_fun varidPred.PredidDecidable VaridMapMod.

Implicit Arguments existT [A P].
Implicit Arguments existTD [A B P].
Implicit Arguments existTT [A B C P].

Notation varidEqb := varidPred.PredidDecidable.eqb.
Infix "=v=" := varidPred.PredidDecidable.eq (at level 70, no associativity).
Infix "=t=" := ProtoEqTSigT (at level 70, no associativity).

Section AST.

  Definition Cost := {n : nat | 1 ≤ n}.

  Inductive Boolconst : Set := bcTrue | bcFalse.

  Inductive Baseconst : Set :=
    BaseconstBool : Boolconst → Baseconst.

  Inductive Vardeclprim : Set :=
  | vardeclprimVar : Varid → TypeRepres → Vardeclprim
  | vardeclprimFun : Varid → TypeRepres → TypeRepres → Cost → Vardeclprim

  with TypeRepres : Set :=
  | TypeRepresBase : LBasetype → TypeRepres
  | TypeRepresTup : list (TypeRepres) → TypeRepres
  | TypeRepresRecord : list (Vardeclprim) → TypeRepres.
  Inductive PattEl : Set :=
  | pattVarid : Varid → PattEl
  | pattPosParam : nat → PattEl.
  Inductive Patt : Set :=
  | patt : Varid → list PattEl → Patt.
  Inductive Expr : Set :=
  | exprConstr : Constr → Expr
  | exprPatt : Patt → Expr
  | exprApp : Patt → Expr → Expr

  with Constr : Set :=
  | constrBase : Baseconst → Constr
  | constrTup : list Expr → Constr
  | constrRecord : list (Patt × Expr) → Constr.
  Inductive Vardefprim : Set :=
    vardefprim : Varid → Expr → Vardefprim.
  Inductive Vardef : Set :=
  | vardefExpr : Expr → Vardef
  | vardefFun : Varid → Expr → Vardef.
  Inductive Vardecl : Set :=
  | vardeclVardeclprim : Vardeclprim → Vardecl
  | vardeclVardef : Vardeclprim → Vardef → Vardecl.
  Inductive Decl : Set :=
  | declVardecl : Vardecl → Decl
  | declVardef : Vardefprim → Decl.
  Definition Decls := list Decl.
  Inductive Program : Set :=
    program : Decls → Program.
End AST.

Section EssoS.
  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).
  Context ‘{ICostDT : CostDT CTDT CTDTP}.
  Context ‘{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context ‘{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.
  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.
  Variable minC : ∀ t : ProtoT, sig (CTDTP t).
  Hypothesis minCMin : ∀ t c u, CT_PD_interp(CostBase := ICostDT)
    t (‘c) u (“c) = 0.

  Lemma minTCeq : ∀ t t′, t =t= t′ →
    existT t (minC t) =tc= existT t′ (minC t′).

  Definition maxCostLt(t t′ : {t : ProtoT & sig (CTDTP t)}) : Prop :=
  ∀ u v,
    CT_PD_interp (projT1 t) (proj1_sig (projT2 t)) u (proj2_sig (projT2 t)) <
    CT_PD_interp (projT1 t′) (proj1_sig (projT2 t′)) v (proj2_sig (projT2 t′))
    ∧
    ¬ (∃ v′,
      CT_PD_interp (projT1 t′) (proj1_sig (projT2 t′)) v
      (proj2_sig (projT2 t′)) <
      CT_PD_interp (projT1 t′) (proj1_sig (projT2 t′)) v′
    (proj2_sig (projT2 t'))).
Instance maxCostLtTrans : Transitive maxCostLt.
Instance maxCostLtIrrflx : Irreflexive maxCostLt.
Instance maxCostStrOrd : StrictOrder maxCostLt :=
  {| StrictOrder_Transitive := maxCostLtTrans;
     StrictOrder_Irreflexive := maxCostLtIrrflx |}.
Definition minCSig(t : ProtoT) :=
  existT (P := fun t' ⇒ sig (CTDTP t')) t (minC t).
Lemma minCSigMin : ∀ (t : ProtoT)(t' : {t : ProtoT & sig (CTDTP t)}),
  ¬ (t' =tc= (minCSig (projT1 t'))) → maxCostLt (minCSig t) t'.

Lemma TssoLeInjFunc : ∀ v t1 t2 v' t1' t2',
  v =v= v' → t1 =t= t1' → t2 <tc= t2' →
  TssoGenFunc v t1 t2 <TC= TssoGenFunc v' t1' t2'.
Lemma ProtoT_eq_refl : ∀ t, t =t= t.
Implicit Arguments ProtoT_eq_refl [t].
Definition CostD(T : Size → Type)(CTDT' : Type)
  (CTDTP' : (sigT T) → CTDT' → Prop) := {t : sigT T & sig (CTDTP' t)}.
Definition CTCombAppPrf
  (t t': ProtoT)
  (c : sig (CTDTP t))(c' : sig (CTDTP t'))(c'' : sig (CTDTP t')) :=
  ∀ u u',
    CT_PD_interp t (‘c) u (“c) +
    CT_PD_interp t' (‘c') u' (“c') =
    CT_PD_interp t' (‘c'') u' (“c'').
Implicit Arguments CTCombAppPrf [t t'].
Definition tInTList(t : ProtoT)(ts : sigT LTypesPS) :=
  ∃ t', t =t= t' ∧
  List.In (existT (projT1 t') (proj1_sig (projT2 t')))
  (List.map (stripCeiling (projT1 ts)) (proj1_sig (projT2 ts))).
Definition tInTMap(t : ProtoT)(tr : sigT LRTypesPS) :=
  ∃ v, ∃ t', t =t= t' ∧
    VaridMapMod.MapsTo v (existT (projT1 t') (proj1_sig (projT2 t')))
    (VaridMapMod.map (stripCeiling (projT1 tr))
    (LRTypesPSRecoverMap (projT1 tr) (projT2 tr))).
Definition tListInT(ts : sigT LTypesPS)(t : ProtoT) :=
  ∃ ts' : sigT LTypesPS, SetoidList.eqlistA ProtoEqTSigT
    (sigifyList (projT1 ts) (projT2 ts))
    (sigifyList (projT1 ts') (projT2 ts')) →
    t = buildProtoTFromSigTS ts'.
Definition tMapInT(tr : sigT LRTypesPS)(t : ProtoT) :=
  ∃ tr' : sigT LRTypesPS, VaridMapMod.Equiv ProtoEqTSigT
    (sigifyMap (projT1 tr) (projT2 tr))
    (sigifyMap (projT1 tr') (projT2 tr')) →
    t = buildProtoTFromSigTR tr'.
Inductive EssoRaw : Type :=
| essoConstr(t : ProtoT)(c : sig (CTDTP t)): CssoRaw → EssoRaw
| essoPatt(t : ProtoT)(c : sig (CTDTP t)) : Patt → EssoRaw
| essoApp(t : ProtoT)(c : sig (CTDTP t))
  (e : EssoRaw) : String.string → EssoRaw

 with CssoRaw : Type :=
| cssoBase(lbt : LBasetype)(ubt : UBasetype lbt)
  (c : sig (CTDTP (existT 1 (BuildBaseTypePS 1 lbt eq_refl)))) : CssoRaw
| cssoTuple(ts : sigT LTypesPS)(cs : sig (CTDTP_TUP ts)) :
  list EssoRaw → CssoRaw
| cssoRecord(tr : sigT LRTypesPS)(cr : sig (CTDTP_REC tr)) :
  VaridMapModRaw.t EssoRaw → CssoRaw.

Inductive EssoRawCeilingL(ts : sigT LTypesPS)(cceil: sig (CTDTP_TUP ts)) :
  EssoRaw → Prop :=
| EssoCeil_introL(raw : EssoRaw) :
  tInTList (projT1 (getEssoRawTC raw)) ts →
  CT_PD_LT_HET tInTList lCostDT lCostDTupT (projT2 (getEssoRawTC raw))
  cceil → EssoRawCeilingL ts cceil raw.
Inductive EssoRawCeilingR(tr : sigT LRTypesPS)(cceil: sig (CTDTP_REC tr)) :
  EssoRaw → Prop :=
| EssoCeil_introR (raw : EssoRaw) :
  tInTMap (projT1 (getEssoRawTC raw)) tr →
  CT_PD_LT_HET tInTMap lCostDT lCostDRecT (projT2 (getEssoRawTC raw))
  cceil → EssoRawCeilingR tr cceil raw.
Record UCost : Type := {
  ucType : sigT LTypePS;
  ucCost : sig (CTDTP ucType);
  ucDat : sig (UDataP ucType)
}.
Record UCostTup : Type := {
  ucTupTypes : sigT LTypesPS;
  ucTupCost : sig (CTDTP_TUP ucTupTypes);
  ucTupDat : sig (UTupleP ucTupTypes)
}.
Record UCostRec : Type := {
  ucRecTypes : sigT LRTypesPS;
  ucRecCost : sig (CTDTP_REC ucRecTypes);
  ucRecDat : sig (URecordP ucRecTypes)
}.
Definition UDatInterp(t : { t : (sigT LTypePS) & sig (CTDTP t)})
  (u : sig (UDataP (projT1 t))) : Potential :=
  CT_PD_interp (projT1 t) (‘ (projT2 t)) u (“ (projT2 (t))).
Definition UDatListFoldFunc(u : UCost)(inPot : Potential) : Potential :=
  inPot + UDatInterp (existT (ucType u) (ucCost u)) (ucDat u).
Definition UDatMapFoldFunc(_ : Varid)(u : UCost)(inPot : Potential) :
  Potential :=
  inPot + UDatInterp (existT (ucType u) (ucCost u)) (ucDat u).
Section UCostTupSigifyLemmasSect.
  Variable tcl : list {t″ : ProtoT & sig (CTDTP t″)}.
  Variable udl : list UDataRaw.
  Hypothesis inprf : ∀ pr : {t″ : ProtoT & sig (CTDTP t″)} × UDataRaw,
    In pr (combine tcl udl) → UDataP (projT1 (fst pr)) (snd pr).
  Variable l : list {t″ : ProtoT & sig (CTDTP t″)}.
  Variable l0 : list UDataRaw.
  Variable tc : {t″ : ProtoT & sig (CTDTP t″)}.
  Variable tcl′ : list {t″ : ProtoT & sig (CTDTP t″)}.
  Variable ud : UDataRaw.
  Variable udl′ : list UDataRaw.
  Variable J : (tc :: tcl′, ud :: udl′) = (tcl, udl).
  Lemma UCostTupSigUDataP : UDataP (projT1 tc) ud.
  Lemma UCostTupInPrfRec :
    ∀ pr : {t″ : ProtoT & sig (CTDTP t″)} × UDataRaw,
      In pr (combine tcl′ udl′) → UDataP (projT1 (fst pr)) (snd pr).
End UCostTupSigifyLemmasSect.
Fixpoint UCostTupSigifyListAssistInner
  (tcl : list ( {t″ : ProtoT & sig (CTDTP t″)})) (udl : list UDataRaw)
  (inprf : ∀ pr, List.In pr (List.combine
    tcl udl) →
  UDataP (projT1 (fst pr)) (snd pr)) { struct udl } : list UCost
  := match (tcl, udl) as tup return tup = _ → list UCost with
       | (tc :: tcl′, ud :: udl′) ⇒
         fun J : (tc :: tcl′, ud :: udl′) = (tcl, udl) ⇒
           { | ucType := (projT1 tc); ucCost := (projT2 tc);
             ucDat := exist (UDataP (projT1 tc)) ud
             (UCostTupSigUDataP tcl udl inprf tc tcl′ ud udl′ J) | } ::
           (UCostTupSigifyListAssistInner tcl′ udl′
             (UCostTupInPrfRec tcl udl inprf tc tcl′ ud udl′ J))
       | _ ⇒ fun _ ⇒ nil
     end eq_refl.
Definition UCostTupSigifyListAssist(ul : list UDataRaw)(erl′ : list EssoRaw)
```

```
  (uprf : ∀ pr : {t : ProtoT & sig (CTDTP t)} × UDataRaw,
     In pr (combine (map (fun e : EssoRaw ⇒ getEssoRawTC e) erl′) ul) →
     UDataP (projT1 (fst pr)) (snd pr)) : list UCost :=
  UCostTupSigifyListAssistInner
  (List.map (fun e ⇒ (getEssoRawTC e)) erl′) ul uprf.

  Section UCostRecSigifyLemmasSect.

  Variable tcm : VaridMapMod.t {t″ : ProtoT & sig (CTDTP t″)}.
  Variable um : VaridMapMod.t UDataRaw.
  Variable udl : list (Varid × UDataRaw).
  Hypothesis inclprf : SetoidList.inclA (@VaridMapMod.eq_key_elt _) udl
                 (VaridMapMod.elements (elt:=UDataRaw) um).

  Hypothesis inprf : ∀ pr, VaridMapMod.MapsTo (fst pr) (snd pr) um →
     ∃ t, VaridMapMod.MapsTo (fst pr) t tcm ∧
       UDataP (projT1 t) (snd pr).
  Variable u : Varid × UDataRaw.
  Variable us : list (Varid × UDataRaw).
  Hypothesis J : u :: us = udl.

  Section UCostRecSfyLemmInS.

     Variable tc : {t″ : ProtoT & sig (CTDTP t″)}.
     Hypothesis J0 : VaridMapMod.find (fst u) tcm = Some tc.

     Lemma UDataPSfyRec : UDataP (projT1 tc) (snd u).

     Lemma PSfyRecNewInclPrf :
       SetoidList.inclA (@VaridMapMod.eq_key_elt _) us
       (VaridMapMod.elements (elt:=UDataRaw) um).

  End UCostRecSfyLemmInS.

  Lemma PSfyNotInFalse : ¬ VaridMapMod.find (fst u) tcm = None.

End UCostRecSigifyLemmasSect.

Fixpoint UCostRecSigifyMapAssistInner
  (tcm : VaridMapMod.t {t″ : ProtoT & sig (CTDTP t″)})
  (um : VaridMapMod.t UDataRaw)
  (udl : list (Varid × UDataRaw))
  (inclprf : SetoidList.inclA (@VaridMapMod.eq_key_elt _ ) udl
     (VaridMapMod.elements um))
  (inprf : ∀ pr, VaridMapMod.MapsTo (fst pr) (snd pr) um →
     ∃ t, VaridMapMod.MapsTo (fst pr) t tcm ∧
       UDataP (projT1 t) (snd pr)) { struct udl } :
  VaridMapMod.t UCost :=
  match udl as udl return udl = _ → VaridMapMod.t UCost with
     | u :: us ⇒ fun J : u :: us = udl ⇒
       match (VaridMapMod.find (fst u) tcm) as tf return _ = tf → _ with
          | Some tc ⇒ fun J0 ⇒ VaridMapMod.add (fst u)
             {| ucType := projT1 tc; ucCost := projT2 tc;
                ucDat := exist (UDataP (projT1 tc)) (snd u)
                (UDataPSfyRec tcm um udl inclprf inprf u us J tc J0)
             |}
             (UCostRecSigifyMapAssistInner tcm um us
                (PSfyRecNewInclPrf tcm um udl inclprf u us J tc J0) inprf)
          | None ⇒ fun J0 ⇒
             False_rect _ (PSfyNotInFalse tcm um udl inclprf inprf u us J J0)
       end eq_refl
     | nil ⇒ fun _ ⇒ (VaridMapMod.empty UCost)
  end eq_refl.

Lemma UCostRecPrfImpl : ∀ um erm′,
  ((∀ v : VaridMapMod.key,
     VaridMapMod.In (elt:=UDataRaw) v um ↔
     VaridMapMod.In (elt:=EssoRaw) v erm′) ∧
  (∀ (v : VaridMapMod.key) (ur : UDataRaw),
     VaridMapMod.MapsTo v ur um →
         ∃ er : EssoRaw,
           VaridMapMod.MapsTo v er erm′ ∧
           UDataP (projT1 (getEssoRawTC er)) ur)) →
(∀ pr : Varid × UDataRaw,
  SetoidList.InA (@VaridMapMod.eq_key_elt _) pr (VaridMapMod.elements um) →
  ∃ t : {t″ : ProtoT & sig (CTDTP t″)},
    VaridMapMod.MapsTo (fst pr) t
    (VaridMapMod.map (fun e : EssoRaw ⇒ getEssoRawTC e) erm′) ∧
    UDataP (projT1 t) (snd pr)).
Lemma UDatImplRecAssist : ∀ erm′ um, (∀ pr : Varid × UDataRaw,
  SetoidList.InA (@VaridMapMod.eq_key_elt _) pr
  (VaridMapMod.elements (elt:=UDataRaw) um) →
  ∃ t : {t″ : ProtoT & sig (CTDTP t″)},
    VaridMapMod.MapsTo (fst pr) t
    (VaridMapMod.map (fun e : EssoRaw ⇒ getEssoRawTC e) erm′) ∧
    UDataP (projT1 t) (snd pr)) →
(∀ pr : VaridMapMod.key × UDataRaw,
  VaridMapMod.MapsTo (fst pr) (snd pr) um →
  ∃ t : {t″ : ProtoT & sig (CTDTP t″)},
    VaridMapMod.MapsTo (fst pr) t
    (VaridMapMod.map (fun e : EssoRaw ⇒ getEssoRawTC e) erm′) ∧
    UDataP (projT1 t) (snd pr)).
Lemma udatInclRefl : ∀ um, SetoidList.inclA
  (@VaridMapMod.eq_key_elt _)
  (VaridMapMod.elements (elt:=UDataRaw) um)
  (VaridMapMod.elements (elt:=UDataRaw) um).
Definition UCostRecSigifyMapAssist(um : VaridMapMod.t UDataRaw)
  (erm′ : VaridMapMod.t EssoRaw)
  (uprf : (∀ v, VaridMapMod.In v um ↔ VaridMapMod.In v erm′) ∧
  (∀ v ur, VaridMapMod.MapsTo v ur um →
    ∃ er, VaridMapMod.MapsTo v er erm′ ∧
      UDataP (projT1 (getEssoRawTC er)) ur)) : VaridMapMod.t UCost :=
  UCostRecSigifyMapAssistInner
  (VaridMapMod.map (fun e ⇒ (getEssoRawTC e)) erm′) um
  (VaridMapMod.elements um) (udatInclRefl um)
  (UDatImplRecAssist erm′ um (UCostRecPrfImpl um erm′ uprf)).
Section UDataPredExtractSect.
  Variable ut : UCostTup.
  Variable erl′ : list EssoRaw.
  Hypothesis uctprf : LTypePSListEqSigT
    (sigifyList (projT1 (ucTupTypes ut)) (projT2 (ucTupTypes ut)))
    (map (fun e : EssoRaw ⇒ projT1 (getEssoRawTC e)) erl′).
  Lemma UDataPredExtract :
    ∀ pr : {t : ProtoT & sig (CTDTP t)} × UDataRaw,
    In pr (combine (map (fun e : EssoRaw ⇒ getEssoRawTC e) erl′)
      (‘ (ucTupDat ut))) → UDataP (projT1 (fst pr)) (snd pr).
End UDataPredExtractSect.
Lemma URecordMapRecoverNoDup : ∀ lrtps (urps : sig (URecordP lrtps)),
  SetoidList.NoDupA (@VaridMapMod.Raw.PX.eqk _) (‘urps).
Definition URecordPSRecoverUDRMap(lrtps : sigT LRTypesPS)
  (urps : sig (URecordP lrtps)) : VaridMapMod.t UDataRaw :=
  VaridMapMod.Build_slist (this := (‘urps))
  (URecordMapRecoverNoDup lrtps urps).
Section UDataPredExtractRSect.
  Variable ur : UCostRec.
  Variable erm : VaridMapMod.t EssoRaw.
  Hypothesis uctprf : LTypePSMapEqSigT
    (sigifyMap (projT1 (ucRecTypes ur)) (projT2 (ucRecTypes ur)))
    (VaridMapMod.map (fun e : EssoRaw ⇒ projT1 (getEssoRawTC e))
      erm).
  Section sigifyMapInnerEquivInAssistS.
    Variable s : Size.
    Variable l : (fun s′ : Size ⇒ list (Varid × sig (LTypeRawCeiling s′))) s.
    Variable dl :
      (fun s′ : Size ⇒ list (Varid × sig (LTypeRawCeiling s′))) s.
    Hypothesis inclp : SetoidList.inclA (@VaridMapMod.eq_key_elt _) dl l.
    Hypothesis inpredp : ∀ pr : Varid × sig (LTypeRawCeiling s),
      SetoidList.InA (@VaridMapMod.eq_key_elt _) pr l →
```

```
        LTypeP (projT1 (‘ (snd pr))) (projT2 (‘ (snd pr))).
      Variable s′ : Size.
      Variable l′ :
        (fun s′ : Size ⇒ list (Varid × sig (LTypeRawCeiling s′))) s′.
      Variable dl′ :
        (fun s′ : Size ⇒ list (Varid × sig (LTypeRawCeiling s′))) s′.
      Hypothesis inclp′ : SetoidList.inclA (@VaridMapMod.eq_key_elt _) dl′ l′.
      Hypothesis inpredp′ : ∀ pr : Varid × sig (LTypeRawCeiling s′),
        SetoidList.InA (@VaridMapMod.eq_key_elt _) pr l′ →
        LTypeP (projT1 (‘ (snd pr))) (projT2 (‘ (snd pr))).
      Hypothesis H : s = s′.
      Hypothesis H0 : existT s l = existT s′ l′.
      Hypothesis H1 : existT s dl = existT s′ dl′.
      Variable v : VaridMapMod.key.

      Lemma sigifyMapInnerEquivInAssist : (∃ e : ProtoT,
      SetoidList.InA (@VaridMapMod.eq_key_elt _) (v, e)
        (VaridMapMod.elements (elt:=ProtoT)
          (sigifyMapInner s l dl inclp inpredp))) →
      ∃ e : ProtoT,
        SetoidList.InA (@VaridMapMod.eq_key_elt _) (v, e)
        (VaridMapMod.elements (elt:=ProtoT)
          (sigifyMapInner s′ l′ dl′ inclp′ inpredp′)).

    End sigifyMapInnerEquivInAssistS.

    Lemma sigifyMapInnerEquiv :
      ∀ s l dl inclp inpredp s′ l′ dl′ inclp′ inpredp′,
        s = s′ → existT s l = existT s′ l′ →
        existT s dl = existT s′ dl′ → VaridMapMod.Equiv ProtoEqTSigT
        (sigifyMapInner s l dl inclp inpredp)
        (sigifyMapInner s′ l′ dl′ inclp′ inpredp′).

    Add Parametric Morphism elt (R : relation elt) : (@VaridMapMod.In elt)
      with signature varidPred.PredidDecidable.eq ==> VaridMapMod.Equiv R
        ==> iff as In_m.

    Theorem MapsToEquivExist : ∀ elt (m m′ : VaridMapMod.t elt) x e R,
        VaridMapMod.Equiv R m m′ → VaridMapMod.MapsTo x e m →
        ∃ e′, R e e′ ∧ VaridMapMod.MapsTo x e′ m′.
    Implicit Arguments MapsToEquivExist [elt R].

    Lemma sigifyMapInvIn : ∀ v (tr : sigT LRTypesPS),
      (∃ mok, VaridMapMod.In v (VaridMapMod.Build_slist
        (this := (‘ (projT2 tr))) mok)) ↔
      VaridMapMod.In v (sigifyMap (projT1 tr) (projT2 tr)).

    Lemma sigifyMapInvInRM : ∀ v tr,
      VaridMapMod.In v (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)) ↔
      VaridMapMod.In v (sigifyMap (projT1 tr) (projT2 tr)).

Lemma eqkEltImplEqkIn : ∀ elt p l,
  SetoidList.InA (@VaridMapMod.eq_key_elt elt) p l →
  SetoidList.InA (@VaridMapMod.eq_key elt) p l.

Lemma eqkEltImplEqkIn2 : ∀ elt k e e′ l,
  SetoidList.InA (@VaridMapMod.eq_key_elt elt) (k, e) l →
  SetoidList.InA (@VaridMapMod.eq_key elt) (k, e′) l.

Lemma eqkEltImplEqkIncl : ∀ elt l l′,
  SetoidList.inclA (@VaridMapMod.eq_key_elt elt) l l′ →
  SetoidList.inclA (@VaridMapMod.eq_key elt) l l′.

Lemma InAKEltKeyImpl : ∀ elt p l,
  SetoidList.InA (@VaridMapMod.eq_key_elt elt) p l →
  SetoidList.InA (@VaridMapMod.eq_key _) p l.

    Lemma sigifyMapInvMapRM : ∀ v tr t,
      VaridMapMod.MapsTo v t (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)) →
      ∃ t′, ‘t = existT (projT1 t′) (‘ (projT2 t′)) ∧
        ∃ t″, t′ =t= t″ ∧
        VaridMapMod.MapsTo v t″ (sigifyMap (projT1 tr) (projT2 tr)).

    Lemma UDataPredExtractR :
      (∀ v : VaridMapMod.key,
        VaridMapMod.In (elt:=UDataRaw) v
      (URecordPSRecoverUDRMap (ucRecTypes ur) (ucRecDat ur)) ↔
      VaridMapMod.In (elt:=EssoRaw) v erm) ∧
    (∀ (v : VaridMapMod.key) (ur' : UDataRaw),
      VaridMapMod.MapsTo v ur'
      (URecordPSRecoverUDRMap (ucRecTypes ur) (ucRecDat ur)) →
      ∃ er : EssoRaw,
        VaridMapMod.MapsTo v er erm ∧ UDataP (projT1 (getEssoRawTC er)) ur').
End UDataPredExtractRSect.

Definition UCostTupSigifyList(ut : UCostTup)(erl : list EssoRaw)
  (uctprf : LTypePSListEqSigT (sigifyList (projT1 (ucTupTypes ut))
    (projT2 (ucTupTypes ut)))
  (List.map (fun e ⇒ (projT1 (getEssoRawTC e))) erl)) : list UCost :=
  UCostTupSigifyListAssist
  (' (ucTupDat ut)) erl (UDataPredExtract ut erl uctprf).

Definition ListCostLe (ut : UCostTup)(erl : list EssoRaw)
  (uctprf : LTypePSListEqSigT (sigifyList (projT1 (ucTupTypes ut))
    (projT2 (ucTupTypes ut)))
  (List.map (fun e ⇒ (projT1 (getEssoRawTC e))) erl)) :=
  List.fold_right UDatListFoldFunc 0 (UCostTupSigifyList ut erl uctprf) ≤
  CT_PD_interp (ucTupTypes ut) (' (ucTupCost ut)) (ucTupDat ut)
  (“(ucTupCost ut)).

Definition ListCostLeTQuant(ts : sigT LTypesPS)(cs : sig (CTDTP_TUP ts))
  (erl : list EssoRaw) :=
  ∀ (us : sig (UTupleP ts)),
    let ut := Build_UCostTup ts cs us in
      ∀ (uctprf : LTypePSListEqSigT (sigifyList (projT1 (ucTupTypes ut))
        (projT2 (ucTupTypes ut)))
      (List.map (fun e ⇒ (projT1 (getEssoRawTC e))) erl)),
      ListCostLe ut erl uctprf.

Definition UCostRecSigifyMap(ur : UCostRec)(erm : VaridMapMod.t EssoRaw)
  (uctprf : LTypePSMapEqSigT (sigifyMap (projT1 (ucRecTypes ur))
    (projT2 (ucRecTypes ur)))
  (VaridMapMod.map (fun e ⇒ (projT1 (getEssoRawTC e))) erm)) :
  VaridMapMod.t UCost :=
  UCostRecSigifyMapAssist
  (URecordPSRecoverUDRMap (ucRecTypes ur) (ucRecDat ur)) erm
  (UDataPredExtractR ur erm uctprf).

Definition MapCostLe (ur : UCostRec)(erm : VaridMapMod.t EssoRaw)
  (uctprf : LTypePSMapEqSigT (sigifyMap (projT1 (ucRecTypes ur))
    (projT2 (ucRecTypes ur)))
  (VaridMapMod.map (fun e ⇒ (projT1 (getEssoRawTC e))) erm)) :=
  VaridMapMod.fold UDatMapFoldFunc (UCostRecSigifyMap ur erm uctprf) 0 ≤
    CT_PD_interp (ucRecTypes ur) (' (ucRecCost ur)) (ucRecDat ur)
    (“(ucRecCost ur)).

Definition MapCostLeTQuant(tr : sigT LRTypesPS)(cr : sig (CTDTP_REC tr))
  (erm : VaridMapMod.t EssoRaw) :=
  ∀ (us : sig (URecordP tr)),
    let ur := Build_UCostRec tr cr us in
      ∀ (uctprf : LTypePSMapEqSigT (sigifyMap (projT1 (ucRecTypes ur))
        (projT2 (ucRecTypes ur)))
      (VaridMapMod.map (fun e ⇒ (projT1 (getEssoRawTC e))) erm)),
      MapCostLe ur erm uctprf.

Let RssoI := Rsso ICostDT.

Inductive PattElsP(s s' : Size)(t : LTypePS s)(t' : LTypePS s') :
  list PattEl → Prop :=
| pattPBase : LTypePSEqHetEx s s' t t' → PattElsP s s' t t' nil
| pattPInd(l : list PattEl)(pe : PattElP s s' t t' l) : s > s' →
  PattElsP s s' t t' l

with PattElP(s s' : Size)(t : LTypePS s)(t' : LTypePS s') :
  list PattEl → Prop :=
| PattElVarid(v : Varid)
  (lrt : HBCL_0_1_Id_S.VaridMapMod.Raw.t (sig (LTypeRawCeiling s)))
  (l : list PattEl) : ∀ mok ltprf sfprf,
  ((∃ t″ : (sig (LTypeRawCeiling s)),
        (∃ mp : (VaridMapMod.MapsTo v t″ (VaridMapMod.Build_slist mok)),
          (existT s t) =t= buildProtoTFromSigTR
          (existT s (exist (LRTypesP s) lrt (RTypes s lrt mok sfprf ltprf))) ∧
          PattElsP (projT1 (‘t″)) s′ (sigifyLCeil2R s v t″
            (exist (LRTypesP s) lrt
              (RTypes s lrt mok sfprf ltprf)) mp) t′ l))) →
  PattElP s s′ t t′ (pattVarid v :: l)
| PattElPosParam(p : nat)(lts : LTypesPS s)(l : list PattEl) :
  (∃ t″ : (sig (LTypeRawCeiling s)),
    ∃ lp : p < length (‘lts),
      ∃ ncp : nth_certain (‘lts) p lp = t″,
        (existT s t) =t= buildProtoTFromSigTS (existT s lts) ∧
        PattElsP (projT1 (‘t″)) s′
        (sigifyLCeil s t″ lts (nth_certain_in lp ncp)) t′ l) →
  PattElP s s′ t t′ (pattPosParam p :: l).

Definition pattRel(pel : list PattEl)(t t′ : sigT LTypePS) :=
  PattElsP (projT1 t) (projT1 t′) (projT2 t) (projT2 t′) pel.

Inductive PattP(r : RssoI)(t : sigT LTypePS)(c : sig (CTDTP t)) :
  Patt → Prop :=
| PattP_intro(v : Varid)(pes : list PattEl) :
  (∃ s, ∃ t′, ∃ c′, VaridMapMod.MapsTo v
    (TssoGenDataT (existT (existT s t′) c′ ))
    r ∧ PattElsP s (projT1 t) t′ (projT2 t) pes ∧
    CT_PD_LT_HET (pattRel pes) ICostDT ICostDT c′ c) →
  PattP r t c (patt v pes).

Inductive EssoP(r : RssoI) : ∀ t (c : sig (CTDTP t)),
  EssoRaw → Prop :=
| essoConstrP(r′ : RssoI)(t : ProtoT)(c : sig (CTDTP t))
  (cr : CssoRaw) :
  r′ <r= r → CssoP r′ t c cr → EssoP r t c (essoConstr t c cr)
| essoPattP(r′ : RssoI)(p : Patt)(t : ProtoT)(c : sig (CTDTP t)) :
  r′ <r= r → PattP r′ t c p → EssoP r t c (essoPatt t c p)
| essoAppP(t : ProtoT)(c : sig (CTDTP t))
  (v : Varid)

  ( r″ : RssoI)
  (e′ : EssoRaw) :
  EssoP r″ (projT1 (getEssoRawTC e′)) (projT2 (getEssoRawTC e′)) e′
  →


  (∃ tso, ∃ tso′, ∃ varg, ∃ c″,
    VaridMapMod.MapsTo v tso r ∧ tso′ =TC= tso ∧
    CTCombAppPrf (projT2 (getEssoRawTC e′))
    c″ c ∧ tso′ = (TssoGenFunc varg (projT1 (getEssoRawTC e′))
      (existT (t) c″))) → r″ <r= r →
  EssoP r t c (essoApp t c e′ (‘ (‘ v)))

with CssoP(r : RssoI) : ∀ t (c : sig (CTDTP t)),
  CssoRaw → Prop :=
| cssoBaseP(lbt : LBasetype)(ubt : UBasetype lbt)
  (c : sig (CTDTP (BTBoolSPT lbt))) : CssoP r (BTBoolSPT lbt) c
  (cssoBase lbt ubt c)
| cssoTupleP (ts : sigT LTypesPS)(tc : sig (CTDTP_TUP ts))
  (c : sig (CTDTP (buildProtoTFromSigTS ts)))
  (les : list EssoRaw ) :
  CT_PD_LT_HET tListInT ICostDTupT ICostDT tc c →
  EssoTupP r ts tc les → CssoP r (buildProtoTFromSigTS ts) c
  (cssoTuple ts tc les )
| cssoRecordP (tr : sigT LRTypesPS)(tc : sig (CTDTP_REC tr))
  (c : sig (CTDTP (buildProtoTFromSigTR tr)))
  (er : VaridMapMod.t EssoRaw ) :
  CT_PD_LT_HET tMapInT ICostDRecT ICostDT tc c →
  EssoRecP r tr tc (VaridMapMod.this er) →
  CssoP r (buildProtoTFromSigTR tr) c
  (cssoRecord tr tc (VaridMapMod.this er
))

with EssoTupP(r : RssoI) :
  ∀ (ts : sigT LTypesPS)(ct : sig (CTDTP_TUP ts)),
  list EssoRaw → Prop :=
| EssoTupPIntro (ts : sigT LTypesPS)
  (cs : sig (CTDTP_TUP ts))
  (les : list EssoRaw ) :
  LTypePSListEqSigT
  (List.map (fun pr′ ⇒ (projT1 (getEssoRawTC pr′))) les)
  (sigifyList (projT1 ts) (projT2 ts)) →
  (∀ pr : EssoRaw, List.In pr les → EssoRawCeilingL ts cs pr ∧
    ∃ r′, r′ <r= r ∧ EssoP r′ (projT1 (getEssoRawTC pr))
      (projT2 (getEssoRawTC pr)) pr) →
  ListCostLeTQuant ts cs les
  → EssoTupP r ts cs les

with EssoRecP(r : RssoI) :
  ∀ (tr : sigT LRTypesPS)(cr : sig (CTDTP_REC tr)),
    VaridMapModRaw.t EssoRaw → Prop :=
| EssoRecPIntro(tr : sigT LRTypesPS)(cr : sig (CTDTP_REC tr))
  (er : VaridMapMod.t EssoRaw ) :
  ((∀ v, VaridMapMod.In v (LRTypesPSRecoverMap (projT1 tr) (projT2 tr))
    ↔
    VaridMapMod.In v er)) ∧
  (∀ v e, VaridMapMod.MapsTo v e er → ∃ r′,
    ∃ t, ∃ t′ : sig (LTypeRawCeiling (projT1 tr)),
      r′ <r= r ∧ t =t= (projT1 (getEssoRawTC e)) ∧
      (existT (projT1 t) (proj1_sig (projT2 t))) = (‘t′) ∧
      VaridMapMod.MapsTo v t′
      (LRTypesPSRecoverMap (projT1 tr) (projT2 tr))
      ∧ EssoRawCeilingR tr cr e ∧
      EssoP r′ (projT1 (getEssoRawTC e))
      (projT2 (getEssoRawTC e)) e
  ) → MapCostLeTQuant tr cr er →
  EssoRecP r tr cr (VaridMapMod.this er).
Definition OneStr : Cost := exist (fun n ⇒ (1 ≤ n)) 1 (le_refl 1).
Definition EssoPS(r : RssoI)(t : ProtoT) (c : sig (CTDTP t)) :=
  sig (EssoP r t c).
Definition EssoWFInR(r : RssoI)(e : sigTT EssoPS) := r <r= (projTT1 e).
Definition EssoMapWFInR(r : RssoI)(em : VaridMapMod.t (sigTT EssoPS)) :=
  ∀(v : Varid)(e : sigTT EssoPS), VaridMapMod.MapsTo v e em →
    (((projTT1 e) <r= r) ∧
      (∃ t, VaridMapMod.MapsTo v (TssoGenDataT t) r ∧
        (projT1 t) =t= (projTT2 e) ∧
        (existT (projTT2 e) (projTT3 e)) <tc= t )) ∨
    (∃ varg, ∃ t : ProtoT, ∃ t′,
      VaridMapMod.MapsTo v (TssoGenFunc varg t t′) r ∧
      (projTT1 e) <r= (VaridMapMod.add varg
        (TssoGenDataT (existT t (minC _))) r) ∧
      (projT1 t′) =t= (projTT2 e) ∧
      (existT (projTT2 e) (projTT3 e)) <tc= t′).
Lemma EssoERIncl : ∀ r r′ t c er, EssoP r t c er → r <r= r′ →
  EssoP r′ t c er.
Add Morphism (@EssoMapWFInR)
  with signature (R2ContainsR1 (ICostDT := ICostDT))
    ++> eq ++> impl as EssoMapRecRWContains.
Implicit Arguments UPot [T uraw u CTDT CTDTP cb].
Inductive FssoRaw : Type :=
| FssoVal(t : ProtoT) :
  UPot t (minC _) → FssoRaw
| FssoBFunc(v : Varid)(t t′ : ProtoT)(c : sig (CTDTP t′))
  (func : sig (UDataP t) → UPot t′ c) : FssoRaw
| FssoDat(r′ : RssoI)(t : ProtoT)(c : sig (CTDTP t))(e : EssoRaw) : FssoRaw
| FssoEFunc(r′ : RssoI)(v : Varid)(t t′ : ProtoT)(c : sig (CTDTP t′))
   (e : EssoRaw) : FssoRaw.
Inductive FssoP(r : RssoI) : TssoGen CTDT CTDTP → FssoRaw → Prop :=
| FssoValP(t : ProtoT)(up : UPot t (minC _)) :
   FssoP r (TssoGenDataT (existT t (minC _))) (FssoVal t up)
| FssoBFuncP(v : Varid)(t t′ : ProtoT)(c : sig (CTDTP t′))
   (func : sig (UDataP t) → UPot t′ c) :
   FssoP r (TssoGenFunc v t (existT t′ c)) (FssoBFunc v t t′ c func)
| FssoDatP(r′ : RssoI)(t : ProtoT)(c : sig (CTDTP t))(e : EssoRaw) :
   r′ <r= r → EssoP r′ t c e →
   FssoP r (TssoGenDataT (existT t c)) (FssoDat r′ t c e)
| FssoEFuncP(r′ : RssoI)(v : Varid)(t t′ : ProtoT)(c : sig (CTDTP t′))
   (e : EssoRaw) : r′ <r= r →
   EssoP (VaridMapMod.add v (TssoGenDataT (existT t (minC t))) r′) t′ c e →
   FssoP r (TssoGenFunc v t (existT t′ c)) (FssoEFunc r′ v t t′ c e).
Definition Fsso(r : RssoI)(T : TssoGen CTDT CTDTP) := sig (FssoP r T).
Definition FssoMapWFInR(r : RssoI)(em : VaridMapMod.t (sigTD Fsso)) :=
   ∀(v : Varid)(e : sigTD Fsso), VaridMapMod.MapsTo v e em →
      (((projTD1 e) <r= r) ∧
         (∃ T, VaridMapMod.MapsTo v T r ∧ (projTD2 e) <TC= T)).
Definition Wsso(r : RssoI) := sig (FssoMapWFInR r).
Definition updateRInWDisjPred(r r′ : RssoI)(w : Wsso r′) :=
   ∀ v : Varid, VaridMapMod.In v r → ¬ VaridMapMod.In v (‘w).
Lemma InWImpInR : ∀(r : RssoI)(w : Wsso r)(v : Varid),
   VaridMapMod.In v (‘w) → VaridMapMod.In v r.
Lemma R2ContainsR1Disj : ∀ r r′, VaridMapWPties.Disjoint r r′ →
   r <r= (VaridMapWPties.update r r′).
Add Morphism (@FssoMapWFInR) with signature
   (R2ContainsR1 (ICostDT := ICostDT)) ++> eq ++> impl
      as FssoMapRecRWContains.
Program Definition updateRInWDisj(r r′: RssoI)(w : Wsso r)
   (pf : VaridMapWPties.Disjoint r r′) :
   sig (updateRInWDisjPred r′ (VaridMapWPties.update r r′)) := w.
Obligation 1.
Obligation 2.
Definition WssoCplt(r : RssoI)(w : Wsso r) := ∀ v, VaridMapMod.In v r →
   VaridMapMod.In v (‘w).
Definition WssoCpltS(r : RssoI) := sig (WssoCplt r).
Definition WssoST := sigT WssoCpltS.
Definition WssoCpltSI := WssoCpltS.
Definition WssoCpltSIL := list (RssoI × sigT WssoCpltSI).
Definition emptyRsso : RssoI := VaridMapMod.empty (TssoGen CTDT CTDTP).
Inductive WssoCpltClos : RssoI → WssoCpltSIL → Prop :=
| WssoCWFPrBase(r : RssoI)(w : WssoCpltSI r) :
   WssoCpltClos r ((emptyRsso, existT r w) :: nil)
| WssoCWFPrInd(r r′ : RssoI)(w : WssoCpltSI r′)
   (wl : WssoCpltSIL) : WssoCpltClos r wl →
   WssoCpltClos (VaridMapWPties.update r r′) ((r, existT r′ w) :: wl).
Definition sigWssoClos(r : RssoI) := sig (WssoCpltClos r).
Definition buildUDataBase(lbt : LBasetype)(ubt : UBasetype lbt)
   (t : sigT LTypePS)(prf : LTypePSEq (projT1 t) 1 (projT2 t)
      (BuildBaseTypePS 1 lbt eq_refl)) : sig (UDataP t ) :=
   exist ((UDataP t )) (
      UBaseData lbt ubt) (UBaseDataP lbt t ubt prf).
Lemma ProtoTSImpl : ∀ t1 t2 : ProtoT, t1 =t= t2 → projT1 t2 = projT1 t1.
Implicit Arguments ProtoTSImpl [t1 t2].
Implicit Arguments UDataConvert [t1 t2].
Implicit Arguments uPPot [T uraw u CTDT CTDTP cb].
Implicit Arguments uPDat [T uraw u CTDT CTDTP cb].
Definition UPotConvert(t t′ : ProtoT)(c : sig (CTDTP t))(c′ : sig (CTDTP t′))
   (upin : UPot t c)(teq : t =t= t′)
```

```
  (cle : CT_PD_LE_HET ProtoEqTSigT ICostDT ICostDT c c′) : UPot t′ c′.
Implicit Arguments UPotConvert [t t′ c c′].
Definition UPotOptionConvert(t t′ : ProtoT)
  (c : sig (CTDTP t))(c′ : sig (CTDTP t′))
  (upin : option (UPot t c ))
  (teq : t =t= t′)(cle : CT_PD_LE_HET ProtoEqTSigT ICostDT ICostDT c c′) :
  option (UPot t′ c′ ) :=
  match upin with
    | Some upin ⇒ Some (UPotConvert upin teq cle)
    | None ⇒ None
  end.
Implicit Arguments UPotOptionConvert [t t′ c c′].
Lemma CssoPRIncl : ∀ r r′ t c cr, CssoP r t c cr → r <r= r′ →
  CssoP r′ t c cr.
Lemma EssoPImplCssoP : ∀ exprType′ exprCost′ er recLocal′ cstrct,
  er = essoConstr exprType′ exprCost′ cstrct →
  EssoP recLocal′ exprType′ exprCost′ er →
  CssoP recLocal′ exprType′ exprCost′ cstrct.
Section sigify2ExprSect.
  Variable r : RssoI.
  Variable etp : sigTT (EssoTupP r).
  Hypothesis erlprf : ∀ er, List.In er (projTT3 etp) →
    EssoP r (projT1 (getEssoRawTC er))
    (projT2 (getEssoRawTC er)) er.
  Hypothesis erlcprf : ∀ er, List.In er (projTT3 etp) →
    EssoRawCeilingL (projTT1 etp) (projTT2 etp) er.
  Let ts := projTT1 etp.
  Let tssf := (sigifyList (projT1 ts) (projT2 ts)).
  Let cs := projTT2 etp.
  Let erl : list EssoRaw := projTT3 etp.
  Hypothesis etpprf : LTypePSListEqSigT tssf
    (List.map (fun er ⇒ projT1 (getEssoRawTC er)) erl).
  Variable rett : ProtoT.
  Variable retc : sig (CTDTP rett).
  Let buildProtoTFromSTTS(ts′ : sigT LTypesPS) :=
    (existT (projT1 ts′)(buildLTypePSFromTS (projT1 ts′) (projT2 ts′))).
  Let LTypeB := fun ts ⇒
    existT (projT1 ts)(buildLTypePSFromTS (projT1 ts) (projT2 ts)).
  Let tsTConstrCmp(ts : sigT LTypesPS)(t : sigT LTypePS) :=
    ProtoEqTSigT (LTypeB ts) t.
  Hypothesis rettPrf : rett =t= buildProtoTFromSTTS ts.
  Hypothesis retcPrf : CT_PD_LT_HET tsTConstrCmp ICostDTupT ICostDT cs retc.
  Let ExprDQualP := sig2
    (EssoRawCeilingL ts cs)
    (fun erp : EssoRaw ⇒
      EssoP r (projT1 (getEssoRawTC erp)) (projT2 (getEssoRawTC erp)) erp).
  Section sig2ifyHelpersSect.
    Variable cl : list EssoRaw .
    Variable prf : incl cl erl.
    Variable e : EssoRaw .
    Variable es : list EssoRaw .
    Variable H : e :: es = cl.
    Lemma sig2ifyHelper1 :
      EssoP r (projT1 (getEssoRawTC e)) (projT2 (getEssoRawTC e)) e.
    Lemma sig2ifyHelper1a :
      EssoRawCeilingL (projTT1 etp) (projTT2 etp) e.
    Lemma sig2ifyHelper2 : incl es (projTT3 etp).
  End sig2ifyHelpersSect.
  Fixpoint sig2ifyExprListInner
    (cl : list EssoRaw )
    (prf : List.incl cl erl) {struct cl} :
```

```
    list ExprDQualP :=
    match cl as cl return cl = _ →
      list ExprDQualP with
      | nil ⇒ fun _ ⇒ nil
      | cons e es ⇒ fun H : (cons e es) = cl ⇒
        cons (exist2 e (sig2ifyHelper1a cl prf e es H)
          (sig2ifyHelper1 cl prf e es H))
        (sig2ifyExprListInner es (sig2ifyHelper2 cl prf e es H))
    end eq_refl.
  Definition sig2ifyExprList : list ExprDQualP :=
    sig2ifyExprListInner erl (List.incl_refl erl).
  Let exprSig2Proj(er : ExprDQualP) := exist (EssoRawCeilingL ts cs)
    (proj1_sig2 er) (proj2_sig2 er).
  Lemma sigifyExprInvInduc : ∀ (e : EssoRaw)
    (l es : list EssoRaw)
    (lin : incl l erl)(H : l = e :: es),
    es = map (proj1_sig2 (P := EssoRawCeilingL ts cs)
      (Q := fun er ⇒ EssoP r (projT1 (getEssoRawTC er))
        (projT2 (getEssoRawTC er)) er))
    (sig2ifyExprListInner es (sig2ifyHelper2 l lin e es (eq_sym H)))
    →
    l = map (proj1_sig2 (P := EssoRawCeilingL ts cs)
      (Q := fun er ⇒ EssoP r (projT1 (getEssoRawTC er))
        (projT2 (getEssoRawTC er)) er))
    (sig2ifyExprListInner l lin).
  Lemma sig2ifyProj :
    erl = List.map (proj1_sig2
      (P := EssoRawCeilingL (projTT1 etp) (projTT2 etp))
      (Q := fun er ⇒ EssoP r (projT1 (getEssoRawTC er))
        (projT2 (getEssoRawTC er)) er))
    sig2ifyExprList.
  Lemma sig2ifyLength :
    length (projTT3 etp) = length (List.map (proj1_sig2
      (P := EssoRawCeilingL (projTT1 etp) (projTT2 etp))
      (Q := fun er ⇒ EssoP r (projT1 (getEssoRawTC er))
        (projT2 (getEssoRawTC er)) er))
    sig2ifyExprList).
  Lemma typeExprLengthEq :
    length (sigifyList (projT1 ts) (projT2 ts)) = length sig2ifyExprList.
End sigify2ExprSect.
Section sigify2ExprRSect.
  Variable r : RssoI.
  Variable erp : sigTT (EssoRecP r).
  Hypothesis ermapok : SetoidList.NoDupA
    (@VaridMapMod.Raw.PX.eqk EssoRaw
      ) (projTT3 erp).
  Let erm := VaridMapMod.Build_slist ermapok.
  Hypothesis ermprf : ∀ v er, VaridMapMod.In v erm →
    VaridMapMod.MapsTo v er erm →
    EssoP r (projT1 (getEssoRawTC er)) (projT2 (getEssoRawTC er)) er.
  Hypothesis ermcprf : ∀ v er, VaridMapMod.In v erm →
    VaridMapMod.MapsTo v er erm →
    EssoRawCeilingR (projTT1 erp) (projTT2 erp) er.
  Let tr := projTT1 erp.
  Let trsf := (sigifyMap (projT1 tr) (projT2 tr)).
  Let cr := projTT2 erp.
  Hypothesis etpprf : (VaridMapMod.Equiv ProtoEqTSigT) trsf
    (VaridMapMod.map (fun er ⇒ projT1 (getEssoRawTC er)) erm).
  Variable rett : ProtoT.
  Variable retc : sig (CTDTP rett).
  Let buildProtoTFromSTTR(tr′ : sigT LRTypesPS) :=
    (existT (projT1 tr′)(buildLTypePSFromRT (projT1 tr′) (projT2 tr′))).
  Let LTypeB := fun tr ⇒
    existT (projT1 tr)(buildLTypePSFromRT (projT1 tr) (projT2 tr)).
  Let trTConstrCmp(tr : sigT LRTypesPS)(t : sigT LTypePS) :=
    ProtoEqTSigT (LTypeB tr) t.

  Hypothesis rettPrf : rett =t= buildProtoTFromSTTR tr.
  Hypothesis retcPrf : CT_PD_LT_HET trTConstrCmp
    ICostDRecT ICostDT cr retc.

  Let datListPotMax := CT_PD_max (CostBase := ICostDRecT) tr cr.
  Let retPotMax := datListPotMax + 1.

  Let ExprDQualP := sig2
    (EssoRawCeilingR tr cr) (fun er : EssoRaw ⇒
      EssoP r (projT1 (getEssoRawTC er)) (projT2 (getEssoRawTC er)) er).

  Section sig2ifyHelpersSect.
    Variable cl : list (Varid × EssoRaw).
    Variable prf : SetoidList.inclA (@VaridMapMod.eq_key_elt _) cl
      (VaridMapMod.elements (elt:=EssoRaw) erm).
    Variable e : Varid × EssoRaw.
    Variable es : list (Varid × EssoRaw).
    Variable H′ : e :: es = cl.

    Lemma sig2ifyFstEInErm : VaridMapMod.In (elt:=EssoRaw) (fst e) erm.

    Lemma sig2ifyEMapsToErm : VaridMapMod.MapsTo (fst e) (snd e) erm.

    Lemma sig2ifyMapHelper1 :
      EssoP r (projT1 (getEssoRawTC (snd e))) (projT2 (getEssoRawTC (snd e)))
      (snd e).

    Lemma sig2ifyMapHelper1a : EssoRawCeilingR tr cr (snd e).

    Lemma sig2ifyMapHelper2 : SetoidList.inclA (@VaridMapMod.eq_key_elt _) es
      (VaridMapMod.elements (elt:=EssoRaw) erm).
  End sig2ifyHelpersSect.

  Fixpoint sig2ifyExprMapInner
    (cl : list (Varid × EssoRaw))
    (prf : SetoidList.inclA (@VaridMapMod.eq_key_elt _ ) cl
      (VaridMapMod.elements erm)) { struct cl }:
    VaridMapMod.t ExprDQualP :=
    match cl as cl return cl = _ → VaridMapMod.t ExprDQualP with
      | nil ⇒ fun _ ⇒ (VaridMapMod.empty ExprDQualP)
      | e :: es ⇒ fun H′ : e :: es = cl ⇒
        VaridMapMod.add (fst e)
        ((exist2 ) (snd e)
        (sig2ifyMapHelper1a cl prf e es H′)
        (sig2ifyMapHelper1 cl prf e es H′))
        (sig2ifyExprMapInner es (sig2ifyMapHelper2 cl prf e es H′))
    end eq_refl.

  Lemma inclReflErmEls : SetoidList.inclA (@VaridMapMod.eq_key_elt _ )
    (VaridMapMod.elements erm)
    (VaridMapMod.elements erm).

  Definition sig2ifyExprMap : VaridMapMod.t ExprDQualP :=
    sig2ifyExprMapInner (VaridMapMod.elements erm) inclReflErmEls.

End sigify2ExprRSect.

Section uPotCostLeSect.

  Variable r : RssoI.
  Variable etp : sigTT (EssoTupP r).

  Hypothesis EssoTListConsist : LTypePSListEqSigT
    (sigifyList (projT1 (projTT1 etp)) (projT2 (projTT1 etp)))
    (map (fun e : EssoRaw ⇒ projT1 (getEssoRawTC e))
      (projTT3 etp)).

  Variable datList : list (sigTD (UPot (CTDT := CTDT) (CTDTP := CTDTP)
  (cb := ICostDT))).

  Let extractUDataRaw(up : (sigTD (UPot (CTDT := CTDT) (CTDTP := CTDTP)
    (cb := ICostDT)))) :=
    ‘ (uPDat (projTD1 up) (projTD2 up) (projTD3 up)).
  Let foldFuncPot(dat : (sigTD (UPot (CTDT := CTDT) (CTDTP := CTDTP)
    (cb := ICostDT))))(inpot : Potential) : Potential :=
    inpot + uPPot (projTD1 dat) (projTD2 dat) (projTD3 dat).
  Variable uTuplePSig : sig (UTupleP (projTT1 etp)).
  Definition UPotCostMap(up : (sigTD (UPot (CTDT := CTDT) (CTDTP := CTDTP)
    (cb := ICostDT)))) : UCost :=
    {| ucType := projTD1 up; ucCost := projTD2 up; ucDat :=
      uPDat (projTD1 up) (projTD2 up) (projTD3 up) |}.
  Hypothesis UCostPotEquivPot :
    fold_right UDatListFoldFunc 0 (map UPotCostMap datList) =
    fold_right UDatListFoldFunc 0
    (UCostTupSigifyList
      {|
        ucTupTypes := projTT1 etp;
        ucTupCost := projTT2 etp;
        ucTupDat := uTuplePSig |}
      (projTT3 etp) EssoTListConsist).
  Lemma uPotCostLe : fold_right foldFuncPot 0 datList ≤
    fold_right UDatListFoldFunc 0
    (UCostTupSigifyList
      {|
        ucTupTypes := projTT1 etp;
        ucTupCost := projTT2 etp;
        ucTupDat := uTuplePSig |}
      (projTT3 etp) EssoTListConsist).
End uPotCostLeSect.
Section UPotCostMapSect.
  Variable r : RssoI.
  Variable etp : sigTT (EssoTupP r).
  Let extractUDataRaw(up : (sigTD (UPot (CTDT := CTDT) (CTDTP := CTDTP)
    (cb := ICostDT)))) :=
  ‘ (uPDat (projTD1 up) (projTD2 up) (projTD3 up)).
  Lemma UCostTupSLAInnerNil : ∀ tcl inprf,
    UCostTupSigifyListAssistInner tcl nil inprf = nil.
  Lemma UCostTupSLAInnerNil2 : ∀ udl inprf,
    UCostTupSigifyListAssistInner nil udl inprf = nil.
  Let ExprDQualP := sig2 (EssoRawCeilingL (projTT1 etp) (projTT2 etp))
    (fun erp : EssoRaw ⇒
      EssoP r (projT1 (getEssoRawTC erp))
      (projT2 (getEssoRawTC erp)) erp).
  Let evalEls(x : sigT LTypesPS)(y : sig (CTDTP_TUP x))(redfunc : ∀
    (na : sig2 (EssoRawCeilingL x y)
      (fun erp : EssoRaw ⇒
        EssoP r (projT1 (getEssoRawTC erp)) (projT2 (getEssoRawTC erp))
        erp)), UPot (projT1 (getEssoRawTC (proj1_sig2 na)))
    (projT2 (getEssoRawTC (proj1_sig2 na)))) :=
  fun er ⇒
    existTD (projT1 (getEssoRawTC (proj1_sig2 er)))
    (projT2 (getEssoRawTC (proj1_sig2 er)))
    (redfunc er).
  Lemma UPotCostMapCorrect : ∀ erlprf erlcprf redfunc H H0,
    fold_right UDatListFoldFunc 0
    (map UPotCostMap (map (evalEls (projTT1 etp) (projTT2 etp) redfunc)
      (sig2ifyExprList r etp erlprf erlcprf))) =
    fold_right UDatListFoldFunc 0
    (UCostTupSigifyList
      {|
        ucTupTypes := projTT1 etp;
        ucTupCost := projTT2 etp;
        ucTupDat := exist (UTupleP (projTT1 etp))
        (map extractUDataRaw
          (map (evalEls (projTT1 etp) (projTT2 etp) redfunc)
            (sig2ifyExprList r etp erlprf erlcprf)))
          (UTuplePIntro (projTT1 etp)
            (map extractUDataRaw
              (map (evalEls (projTT1 etp) (projTT2 etp) redfunc)
                (sig2ifyExprList r etp erlprf erlcprf)))
            H) |} (projTT3 etp)
        H0).
  End UPotCostMapSect.
End EssoS.
Local Close Scope program_scope.
```

Listing D.30: The expression language large type

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Bool.Bool.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.

Require Import HBCL.HBCL_0_1.FuncLibs.UCost.UCostFacts.
Module UCostFactsSB := UCostFacts HBCL_0_1_L_UTS HBCL_0_1_L_UTSCost.

Export UCostFactsSB.
Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.

Local Open Scope program_scope.

Module VaridMapWFacts :=
  FMapFacts.WFacts_fun varidPred.PredidDecidable VaridMapMod.
Module VaridMapWPties :=
  FMapFacts.WProperties_fun varidPred.PredidDecidable VaridMapMod.

Notation varidEqb := varidPred.PredidDecidable.eqb.
Infix "=v=" := varidPred.PredidDecidable.eq (at level 70, no associativity).
Infix "=t=" := ProtoEqTSigT (at level 70, no associativity).

Section TssoS.

  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).

  Context ‘{ICostDT : CostDT CTDT CTDTP}.
  Context ‘{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context ‘{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.

  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.

  Variable minC : ∀ t : ProtoT, sig (CTDTP t).
  Hypothesis minCMin : ∀ t c u, CT_PD_interp(CostBase := ICostDT)
    t (‘c) u (“c) = 0.
```

```
Lemma equivEq : ∀(ptm ptm′ : VaridMapMod.t ProtoT),
  VaridMapMod.equal ProtoEqbTSigT ptm ptm′ = true →
  VaridMapMod.Equiv ProtoEqTSigT ptm ptm′.
Implicit Arguments equivEq [ptm ptm′].
Implicit Arguments ProtoTSigT_eqb_eq [t1 t2].

  Definition BTBoolSPTC(lbt : LBasetype) : {t : ProtoT & sig (CTDTP t)} :=
    existT (BTBoolSPT lbt) (minC (BTBoolSPT lbt)).

  Definition buildProtoTFromTSC
    (ts : {ts : sigT LTypesPS & sig (CTDTP_TUP ts)}) :=
    (existT (projT1 (projT1 ts)) (buildLTypePSFromTS (projT1 (projT1 ts))
        (projT2 (projT1 ts)))).

  Definition buildProtoT(T T′ : Size → Type)
   (CTDT′ : Type)(CTDTP′ : (sigT T) → CTDT′ → Prop)
   (func : (∀ s : Size, T s → T′ s))
   (arg : {arg : sigT T & sig (CTDTP′ arg)}) :=
   (existT (projT1 (projT1 arg))
     (func (projT1 (projT1 arg))
       (projT2 (projT1 arg)))).
  Implicit Arguments buildProtoT [T T′ CTDT′ CTDTP′].

  Definition buildProtoTSigT(T T′ : Size → Type)
   (CTDT′ : Type)(CTDTP′ : (sigT T) → CTDT′ → Prop)
   (func : sigT T → sigT T′)
   (arg : {arg : sigT T & sig (CTDTP′ arg)}) := func (projT1 arg).

  Definition buildProtoTFromRTC
    (tr : {tr : sigT LRTypesPS & sig (CTDTP_REC tr)}) :=
    (existT (projT1 (projT1 tr))
      (buildLTypePSFromRT (projT1 (projT1 tr))
        (projT2 (projT1 tr)))).

  Definition BuildLTypePSTC(ts : {ts : sigT LTypesPS & sig (CTDTP_TUP ts)})
    (c : sig (CTDTP (buildProtoTFromTSC ts))): {t : ProtoT & sig (CTDTP t)} :=
    existT (buildProtoTFromTSC ts) c.

  Definition BuildLTypePSRC(tr : {tr : sigT LRTypesPS & sig (CTDTP_REC tr)})
    (c : sig (CTDTP (buildProtoTFromRTC tr))): {t : ProtoT & sig (CTDTP t)} :=
    existT (buildProtoTFromRTC tr) c.

   Inductive TssoGen : Type :=
  | TssoGenDataT : {t : ProtoT & sig (CTDTP t)} → TssoGen
  | TssoGenFunc : Varid → ProtoT → {t : ProtoT & sig (CTDTP t)} →
    TssoGen.

 Definition TssoGenStructEqb
    (t1 t2 : TssoGen ) : bool :=
    match t1, t2 with
      | TssoGenDataT l, TssoGenDataT m ⇒
        CT_PD_T_eqb_struct l m
      | TssoGenFunc v i o, TssoGenFunc v′ i′ o′ ⇒ varidEqb v v′ &&
        ProtoEqbTSigT i i′ && ProtoEqbTSigT (projT1 o) (projT1 o′) &&
      CT_PD_T_eqb_struct o o′
      | _, _ ⇒ false
    end.

 Definition TssoGenStructEq (t1 t2 : TssoGen) : Prop :=
  match t1, t2 with
    | TssoGenDataT l, TssoGenDataT m ⇒ CT_PD_T_eq_struct l m
    | TssoGenFunc v i o, TssoGenFunc v′ i′ o′ ⇒
      v =v= v′ ∧ ProtoEqTSigT i i′ ∧ ProtoEqTSigT (projT1 o) (projT1 o′) ∧
      CT_PD_T_eq_struct o o′
    | _, _ ⇒ False
  end.

 Infix ”=TS=” := TssoGenStructEq (at level 70, no associativity).

 Lemma TssoGenStructEqbIff : ∀(t1 t2 : TssoGen),
    TssoGenStructEqb t1 t2 = true ↔ TssoGenStructEq t1 t2.

  Lemma TssoGen_struct_eq_refl : ∀ x : TssoGen, TssoGenStructEq x x.

  Lemma TssoGen_struct_eq_sym :
    ∀ x y : TssoGen, TssoGenStructEq x y →
```

```
        TssoGenStructEq y x.

  Theorem TssoGen_struct_eq_trans: ∀ x y z : TssoGen,
     TssoGenStructEq x y → TssoGenStructEq y z → TssoGenStructEq x z.

  Add Relation (TssoGen) (TssoGenStructEq)
  reflexivity proved by (@TssoGen_struct_eq_refl)
  symmetry proved by (@TssoGen_struct_eq_sym)
     transitivity proved by (@TssoGen_struct_eq_trans)
        as TssoGen_struct_eq_rel.

Definition TssoGenCostEqb
     (t1 t2 : TssoGen ) : bool :=
     match t1, t2 with
        | TssoGenDataT l, TssoGenDataT m ⇒
           CT_PD_T_eqb_pot l m
        | TssoGenFunc v i o, TssoGenFunc v' i' o' ⇒ varidEqb v v' &&
           ProtoEqbTSigT i i' && ProtoEqbTSigT (projT1 o) (projT1 o') &&
        CT_PD_T_eqb_pot o o'
        | _, _ ⇒ false
     end.

Definition TssoGenCostEq (t1 t2 : TssoGen) : Prop :=
  match t1, t2 with
     | TssoGenDataT l, TssoGenDataT m ⇒ CT_PD_T_eq_pot l m
     | TssoGenFunc v i o, TssoGenFunc v' i' o' ⇒
        v =v= v' ∧ ProtoEqTSigT i i' ∧ ProtoEqTSigT (projT1 o) (projT1 o') ∧
        CT_PD_T_eq_pot o o'
     | _, _ ⇒ False
  end.

Infix "=TC=" := TssoGenCostEq (at level 70, no associativity).

Lemma TssoGenCostEqbIff : ∀(t1 t2 : TssoGen),
     TssoGenCostEqb t1 t2 = true ↔ t1 =TC= t2.

  Lemma TssoGen_cost_eq_refl : ∀ x : TssoGen, x =TC= x.

  Lemma TssoGen_cost_eq_sym :
     ∀ x y : TssoGen, x =TC= y → y =TC= x.

  Theorem TssoGen_cost_eq_trans: ∀ x y z : TssoGen,
     TssoGenCostEq x y → TssoGenCostEq y z → TssoGenCostEq x z.

  Add Relation (TssoGen) (TssoGenCostEq)
  reflexivity proved by (@TssoGen_cost_eq_refl)
  symmetry proved by (@TssoGen_cost_eq_sym)
     transitivity proved by (@TssoGen_cost_eq_trans)
        as TssoGen_cost_eq_rel.

  Definition TssoGenCostLeb
     (t1 t2 : TssoGen ) : bool :=
     match t1, t2 with
        | TssoGenDataT l, TssoGenDataT m ⇒
           CT_PD_T_leb_pot l m
        | TssoGenFunc v i o, TssoGenFunc v' i' o' ⇒ varidEqb v v' &&
           ProtoEqbTSigT i i' && ProtoEqbTSigT (projT1 o) (projT1 o') &&
        CT_PD_T_leb_pot o o'
        | _, _ ⇒ false
     end.

  Definition TssoGenCostLe (t1 t2 : TssoGen) : Prop :=
     match t1, t2 with
        | TssoGenDataT l, TssoGenDataT m ⇒ CT_PD_T_le_pot l m
        | TssoGenFunc v i o, TssoGenFunc v' i' o' ⇒
           v =v= v' ∧ ProtoEqTSigT i i' ∧ ProtoEqTSigT (projT1 o) (projT1 o') ∧
           CT_PD_T_le_pot o o'
        | _, _ ⇒ False
     end.

  Definition TssoGenCostLt (t1 t2 : TssoGen) : Prop :=
     match t1, t2 with
        | TssoGenDataT l, TssoGenDataT m ⇒ CT_PD_T_lt_pot l m
        | TssoGenFunc v i o, TssoGenFunc v' i' o' ⇒
           v =v= v' ∧ ProtoEqTSigT i i' ∧ ProtoEqTSigT (projT1 o) (projT1 o') ∧
           CT_PD_T_lt_pot o o'
```

```
      | _, _ ⇒ False
    end.
 Infix "<TC=" := TssoGenCostLe (at level 70, no associativity).
 Infix "<TC<" := TssoGenCostLt (at level 70, no associativity).
 Lemma TssoGenCostLebIff : ∀(t1 t2 : TssoGen),
    TssoGenCostLeb t1 t2 = true ↔ t1 <TC= t2.
  Lemma TssoGen_Cost_le_refl : ∀ x : TssoGen, x <TC= x.
  Theorem TssoGen_Cost_le_trans: ∀ x y z : TssoGen,
    x <TC= y → y <TC= z → x <TC= z.
  Theorem TssoGen_Cost_le_antisym : ∀ {x y},
    x <TC= y → y <TC= x → x =TC= y.
End TssoS.
Implicit Arguments TssoGenCostEq [CTDT CTDTP ICostDT].
Definition vmmE(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :=
  VaridMapMod.Equiv (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP)
  (ICostDT := ICostDT)).
Implicit Arguments vmmE [CTDT CTDTP ICostDT].
Infix "=r=" := vmmE (at level 70, no associativity).
Implicit Arguments TssoGenDataT [CTDT CTDTP].
Implicit Arguments TssoGenFunc [CTDT CTDTP].
Implicit Arguments TssoGenCostLe [CTDT CTDTP ICostDT].
Implicit Arguments TssoGenCostLt [CTDT CTDTP ICostDT].
Infix "=TC=" := TssoGenCostEq (at level 70, no associativity).
Infix "<TC=" := TssoGenCostLe (at level 70, no associativity).
Infix "<TC<" := TssoGenCostLt (at level 70, no associativity).
Add Parametric Relation (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : (TssoGen CTDT CTDTP)
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT))
  reflexivity proved by (@TssoGen_cost_eq_refl CTDT CTDTP ICostDT)
  symmetry proved by (@TssoGen_cost_eq_sym CTDT CTDTP ICostDT)
  transitivity proved by (@TssoGen_cost_eq_trans CTDT CTDTP ICostDT)
    as TssoGenParam_cost_eq_rel.
Check TssoGenParam_cost_eq_rel.
Add Parametric Relation (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : (TssoGen CTDT CTDTP)
  (TssoGenStructEq CTDT CTDTP)
  reflexivity proved by (@TssoGen_struct_eq_refl CTDT CTDTP ICostDT)
  symmetry proved by (@TssoGen_struct_eq_sym CTDT CTDTP ICostDT)
  transitivity proved by (@TssoGen_struct_eq_trans CTDT CTDTP ICostDT)
    as TssoGenParam_eq_rel.
  Instance TssoGen_ReflI (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
    (ICostDT : CostDT CTDT CTDTP) :
    Reflexive (TssoGenCostLe (CTDT := CTDT) (CTDTP := CTDTP)
      (ICostDT := ICostDT)) :=
    { reflexivity := TssoGen_Cost_le_refl CTDT CTDTP}.
  Instance TssoGen_TransI(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
    (ICostDT : CostDT CTDT CTDTP) :
    Transitive (TssoGenCostLe (CTDT := CTDT) (CTDTP := CTDTP)
      (ICostDT := ICostDT)):=
    { transitivity := TssoGen_Cost_le_trans CTDT CTDTP}.
  Instance TssoGenCostPreOrder(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
    (ICostDT : CostDT CTDT CTDTP) :
    PreOrder (TssoGenCostLe (CTDT := CTDT) (CTDTP := CTDTP)
      (ICostDT := ICostDT)) := {
    PreOrder_Reflexive := TssoGen_ReflI CTDT CTDTP ICostDT;
    PreOrder_Transitive := TssoGen_TransI CTDT CTDTP ICostDT
  }.
  Instance TssoGenCostPOI(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
    (ICostDT : CostDT CTDT CTDTP)
    : PartialOrder (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP)
      (ICostDT := ICostDT))
  (TssoGenCostLe (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)).


Definition Rsso(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :=
  (VaridMapMod.t (TssoGen CTDT CTDTP)).
Implicit Arguments Rsso [CTDT CTDTP].
Definition R2ContainsR1(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP)(r1 r2 : Rsso ICostDT) : Prop :=
  (∀ k, VaridMapMod.In k r1 → VaridMapMod.In k r2) ∧
  (∀ k t u, VaridMapMod.MapsTo k t r1 → VaridMapMod.MapsTo k u r2 →
    t =TC= u).
Implicit Arguments R2ContainsR1 [CTDT CTDTP ICostDT].
Infix "<r=" := R2ContainsR1 (at level 70, no associativity).
Theorem Rsso_equiv_refl(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP)
:
∀ x : Rsso ICostDT, VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)) x x.
Theorem Rsso_equiv_sym(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP) :
∀ x y : (Rsso ICostDT), VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)) x y
  → VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)) y x.
Theorem Rsso_equiv_trans(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP): ∀ x y z :
  (Rsso ICostDT),
  VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)) x y →
  VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)) y z →
  VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)) x z.
Add Parametric Relation (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP) : (Rsso ICostDT)
(VaridMapMod.Equiv
  (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)))
reflexivity proved by (@Rsso_equiv_refl CTDT CTDTP ICostDT)
symmetry proved by (@Rsso_equiv_sym CTDT CTDTP ICostDT)
  transitivity proved by (@Rsso_equiv_trans CTDT CTDTP ICostDT)
    as Rsso_vmmE_param_rel.
Definition vmmEqb(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP) :=
VaridMapMod.equal (TssoGenCostEqb CTDT CTDTP).
Implicit Arguments vmmEqb [CTDT CTDTP ICostDT].
Lemma vmmEquivEqb(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP) : ∀ m1 m2 :
  (VaridMapMod.t (TssoGen CTDT CTDTP)),
  vmmEqb m1 m2 = true ↔ m1 =r= m2.
Add Parametric Morphism (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP) : (@VaridMapMod.In (TssoGen CTDT CTDTP))
  with signature varidPred.PredidDecidable.eq ==>
    (VaridMapMod.Equiv (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP)
    (ICostDT := ICostDT))) ==> iff as In_RssoEquiv.
Add Parametric Morphism (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
(ICostDT : CostDT CTDT CTDTP) : (@vmmE CTDT CTDTP ICostDT)
  with signature (VaridMapMod.Equiv
    (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT))) ==>
    (VaridMapMod.Equiv
      (TssoGenCostEq (CTDT := CTDT) (CTDTP := CTDTP) (ICostDT := ICostDT)))
    ==> iff as RssoEquivEquiv.
Instance R2ContainsR1SubRel(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
```

```
  (ICostDT : CostDT CTDT CTDTP) :
    subrelation (@vmmE CTDT CTDTP ICostDT) (@R2ContainsR1 CTDT CTDTP ICostDT).
  Theorem R2Contains_refl(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : ∀ x : Rsso ICostDT, x <r= x.
  Theorem R2Contains_trans(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP): ∀ x y z : Rsso ICostDT,
    x <r= y → y <r= z → x <r= z.
  Theorem R2Contains_antisym(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : ∀ {x y}, x <r= y → y <r= x → x =r= y.
  Instance R2ContainsReflI(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  Reflexive (R2ContainsR1 (ICostDT := ICostDT)) :=
  { reflexivity := R2Contains_refl CTDT CTDTP ICostDT}.
  Instance R2ContainsTransI(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
    Transitive (R2ContainsR1 (ICostDT := ICostDT)) :=
    { transitivity := R2Contains_trans CTDT CTDTP ICostDT}.
  Instance R2ContainsPreOrder(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  PreOrder (R2ContainsR1 (ICostDT := ICostDT)) := {
    PreOrder_Reflexive := R2ContainsReflI CTDT CTDTP ICostDT;
    PreOrder_Transitive := R2ContainsTransI CTDT CTDTP ICostDT
  }.
  Instance R2ContainsPOI(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  PartialOrder (vmmE (ICostDT := ICostDT)) (R2ContainsR1 (ICostDT := ICostDT)).
  Add Parametric Morphism (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : (R2ContainsR1 (ICostDT := ICostDT))
    with signature eq ==>
      (VaridMapMod.Equiv (TssoGenCostEq (ICostDT := ICostDT))) ==>
      iff as R2ContainsR1_RssoEqToEqEquiv.
  Add Parametric Morphism(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : (R2ContainsR1 (ICostDT := ICostDT))
    with signature
      (VaridMapMod.Equiv (TssoGenCostEq (ICostDT := ICostDT))) ==> eq ==>
      iff as R2ContainsR1_RssoEqEquivToEq.
  Add Parametric Morphism(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : (R2ContainsR1 (ICostDT := ICostDT))
    with signature
      (VaridMapMod.Equiv (TssoGenCostEq (ICostDT := ICostDT)))
      ==> (VaridMapMod.Equiv (TssoGenCostEq (ICostDT := ICostDT))) ==>
      iff as R2ContainsR1_RssoEquiv.
  Add Parametric Morphism(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  (VaridMapMod.In (elt := (TssoGen CTDT CTDTP)))
    with signature varidPred.PredidDecidable.eq ++>
      (R2ContainsR1 (ICostDT := ICostDT)) ++> impl as In_RssoInContains.
Lemma R2C1MapsToExistTsso(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : ∀ v t r r′, VaridMapMod.MapsTo v t r →
  r <r= r′ → (∃ t′, VaridMapMod.MapsTo v t′ r′ ∧ t =TC= t′).
Implicit Arguments R2C1MapsToExistTsso [CTDT CTDTP ICostDT].
Lemma R2C1MapsToExistProtoT(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : ∀ v t r r′,
  VaridMapMod.MapsTo v (TssoGenDataT t) r → r <r= r′ →
  (∃ t′, VaridMapMod.MapsTo v (TssoGenDataT t′) r′ ∧ t =tc= t′).
Implicit Arguments R2C1MapsToExistProtoT [CTDT CTDTP ICostDT].
Lemma R2C1MapsToExistFunc(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : ∀ v v′ t1 t2 r r′,
  VaridMapMod.MapsTo v (TssoGenFunc v′ t1 t2) r → r <r= r′ →
  (∃ v″, ∃ t1′, ∃ t2′,
    VaridMapMod.MapsTo v (TssoGenFunc v″ t1′ t2′) r′ ∧
    v′ =v= v″ ∧ t1′ =t= t1 ∧ t2 =tc= t2′).
Implicit Arguments R2C1MapsToExistFunc [CTDT CTDTP ICostDT].
```

```
Add Parametric Morphism (CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  (VaridMapMod.add (elt := TssoGen CTDT CTDTP))
  with signature varidPred.PredidDecidable.eq ++>
    (TssoGenCostEq (ICostDT := ICostDT)) ++>
    (R2ContainsR1 (ICostDT := ICostDT)) ++>
    (R2ContainsR1 (ICostDT := ICostDT)) as R2ContainsUnderMAdd.

Lemma AddRssoEquivR2C(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  ∀ (t t′ : TssoGen CTDT CTDTP)(v v′ : Varid)(r : Rsso ICostDT),
  v =v= v′ → t =TC= t′ →
  (VaridMapMod.add v t r) <r= (VaridMapMod.add v′ t′ r).
Implicit Arguments AddRssoEquivR2C [CTDT CTDTP ICostDT].

Lemma AddRssoEquivEq(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  ∀ (t t′ : TssoGen CTDT CTDTP)(v v′ : Varid)(r : Rsso ICostDT),
  v =v= v′ → t =TC= t′ →
  (VaridMapMod.add v t r) =r= (VaridMapMod.add v′ t′ r).
Implicit Arguments AddRssoEquivEq [CTDT CTDTP ICostDT].

Lemma TssoLeInjDat(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  ∀ t t′, t <tc= t′ → (TssoGenDataT t) <TC=
  (TssoGenDataT t′).

Lemma TssoEqInjDat(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) :
  ∀ t t′, t =tc= t′ → (TssoGenDataT t) =TC=
  (TssoGenDataT t′).

Lemma updateROvwrt(CTDT : Type)(CTDTP : ProtoT → CTDT → Prop)
  (ICostDT : CostDT CTDT CTDTP) : ∀ (r r′ : Rsso ICostDT),
    r <r= (VaridMapWPties.update r′ r).
```

Listing D.31: The expression language built-in types

```
Require Export HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Export HBCL_0_1_L_UTS.
Require Import Program.

Definition BaseTypeBool :=
  BuildBaseTypePS 1 HBCL_0_1_L_UTS.BasetypeBool eq_refl.

Definition BaseTypeBoolRaw : sigT LTypeRaw :=
  existT _ (‘BaseTypeBool).

Definition pairListRaw :=
  (BaseTypeBoolRaw :: BaseTypeBoolRaw :: nil)%list.

Definition pairRawTup := LTupleType 2 pairListRaw.

Program Definition pairTupStrong : LTypesPS 2 :=
  (BaseTypeBoolRaw :: BaseTypeBoolRaw :: nil)%list.
Obligation 1.
Admitted.
Obligation 2.
Admitted.
Obligation 3.
Admitted.

Program Definition bitPair : sig (LTypeP 2) := pairRawTup.
Obligation 1.
Admitted.
```

Listing D.32: The expression language built-in functions

```
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UBoxEmptyEnc_SB.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UTypeSysOid_SB.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.exprLang.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.builtInTypes.

Import sigTypes.
Import String.

Definition emptyRsso : TypeSSO.Rsso instBTSCostBase :=
  HBCL_0_1_Id_S.VaridMapMod.empty (TypeSSO.TssoGen CTDTtriv CTDTPtriv).

Program Definition notId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  ("not")%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

Program Definition BoolNotCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang (existT 1 BaseTypeBool)) :=
  2.
Obligation 1.
Admitted.

Definition notRetType :
  {t : HBCL_0_1_L_UTS.ProtoT & sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)}
  := existT (existT 1 BaseTypeBool) BoolNotCost.

Definition notFuncTsso := TypeSSO.TssoGenFunc

  notId (existT 1 BaseTypeBool) notRetType.

Definition RssoWithNotFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add notId notFuncTsso emptyRsso.

Section notFuncS.

  Variable inDat : sig (HBCL_0_1_L_UTS.UDataP (existT 1 BaseTypeBool)).

  Definition notFunc :
    HBCL_0_1_L_UTSCost.UPot (existT 1 BaseTypeBool) BoolNotCost.
  Defined.

End notFuncS.

Definition notFuncFssoRaw :=
  ExprSSO.FssoBFunc _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  minCtriv notId (existT 1 BaseTypeBool) (existT 1 BaseTypeBool)
  BoolNotCost notFunc.

Program Definition notFuncFsso :=
  sigTypes.existTD (P := ExprSSO.Fsso _ _ _ _ _ _ _) RssoWithNotFunc
  notFuncTsso
  (exist (ExprSSO.FssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    minCtriv RssoWithNotFunc notFuncTsso) notFuncFssoRaw _).
Obligation 1.
Admitted.

Definition emptyFssoMap := HBCL_0_1_Id_S.VaridMapMod.empty
  (sigTypes.sigTD (ExprSSO.Fsso _ _ _ _ _ _ minCtriv
    (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord))).

Definition FssoMapNotFunc := HBCL_0_1_Id_S.VaridMapMod.add notId notFuncFsso
  emptyFssoMap.

Program Definition negFuncWFInR : sig
  (ExprSSO.FssoMapWFInR _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv RssoWithNotFunc) := FssoMapNotFunc.
Obligation 1.
Admitted.

Program Definition xorId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  ("xor")%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

Program Definition BoolXorCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang (existT _ BaseTypeBool)) :=
  2.
Obligation 1.
Admitted.

Definition xorRetType :
  {t : HBCL_0_1_L_UTS.ProtoT & sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)}
  := existT (existT 1 BaseTypeBool) BoolXorCost.

Print TypeSSO.TssoGenFunc.

Definition xorFuncTsso := TypeSSO.TssoGenFunc
  xorId (existT _ bitPair) xorRetType.

Definition RssoWithXorFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add xorId xorFuncTsso emptyRsso.

Definition RssoWithNotAndXorFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add xorId xorFuncTsso RssoWithNotFunc.

Section xorFuncS.

  Variable inDat : sig (HBCL_0_1_L_UTS.UDataP (existT _ bitPair)).

  Definition xorFunc : HBCL_0_1_L_UTSCost.UPot
    (existT _ BaseTypeBool) BoolXorCost.
  Defined.

End xorFuncS.

Definition xorFuncFssoRaw :=
  ExprSSO.FssoBFunc _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  minCtriv xorId (existT _ bitPair) (existT 1 BaseTypeBool)
  BoolXorCost xorFunc.

Program Definition xorFuncFsso :=
  sigTypes.existTD (P := ExprSSO.Fsso _ _ _ _ _ _ _) RssoWithXorFunc
  xorFuncTsso
  (exist (ExprSSO.FssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    minCtriv RssoWithXorFunc xorFuncTsso) xorFuncFssoRaw _).
Obligation 1.
Admitted.

Definition FssoMapNotAndXorFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add xorId xorFuncFsso FssoMapNotFunc.

Program Definition negAndXorFuncWFInR : sig
  (ExprSSO.FssoMapWFInR _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv RssoWithNotAndXorFunc) := FssoMapNotAndXorFunc.
Obligation 1.
Admitted.

Program Definition orId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  ("or")%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

Program Definition BoolOrCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang (existT _ BaseTypeBool)) :=
  2.
Obligation 1.
Admitted.

Definition orRetType :
  {t : HBCL_0_1_L_UTS.ProtoT & sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)}
  := existT (existT 1 BaseTypeBool) BoolOrCost.
```

```
Print TypeSSO.TssoGenFunc.
Definition orFuncTsso := TypeSSO.TssoGenFunc
  orId (existT _ bitPair) orRetType.

Definition RssoWithOrFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add orId orFuncTsso emptyRsso.
Definition cumRssoWithOrFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add orId orFuncTsso RssoWithNotAndXorFunc.

Section orFuncS.
  Variable inDat : sig (HBCL_0_1_L_UTS.UDataP (existT _ bitPair)).
  Definition orFunc : HBCL_0_1_L_UTSCost.UPot
    (existT _ BaseTypeBool) BoolOrCost.
  Defined.
End orFuncS.
Definition orFuncFssoRaw :=
  ExprSSO.FssoBFunc _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  minCtriv orId (existT _ bitPair) (existT 1 BaseTypeBool)
  BoolOrCost orFunc.
Program Definition orFuncFsso :=
  sigTypes.existTD (P := ExprSSO.Fsso _ _ _ _ _ _ _) RssoWithOrFunc
  orFuncTsso
  (exist (ExprSSO.FssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    minCtriv RssoWithOrFunc orFuncTsso) orFuncFssoRaw _).
Obligation 1.
Admitted.
Definition cumFssoMapWithOrFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add orId orFuncFsso FssoMapNotAndXorFunc.
Program Definition cumFuncOrFuncWFInR : sig
  (ExprSSO.FssoMapWFInR _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv cumRssoWithOrFunc) := cumFssoMapWithOrFunc.
Obligation 1.
Admitted.

Program Definition andId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  ("and")%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.
Program Definition BoolAndCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang (existT _ BaseTypeBool)) :=
  2.
Obligation 1.
Admitted.
Definition andRetType :
  {t : HBCL_0_1_L_UTS.ProtoT & sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)}
  := existT (existT 1 BaseTypeBool) BoolAndCost.
Print TypeSSO.TssoGenFunc.
Definition andFuncTsso := TypeSSO.TssoGenFunc
  andId (existT _ bitPair) andRetType.

Definition RssoWithAndFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add andId andFuncTsso emptyRsso.
Definition cumRssoWithAndFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add andId andFuncTsso cumRssoWithOrFunc.

Section andFuncS.
  Variable inDat : sig (HBCL_0_1_L_UTS.UDataP (existT _ bitPair)).
  Definition andFunc : HBCL_0_1_L_UTSCost.UPot
    (existT _ BaseTypeBool) BoolAndCost.
  Defined.
```

```
End andFuncS.

Definition andFuncFssoRaw :=
  ExprSSO.FssoBFunc _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  minCtriv andId (existT _ bitPair) (existT 1 BaseTypeBool)
  BoolAndCost andFunc.

Program Definition andFuncFsso :=
  sigTypes.existTD (P := ExprSSO.Fsso _ _ _ _ _ _ _) RssoWithAndFunc
  andFuncTsso
  (exist (ExprSSO.FssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    minCtriv RssoWithAndFunc andFuncTsso) andFuncFssoRaw _).
Obligation 1.
Admitted.

Definition cumFssoMapWithAndFunc :=
  HBCL_0_1_Id_S.VaridMapMod.add andId andFuncFsso cumFssoMapWithOrFunc.

Program Definition cumFuncAndFuncWFInR : sig
  (ExprSSO.FssoMapWFInR _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv cumRssoWithAndFunc) := cumFssoMapWithAndFunc.
Obligation 1.
Admitted.

Definition builtinsRsso := cumRssoWithAndFunc.

Program Definition builtinsWC :
  ExprSSO.sigWssoClos _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv builtinsRsso :=
  ((builtinsRsso, existT builtinsRsso cumFuncAndFuncWFInR)
    :: nil)%list.
Obligation 1.
Admitted.
Obligation 2.
Admitted.
Obligation 3.
Admitted.

Program Definition mainId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  ("main")%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

Program Definition mainArgId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  ("env")%string.
Obligation 1.
Admitted.

Obligation 2.
Admitted.
```

D.6.1 Expression reduction

Listing D.33: The expression language reduction function: pattern branch

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Arith.Lt.
Require Import Coq.Arith.Wf_nat.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Wellfounded.Inverse_Image.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.ExprSSO.

Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.

Local Open Scope program_scope.

Section UPattMakeSect.

  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).

  Context ‘{ICostDT : CostDT CTDT CTDTP}.
  Context ‘{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context ‘{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.

  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.

  Variable minC : ∀ t : ProtoT, sig (CTDTP t).
  Hypothesis minCMin : ∀ t u, CT_PD_interp(CostBase := ICostDT)
    t (‘ (minC t)) u (“ (minC t)) = 0.

  Let RssoI := Rsso ICostDT.
  Let EssoPI := EssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let EssoTupPI := EssoTupP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let EssoRawI := EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let UPotI := UPot CTDT CTDTP ICostDT.

  Let EssoRawCeilingT(r : RssoI)(t : sigT LTypePS)(c : sig (CTDTP t))
    (t′ : sigT LTypePS)(c′ : sig (CTDTP t′))(ep : sig (EssoPI r t′ c′)) :=
    existT t′ c′ <tc< existT t c.

  Implicit Arguments sigWssoClos [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDTupT ICostDRecT].
  Implicit Arguments emptyRsso [CTDT CTDTP ICostDT].
  Implicit Arguments WssoCpltClos [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDT ICostDTupT ICostDRecT minC].
  Implicit Arguments WssoCpltSIL [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDT ICostDTupT ICostDRecT].
  Implicit Arguments WssoCpltSI [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDT ICostDTupT ICostDRecT].

  Implicit Arguments WssoCpltS [CTDT CTDT_TUP CTDT_REC CTDTP
    CTDTP_TUP CTDTP_REC].
  Implicit Arguments PattP [CTDT CTDTP].
  Implicit Arguments UDataConvert [t1 t2].

  Lemma extractTypSigLEquiv : ∀ n slt ts prf prf′,
    LTypePSListEqSigT slt (sigifyList (projT1 ts) (projT2 ts)) →
    (nth_certain slt n prf) =t= (extractType n ts prf′).

  Definition extractDat(n : nat)(ts : sigT LTypesPS)(lu : list UDataRaw)
    (utp : UTupleP ts lu)(prf : n < length (‘ (projT2 ts))) :
    sig (UDataP (extractType n ts prf)).
```

```
Lemma LTypeMapEqExistFromLRT : ∀ s s′ ltr ltr′ lrps lrps′ v e,
  LRTypesPSEq s s′
  (exist (LRTypesP s) ltr lrps) (exist (LRTypesP s′) ltr′ lrps′) →
  VaridMapMod.Raw.PX.MapsTo v e ltr → (LTmapOK s) ltr → (LTmapOK s′) ltr′ →
  ∃ e′ : sig (LTypeRawCeiling s′), VaridMapMod.Raw.PX.MapsTo v e′ ltr′.
Lemma sigifyLCeil2Rinv : ∀ s (e : sig (LTypeRawCeiling s)) v lrt mp,
  ‘e = existT (projT1 (‘e)) (‘ (sigifyLCeil2R s v e lrt mp)).
Lemma sigifyLCeilInv : ∀ s (e : sig (LTypeRawCeiling s)) lts mp,
  ‘e = existT (projT1 (‘e)) (‘ (sigifyLCeil s e lts mp)).
Lemma LTypePSEqSeq : ∀ s s′ t t′, LTypePSEq s s′ t t′ → s = s′.

Section pattElsPEquivAssistS.
  Variable sn : Size.
  Variable sn′ : Size.
  Variable sr : Size.
  Variable tn : LTypePS sn.
  Variable tn′ : LTypePS sn′.
  Variable tr : LTypePS sr.
  Variable pel : list PattEl.
  Variable H : existT sn tn =t= existT sn′ tn′.
  Variable H0 : PattElsP sn sr tn tr pel.
  Variable pel′ : list PattEl.
  Variable snn : Size.
  Variable snn′ : Size.
  Variable tnn : LTypePS snn.
  Variable tnn′ : LTypePS snn′.
  Variable teq : existT snn tnn =t= existT snn′ tnn′.
  Variable pep : PattElsP snn sr tnn tr pel′.
  Variable pe : PattEl.
  Variable pel″ : list PattEl.
  Variable J : pel′ = pe :: pel″.
  Variable pattElsPEquivInner : ∀ (peli′ : list PattEl)
    (snn snn′ : Size) (tnn : LTypePS snn)
    (tnn′ : LTypePS snn′),
    existT snn tnn =t= existT snn′ tnn′ →
    PattElsP snn sr tnn tr peli′ →
    length peli′ < length pel′ →
    PattElsP snn′ sr tnn′ tr peli′.

  Section pattElsPEquivAssistRecS.

    Variable v : Varid.
    Variable J0 : pe = pattVarid v.

    Lemma PattElsPRecAss : PattElsP snn′ sr tnn′ tr pel′.

  End pattElsPEquivAssistRecS.
  Section pattElsPEquivAssistTupS.

    Variable p : nat.
    Variable J0 : pe = pattPosParam p.

    Lemma PattElsPTupAss : PattElsP snn′ sr tnn′ tr pel′.
  End pattElsPEquivAssistTupS.
End pattElsPEquivAssistS.

Lemma pattElsPEquiv : ∀ sn sn′ sr tn tn′ tr pel,

  existT sn tn =t= existT sn′ tn′ →
  PattElsP sn sr tn tr pel → PattElsP sn′ sr tn′ tr pel.

Lemma PattPRImpl : ∀ r r′ t c p, r <r= r′→
  PattP ICostDT r t c p → PattP ICostDT r′ t c p.

Lemma extractTypeRInvar : ∀ v v′ tr t inprf mprf, v =v= v′ →
  existT (projT1 (‘t)) (sigifyLCeil2R (projT1 tr) v t (projT2 tr) mprf) =t=
  extractTypeR v′ tr inprf.

Definition extractDatR(v : Varid)(tr : sigT LRTypesPS)
  (ru : VaridMapModRaw.t UDataRaw)(rup : URecordP tr ru)
  (inprf : VaridMapMod.In v (LRTypesPSRecoverMap (projT1 tr) (projT2 tr))) :
  sig (UDataP (extractTypeR v tr inprf)).

Definition pattExtract (r : RssoI)(t : sigT LTypePS)(c : sig (CTDTP t))
```

```
    (t′ : sigT LTypePS)(c′ : sig (CTDTP t′))(upin : UPotI t c)
    (pelps : sig (PattElsP (projT1 t) (projT1 t′) (projT2 t) (projT2 t′)))
    (costconstr : CT_PD_LT_HET (pattRel (‘pelps)) ICostDT ICostDT c c′)
    : UPotI t′ c′.
  Lemma TEquivRIncl : ∀ v T T′ r r′, r <r= r′ →
    VaridMapMod.MapsTo v T r → VaridMapMod.MapsTo v T′ r′ →
    T =TC= T′.
  Section extractDatNavWClosS.
    Variable r : RssoI.
    Variable t : ProtoT.
    Variable c : sig (CTDTP t).
    Variable p : sig (PattP ICostDT r t c).
    Variable wc : sigWssoClos minC r.

    Variable rfunc : ∀ (t′ : sigT LTypePS)
      (c′ : sig (CTDTP t′)) r′, (sigWssoClos minC r′) →
      sig (EssoRawCeilingT r′ t c t′ c′) → UPotI t′ c′.
    Variable v : Varid.
    Variable lpe : list PattEl.
    Hypothesis pp : PattP ICostDT r t c (patt v lpe).
    Let patMapPred(r′ : RssoI) :=
      ∃ s : Size,
        ∃ t′ : LTypePS s,
          ∃ c′ : sig (CTDTP (existT s t′)),
            VaridMapMod.MapsTo v (TssoGenDataT (existT (existT s t′) c′)) r′ ∧
            PattElsP s (projT1 t) t′ (projT2 t) lpe ∧
            CT_PD_LT_HET (pattRel lpe) ICostDT ICostDT c′ c.
    Section DatExtractAssistS.
      Variable ri : RssoI.
      Variable wcir : WssoCpltSIL minC.
      Variable wcp : WssoCpltClos ri wcir.
      Variable wpl : list (Rsso ICostDT × sigT (WssoCpltSI minC)).
      Variable rn : Rsso ICostDT.
      Variable wt : sigT (WssoCpltSI minC).
      Variable J0 : wcir = (rn, wt) :: wpl.
      Variable f :
        sigTD (Fsso CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC).
      Variable J1 : VaridMapMod.MapsTo v f (‘ (‘ (projT2 wt))).
      Hypothesis H : patMapPred ri.
      Lemma riWtProjEq : (projT1 wt) <r= ri.
      Definition DatExtractAssist : UPotI t c.
    End DatExtractAssistS.
    Definition extractDatNavWClos : UPotI t c.
  End extractDatNavWClosS.
  Definition extractDatFromEnv(r : RssoI)(t : ProtoT)(c : sig (CTDTP t))
    (p : sig (PattP ICostDT r t c))(wc : sigWssoClos minC r)
    (rfunc : ∀ (t′ : sigT LTypePS)
      (c′ : sig (CTDTP t′)) r′, (sigWssoClos minC r′) →
      sig (EssoRawCeilingT r′ t c t′ c′) → UPotI t′ c′) : UPotI t c.
End UPattMakeSect.
Local  Close Scope program_scope.
```

Listing D.34: The expression language reduction function: tuple branch

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Arith.Plus.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Wellfounded.Inverse_Image.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.ExprSSO.

Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.

Local Open Scope program_scope.

Section UTupleMakeSect.
 Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).

  Context ‘{ICostDT : CostDT CTDT CTDTP}.
  Context ‘{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context ‘{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.

  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.

  Variable minC : ∀ t : ProtoT, sig (CTDTP t).
  Hypothesis minCMin : ∀ t c u, CT_PD_interp(CostBase := ICostDT)
    t (‘c) u (“c) = 0.

  Let RssoI := Rsso ICostDT.
  Let EssoPI := EssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let EssoTupPI := EssoTupP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let EssoRawI := EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let UPotI := UPot CTDT CTDTP ICostDT.
  Implicit Arguments getEssoRawTC [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC].
  Implicit Arguments EssoRawCeilingL [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDT ICostDTupT].
  Implicit Arguments sig2ifyExprList [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDT ICostDTupT ICostDRecT].
  Implicit Arguments sig2ifyProj [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDTupT ICostDRecT].
  Implicit Arguments uPPot [CTDT CTDTP ].
  Implicit Arguments uPDat [CTDT CTDTP ].
  Implicit Arguments ListCostLeTQuant [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
    CTDTP_REC ICostDT ICostDTupT].
  Implicit Arguments UDatListFoldFunc [CTDT CTDTP ICostDT].
  Implicit Arguments UCostTupSigifyList [CTDT CTDT_TUP CTDT_REC CTDTP
    CTDTP_TUP CTDTP_REC].
  Implicit Arguments Build_UPot [CTDT CTDTP ].
  Implicit Arguments uPPot [T uraw u CTDT CTDTP cb].
  Implicit Arguments uPDat [T uraw u CTDT CTDTP cb].

  Variable r : RssoI.

  Variable etp : sigTT (EssoTupPI r).
  Hypothesis erlprf : ∀ er, List.In er (projTT3 etp) →
  EssoPI r (projT1 (getEssoRawTC er))
  (projT2 (getEssoRawTC er)) er.
Hypothesis erlcprf : ∀ er, List.In er (projTT3 etp) →
  EssoRawCeilingL (projTT1 etp) (projTT2 etp) er.
Let ts := projTT1 etp.
Let tssf := (sigifyList (projT1 ts) (projT2 ts)).
Let cs := projTT2 etp.
Let erl := projTT3 etp.
Hypothesis etpprf : LTypePSListEqSigT tssf
  (List.map (fun er ⇒ projT1 (getEssoRawTC er)) erl).
Variable rett : ProtoT.
Variable retc : sig (CTDTP rett).
Let buildProtoTFromSTTS(ts′ : sigT LTypesPS) :=
  (existT (projT1 ts′)(buildLTypePSFromTS (projT1 ts′) (projT2 ts′))).
Let LTypeB := fun ts ⇒
  existT (projT1 ts)(buildLTypePSFromTS (projT1 ts) (projT2 ts)).
Let tsTConstrCmp(ts : sigT LTypesPS)(t : sigT LTypePS) :=
  ProtoEqTSigT (LTypeB ts) t.
Hypothesis rettPrf : rett =t= buildProtoTFromSTTS ts.
Hypothesis retcPrf : CT_PD_LT_HET tsTConstrCmp
  ICostDTupT ICostDT cs retc.
Let datListPotMax := CT_PD_max (CostBase := ICostDTupT) ts cs.
Let retPotMax := datListPotMax + 1.
Let ExprDQualP := sig2
    (EssoRawCeilingL (ICostDT := ICostDT) (ICostDTupT := ICostDTupT) ts cs)
  (fun erp : EssoRawI ⇒
    EssoPI r (projT1 (getEssoRawTC erp)) (projT2 (getEssoRawTC erp)) erp).
Variable (redfunc : ∀ na : ExprDQualP,
  (UPotI (projT1 (getEssoRawTC (proj1_sig2 na)))
    (projT2 (getEssoRawTC (proj1_sig2 na))))).
Let evalEls(er : ExprDQualP) :=
  (existTD (projT1 (getEssoRawTC (proj1_sig2 er)))
    (projT2 (getEssoRawTC (proj1_sig2 er))) (redfunc er)).
Let datList := List.map evalEls (sig2ifyExprList r etp erlprf erlcprf).
Let foldFuncPot(dat : sigTD UPotI)(inpot : Potential) : Potential :=
  inpot + uPPot (projTD1 dat) (projTD2 dat) (projTD3 dat).
Let actualListPot := List.fold_right foldFuncPot 0 datList.
Let extractUDataRaw(up : sigTD UPotI) :=
 ‘ (uPDat (projTD1 up) (projTD2 up) (projTD3 up)).
Let uDataRawList := List.map extractUDataRaw datList.
Let extractTFromEDQ(edq : ExprDQualP) :=
  projT1 (getEssoRawTC (proj1_sig2 edq)).
Lemma datListTypInf : ∀ pr, List.In pr (combine tssf datList) →
  fst pr =t= (projTD1 (snd pr)).
Lemma uDataPSigPrf : LTypePSEq (projT1 rett) (projT1 ts) (projT2 rett)
          (buildLTypePSFromTS (projT1 ts) (projT2 ts)).
Lemma uTuplePSig_assist : ∃ lus′ : list (sigT LTypePS × UDataRaw),
          LTypePSListEqSigT (fst (split lus′))
            (sigifyList (projT1 ts) (projT2 ts)) ∧
          snd (split lus′) = uDataRawList ∧
          (∀ pr : sigT LTypePS × UDataRaw,
           In pr lus′ → UDataP (fst pr) (snd pr)).
Let uDataPSig : sig (UDataP rett) :=
  exist (UDataP rett) (UTupleData (projT1 ts) (projT2 ts) uDataRawList)
  (UTupleDataP ts rett uDataRawList uDataPSigPrf
    (UTuplePIntro ts uDataRawList uTuplePSig_assist)).
Let uTuplePSig : sig (UTupleP (projTT1 etp)) :=
  exist (UTupleP (projTT1 etp)) uDataRawList
  (UTuplePIntro ts uDataRawList uTuplePSig_assist).
Let extractUDataPot := fun up : sigTD UPotI ⇒
```

```
                    '(uPPot (projTD1 up) (projTD2 up) (projTD3 up)).
  Lemma tcCorrect : ListCostLeTQuant (ICostDT := ICostDT)
    (ICostDTupT := ICostDTupT) ts cs (projTT3 etp).
  Lemma uTupleInterpEq : fold_right foldFuncPot 0 datList ≤
    CT_PD_interp (projTT1 etp) ('cs) uTuplePSig (''cs).
  Theorem retPotCorrectStrong :
    actualListPot < CT_PD_interp rett (' retc) uDataPSig (''retc).
  Lemma retPotCorrectWeak : actualListPot < CT_PD_max rett retc.
  Lemma retPotCorrectWeakS : actualListPot + 1 ≤ CT_PD_max rett retc.
  Lemma retPotCorrectStrongS :
    actualListPot + 1 ≤ CT_PD_interp rett (' retc) uDataPSig (''retc).
  Definition UTupleMake : UPotI rett retc :=
    Build_UPot _ _ _ ICostDT rett retc (actualListPot + 1) uDataPSig
    retPotCorrectStrongS.
End UTupleMakeSect.
Local Close Scope program_scope.
```

Listing D.35: The expression language reduction function: record branch

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Arith.Plus.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Wellfounded.Inverse_Image.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.
Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.MapLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.ExprSSO.
Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.
Local Open Scope program_scope.
Module VaridMapAuxFacts := MapAuxFacts varidPred.PredidDecidable VaridMapMod.
Section URecordMakeSect.
  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).
  Context '{ICostDT : CostDT CTDT CTDTP}.
  Context '{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context '{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.
  Hypothesis ICostDTInProtoT :
```

```
  (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
Hypothesis ICostDTupTInSigTTsEq :
  (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.

Variable minC : ∀ t : ProtoT, sig (CTDTP t).
Hypothesis minCMin : ∀ t c u, CT_PD_interp(CostBase := ICostDT)
  t (‘c) u (“c) = 0.

Let RssoI := Rsso ICostDT.
Let EssoPI := EssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let EssoRecPI := EssoRecP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let EssoRawI := EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let UPotI := UPot CTDT CTDTP ICostDT.
Implicit Arguments getEssoRawTC [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
CTDTP_REC].
Implicit Arguments EssoRawCeilingR [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDRecT].
Implicit Arguments sig2ifyExprMap [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDTupT ICostDRecT].
Implicit Arguments sig2ifyProj [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDTupT ICostDRecT].
Implicit Arguments ListCostLeTQuant [CTDT CTDT_TUP CTDTP CTDTP_TUP
  ICostDT ICostDTupT].
Implicit Arguments MapCostLeTQuant [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDRecT].
Implicit Arguments UDatMapFoldFunc [CTDT CTDTP ICostDT].
Implicit Arguments UCostRecSigifyMap [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
CTDTP_REC].
Implicit Arguments Build_UPot [CTDT CTDTP ].
Implicit Arguments uPPot [T uraw u CTDT CTDTP cb].
Implicit Arguments uPDat [T uraw u CTDT CTDTP cb].

Variable r : RssoI.

Variable erp : sigTT (EssoRecPI r).

Hypothesis ermapok : SetoidList.NoDupA
  (@VaridMapMod.Raw.PX.eqk _) (projTT3 erp).

Let erm := VaridMapMod.Build_slist ermapok.

Hypothesis ermprf : ∀ v er, VaridMapMod.In v erm →
  VaridMapMod.MapsTo v er erm →
  EssoPI r (projT1 (getEssoRawTC er)) (projT2 (getEssoRawTC er)) er.

Hypothesis ermcprf : ∀ v er, VaridMapMod.In v erm →
  VaridMapMod.MapsTo v er erm →
  EssoRawCeilingR (projTT1 erp) (projTT2 erp) er.

Let tr := projTT1 erp.
Let trsf := (sigifyMap (projT1 tr) (projT2 tr)).
Let cr := projTT2 erp.

Hypothesis erpprf : (VaridMapMod.Equiv ProtoEqTSigT) trsf
  (VaridMapMod.map (fun er ⇒ projT1 (getEssoRawTC er)) erm).

Variable rett : ProtoT.
Variable retc : sig (CTDTP rett).

Let buildProtoTFromSTTR(tr′ : sigT LRTypesPS) :=
  (existT (projT1 tr′)(buildLTypePSFromRT (projT1 tr′) (projT2 tr′))).
Let LTypeB := fun tr ⇒
  existT (projT1 tr)(buildLTypePSFromRT (projT1 tr) (projT2 tr)).
Let trTConstrCmp(tr : sigT LRTypesPS)(t : sigT LTypePS) :=
  ProtoEqTSigT (LTypeB tr) t.

Hypothesis rettPrf : rett =t= buildProtoTFromSTTR tr.
Hypothesis retcPrf : CT_PD_LT_HET trTConstrCmp
  ICostDRecT ICostDT cr retc.

Let datListPotMax := CT_PD_max (CostBase := ICostDRecT) tr cr.
Let retPotMax := datListPotMax + 1.

Let ExprDQualP := sig2
  (EssoRawCeilingR (ICostDT := ICostDT) (ICostDRecT := ICostDRecT) tr cr)
  (fun er : EssoRawI ⇒
    EssoPI r (projT1 (getEssoRawTC er)) (projT2 (getEssoRawTC er)) er).
```

```
Variable (redfunc : ∀ na : ExprDQualP,
  (UPotI (projT1 (getEssoRawTC (proj1_sig2 na)))
    (projT2 (getEssoRawTC (proj1_sig2 na))))).
Let evalEls(er : ExprDQualP) :=
  (existTD (projT1 (getEssoRawTC (proj1_sig2 er)))
    (projT2 (getEssoRawTC (proj1_sig2 er))) (redfunc er)).
Let datMap := VaridMapMod.map evalEls
  (sig2ifyExprMap r erp ermapok ermprf ermcprf).
Let foldFuncPot(_ : Varid)(dat : sigTD UPotI)(inpot : Potential) :
  Potential := inpot + uPPot (projTD1 dat) (projTD2 dat) (projTD3 dat).
Let actualListPot := VaridMapMod.fold foldFuncPot datMap 0.
Let extractUDataRaw(up : sigTD UPotI) :=
 ‘ (uPDat (projTD1 up) (projTD2 up) (projTD3 up)).
Let uDataRawMap := VaridMapMod.map extractUDataRaw datMap.
Let extractTFromEDQ(edq : ExprDQualP) :=
  projT1 (getEssoRawTC (proj1_sig2 edq)).

Lemma uDataPSigPrf : LTypePSEq (projT1 rett) (projT1 tr) (projT2 rett)
  (buildLTypePSFromRT (projT1 tr) (projT2 tr)).
Lemma uRecordPSig_assist : (∃ tr′ : sigT LRTypesPS,
        LRTypesPSEq (projT1 tr) (projT1 tr′) (projT2 tr) (projT2 tr′) ∧
        (∀ v : VaridMapMod.key,
         VaridMapMod.In (elt:=sig (LTypeRawCeiling (projT1 tr))) v
           (LRTypesPSRecoverMap (projT1 tr) (projT2 tr)) →
         ∃ v′ : varidPred.PredidDecidable.t,
           v =v= v′ ∧ VaridMapMod.In (elt:=UDataRaw) v uDataRawMap) ∧
        (∀ (v : VaridMapMod.key) (u : UDataRaw),
         VaridMapMod.MapsTo v u uDataRawMap →
         ∃ v′ : varidPred.PredidDecidable.t,
           v′ =v= v ∧
           (∃ t : sig (LTypeRawCeiling (projT1 tr)),
              ∃ t′ : sigT LTypePS,
                EqdepFacts.eq_dep Size LTypeRaw (projT1 (‘t))
                  (projT2 (‘t)) (projT1 t′) (‘ (projT2 t′)) ∧
                UDataP t′ u ∧
                VaridMapMod.MapsTo v′ t
                  (LRTypesPSRecoverMap (projT1 tr) (projT2 tr))))).
Let uDataPSig : sig (UDataP rett) :=
  exist (UDataP rett) (URecordData (projT1 tr) (projT2 tr)
    (VaridMapMod.this uDataRawMap))
  (URecordDataP tr rett uDataRawMap uDataPSigPrf
    (URecordPIntro tr uDataRawMap uRecordPSig_assist)).
Let uRecordPSig : sig (URecordP (projTT1 erp)) :=
  exist (URecordP (projTT1 erp)) (VaridMapMod.this uDataRawMap)
  (URecordPIntro tr uDataRawMap uRecordPSig_assist).
Let extractUDataPot := fun up : sigTD UPotI ⇒
                      ‘(uPPot (projTD1 up) (projTD2 up) (projTD3 up)).
Lemma FoldFuncTranspose :
  VaridMapWPties.transpose_neqkey eq (UDatMapFoldFunc (ICostDT := ICostDT)).
Add Morphism (UDatMapFoldFunc (ICostDT := ICostDT)) with signature
  (varidPred.PredidDecidable.eq ==> eq ==> eq ==> eq) as
    UDatMapFoldFuncVaridLeibM.
Lemma UDatMapFoldFuncEqual : ∀ uc uc′ n,
  VaridMapMod.Equal uc uc′ →
  VaridMapMod.fold (UDatMapFoldFunc (ICostDT := ICostDT)) uc n =
    VaridMapMod.fold (UDatMapFoldFunc (ICostDT := ICostDT)) uc′ n.
Lemma ExprMapEqual : ∀
  (eraw eraw′: VaridMapMod.Raw.t (EssoRaw CTDT CTDT_TUP CTDT_REC
    CTDTP CTDTP_TUP CTDTP_REC))
  (erawp : SetoidList.NoDupA (@VaridMapMod.Raw.PX.eqk _) eraw)
  (erawp′ : SetoidList.NoDupA (@VaridMapMod.Raw.PX.eqk _) eraw′),
  eraw = eraw′ → VaridMapMod.Equal
  {| VaridMapMod.this := eraw; VaridMapMod.NoDup := erawp |}
```

```
    {| VaridMapMod.this := eraw'; VaridMapMod.NoDup := erawp' |}.
  Lemma FindInMapsToInv : ∀ elt elt' k l (e : elt) m m'
    (x : ∀ (t : elt'), VaridMapMod.find l m = Some t → VaridMapMod.t elt)
    (x' : ∀ (t : elt'),
      VaridMapMod.find l m' = Some t → VaridMapMod.t elt)
    (nm : VaridMapMod.find l m = None → False)
    (nm' : VaridMapMod.find l m' = None → False)
    ef ef',
    (∀ t t' e e', VaridMapMod.Equal (x t e) (x' t' e')) → (
      VaridMapMod.MapsTo k e
      (match (VaridMapMod.find (elt := elt') l m) as tf return
           (VaridMapMod.find l m) = tf → VaridMapMod.t elt with
           | Some tc ⇒ fun J0 ⇒ x tc J0
           | None ⇒ fun J0 ⇒ False_rect _ (nm J0)
        end ef) ↔
      VaridMapMod.MapsTo k e
      (match (VaridMapMod.find (elt := elt') l m') as tf return
           (VaridMapMod.find l m') = tf → VaridMapMod.t elt with
           | Some tc ⇒ fun J0 ⇒ x' tc J0
           | None ⇒ fun J0 ⇒ False_rect _ (nm' J0)
        end ef')).
  Lemma UCostRecSfyEq : ∀ tcm tcm' um udl inclprf inclprf' inprf inprf',
    VaridMapMod.Equal tcm tcm' →
    VaridMapMod.Equal
    (UCostRecSigifyMapAssistInner CTDT CTDTP tcm um udl inclprf inprf)
    (UCostRecSigifyMapAssistInner CTDT CTDTP tcm' um udl inclprf' inprf').
  Lemma ExpMapEqImplUCostRecSigEqual : ∀ tr cr ud em em' uctprf uctprf',
    VaridMapMod.Equal em em' →
    VaridMapMod.Equal
    (UCostRecSigifyMap (CTDTP := CTDTP) (CTDTP_TUP := CTDTP_TUP)
      (Build_UCostRec CTDT_REC CTDTP_REC tr cr ud) em uctprf)
    (UCostRecSigifyMap
      (Build_UCostRec CTDT_REC CTDTP_REC tr cr ud) em' uctprf').

  Lemma tcCorrect : MapCostLeTQuant (ICostDT := ICostDT)
    (ICostDRecT := ICostDRecT) tr cr erm.
  Lemma uRecordInterpEq : VaridMapMod.fold foldFuncPot datMap 0 ≤
      CT_PD_interp (projTT1 erp) ('cr) uRecordPSig ("cr).
Check UCostRecSigifyMap.
Admitted.
  Theorem retPotCorrectStrong :
    actualListPot < CT_PD_interp rett (' retc) uDataPSig ("retc).

  Lemma retPotCorrectStrongS :
    actualListPot + 1 ≤ CT_PD_interp rett (' retc) uDataPSig ("retc).
  Definition URecordMake : UPotI rett retc :=
    Build_UPot _ _ _ ICostDT rett retc (actualListPot + 1) uDataPSig
    retPotCorrectStrongS.
End URecordMakeSect.
Local Close Scope program_scope.
```

Listing D.36: The expression language reduction function: application branch

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Arith.Plus.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Wellfounded.Inverse_Image.
Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.ExprSSO.

Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.

Local Open Scope program_scope.

Section UAppMakeSect.
  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).

  Context ‘{ICostDT : CostDT CTDT CTDTP}.
  Context ‘{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context ‘{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.

  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.

  Variable minC : ∀ t : ProtoT, sig (CTDTP t).

  Hypothesis minCMin : ∀ t u, CT_PD_interp(CostBase := ICostDT)
    t (‘ (minC t)) u (“ (minC t)) = 0.

  Let RssoI := Rsso ICostDT.
  Let EssoPI := EssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let EssoRawI := EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Let UPotI := UPot CTDT CTDTP ICostDT.

  Implicit Arguments WssoCpltS [CTDT CTDT_TUP CTDT_REC CTDTP
    CTDTP_TUP CTDTP_REC].
  Implicit Arguments PattP [CTDT CTDTP].
  Implicit Arguments UDataConvert [t1 t2].
  Implicit Arguments CTCombAppPrf [CTDT CTDTP t t′].
  Implicit Arguments uPPot [T uraw u CTDT CTDTP cb].
  Implicit Arguments uPDat [T uraw u CTDT CTDTP cb].

  Definition addR(v : Varid)(t : ProtoT)(r : RssoI) :=
    (VaridMapMod.add v (TssoGenDataT (existT t (minC t))) r).

  Program Definition emptyWsso(r : RssoI) :
    (Wsso CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC r) :=
    VaridMapMod.empty (sigTD
      (Fsso CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC)).
  Obligation 1.
    red.
    intros v e H.
    apply VaridMapWFacts.empty_mapsto_iff in H.
    exfalso; assumption.
  Qed.

Implicit Arguments sigWssoClos [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDTupT ICostDRecT].
Implicit Arguments emptyRsso [CTDT CTDTP ICostDT].
Implicit Arguments WssoCpltClos [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDTupT ICostDRecT minC].
```

```
Implicit Arguments WssoCpltSIL [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDTupT ICostDRecT].
Implicit Arguments WssoCpltSI [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
  CTDTP_REC ICostDT ICostDTupT ICostDRecT].

  Definition bindVar(r : RssoI)(v : Varid)(t : ProtoT)(u : sig (UDataP t))
    (wc : sigWssoClos minC r) : sigWssoClos minC (addR v t r).
  refine (
    let upot := Build_UPot _ _ _ CTDT CTDTP ICostDT t (minC t) 1 u _ in
      let fv := FssoVal CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
        CTDTP_REC minC t upot in
        let nfps :=
          exist (FssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC
            (addR v t emptyRsso) (TssoGenDataT (existT t (minC t))))
          fv _ in
          let fsso := existTD
            (P := (Fsso CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC))
            (addR v t emptyRsso) (TssoGenDataT (existT t (minC t))) nfps in
            let wraw :=
              VaridMapMod.add v fsso (VaridMapMod.empty (sigTD
                (Fsso CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC)))
              in let fwf :
                FssoMapWFInR CTDT CTDT_TUP CTDT_REC CTDTP
                CTDTP_TUP CTDTP_REC minC (addR v t emptyRsso) wraw := _ in
              let wso := exist
                (FssoMapWFInR CTDT CTDT_TUP CTDT_REC CTDTP
                  CTDTP_TUP CTDTP_REC minC (addR v t emptyRsso)) wraw fwf in
              let wcplt := exist
                (WssoCplt CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC
                  (addR v t emptyRsso)) wso _
                in
      exist (WssoCpltClos (VaridMapWPties.update r (addR v t emptyRsso)))
      ((r, (existT (addR v t emptyRsso) wcplt)) :: (‘wc)) _
  ).
  Proof.
    apply CT_PD_interp_prf.
    exact (FssoValP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC
      (addR v t emptyRsso) t upot).
    subst wraw.
    red.
    intros v′ e H.
    apply VaridMapWFacts.add_mapsto_iff in H.
    red in wc.
    destruct H.
    split.
    destruct H as [H H0].
    rewrite ← H0.
    subst fsso.
    simpl.
    reflexivity.
    ∃ (TssoGenDataT (existT t (minC t))).
    destruct H as [H H0].
    split.
    rewrite ← H.
    unfold addR.
    simpl.
    apply VaridMapWFacts.add_mapsto_iff.
    left.
    split; reflexivity.
    rewrite ← H0.
    subst fsso.
    simpl.
    reflexivity.
    destruct H as [H H0].
    apply VaridMapWFacts.empty_mapsto_iff in H0.
    exfalso; assumption.
    red.
  intros v′ H.
  subst wso.
  simpl.
  assert (v =v= v′) as H0.
  unfold addR in H.
  apply VaridMapWFacts.add_in_iff in H.
  destruct H.
  assumption.
  apply VaridMapWFacts.empty_in_iff in H.
  exfalso; assumption.
  rewrite ← H0.
  apply VaridMapWFacts.add_in_iff.
  left.
  reflexivity.
  simpl.
  eapply WssoCWFPrInd.
  exact (“wc).
Defined.

Section FuncExtractAssistS.

  Variables t t′ : ProtoT.
  Variable c : sig (CTDTP t).
  Variable c′ : sig (CTDTP t′).
  Variable ri : RssoI.
  Variable wci : sigWssoClos minC ri.
  Variable wfp : Acc lt (length ((‘wci))).
  Variable w : WssoCpltSIL minC.
  Variable wt : sigT (WssoCpltSI minC).
  Variable wl : list (RssoI × sigT (WssoCpltSI minC)).
  Variable rn : RssoI.
  Variable J0 : (‘wci) = ((rn, wt) :: wl).
  Variable f :
    sigTD (Fsso CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC minC).
  Variable v : Varid.
  Variable J1 : VaridMapMod.MapsTo v f (‘ (‘ (projT2 wt))).
  Variable upin : UPotI t′ c′.
  Variable rincl : projT1 wt <r= ri.
  Variable H : ∃ tso : TssoGen CTDT CTDTP,
        ∃ tso′ : TssoGen CTDT CTDTP,
          ∃ varg : Varid,
            ∃ c″ : sig (CTDTP t),
              VaridMapMod.MapsTo v tso (projT1 wt) ∧
              tso′ =TC= tso ∧
              CTCombAppPrf ICostDT c′ c″ c ∧
              tso′ = TssoGenFunc varg t′ (existT t c″).
  Variable newRed : ∀ (v′ : Varid) (r′ : Rsso ICostDT)
    (t″ : sigT LTypePS)
    (c″ : (fun t′ : sigT LTypePS ⇒ sig (CTDTP t′)) t″),
    existT t″ c″ <tc< existT t c →
    sigWssoClos minC (addR v′ t′ r′) →
    sig (EssoPI (addR v′ t′ r′) t″ c″) → UPotI t c.

  Definition FuncExtractAssist : UPotI t c.
  refine (
    match (‘ (projTD3 f)) as ft return _ = ft → (UPotI t c) with
      | FssoVal t″ upot ⇒ fun J ⇒ !
      | FssoBFunc v′ ti to toc func ⇒ fun J ⇒
        let uorig := uPDat t′ c′ upin in
          let uretp := _ in
          let uret : UPotI to toc :=
            func (exist (UDataP ti) (‘uorig) uretp)
            in UPotConvert CTDT CTDTP to t toc c uret _ _
      | FssoDat r′ t″ c″ e ⇒ fun J ⇒ !
      | FssoEFunc r′ v′ ti to toc e ⇒ fun J ⇒
        let uorig := uPDat t′ c′ upin in
          let newWCEnv := bindVar ri v′ t′ uorig wci in
            let ep := (exist (EssoPI (addR v′ t′ ri) to toc) e _)
              in newRed v′ ri to toc _ newWCEnv ep
```

```
                    end eq_refl
).
destruct H as [tso H0].
destruct H0 as [tso' H1].
destruct H1 as [varg H2].
destruct H2 as [c'' H3].
decompose [and] H3.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
clear J1.
rewrite J in f0.
dependent destruction f0.
destruct H8 as [T H8].
destruct H8 as [H8 H9].
assert (H10 := VaridMapWFacts.MapsTo_fun H0 H8).
rewrite ← H10 in H9.
rewrite ← H2 in H9.
rewrite H5 in H9.
red in H9.
assumption.
destruct uorig as [u up].
simpl.
eapply UDataConvertP; eauto.
destruct H as [tso H0].
destruct H0 as [tso' H0].
destruct H0 as [varg H0].
destruct H0 as [c'' H0].
decompose [and] H0.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
clear J1.
rewrite J in f0.
dependent destruction f0.
destruct H8 as [T H8].
destruct H8 as [H8 H9].
assert (H10 := VaridMapWFacts.MapsTo_fun H1 H8).
rewrite ← H10 in H9.
rewrite ← H3 in H9.
rewrite H5 in H9.
red in H9.
decompose [and] H9.
symmetry; assumption.
destruct H as [tso H0].
destruct H0 as [tso' H0].
destruct H0 as [varg H0].
destruct H0 as [c'' H0].
decompose [and] H0.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
```

```
clear J1.
rewrite J in f0.
dependent destruction f0.
destruct H8 as [T H8].
destruct H8 as [H8 H9].
assert (H10 := VaridMapWFacts.MapsTo_fun H1 H8).
rewrite ← H10 in H9.
rewrite ← H3 in H9.
rewrite H5 in H9.
red in H9.
decompose [and] H9.
simpl in H11.
assumption.
destruct H as [tso H0].
destruct H0 as [tso' H0].
destruct H0 as [varg H0].
destruct H0 as [c'' H0].
decompose [and] H0.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
clear J1.
rewrite J in f0.
dependent destruction f0.
destruct H8 as [T H8].
destruct H8 as [H8 H9].
assert (H10 := VaridMapWFacts.MapsTo_fun H1 H8).
rewrite ← H10 in H9.
rewrite ← H3 in H9.
rewrite H5 in H9.
red in H9.
decompose [and] H9.
red in H2.
red.
intros u u' H15.
rewrite ← H2 with (u := uorig).
rewrite CT_PD_T_le_pot_eq in H14.
simpl in H14.
assert (H16 := H14 u u').
rewrite plus_comm.
replace (CT_PD_interp to (‘toc) u (“toc)) with
   (CT_PD_interp to (‘toc) u (“toc) + 0).
eapply plus_le_compat; eauto.
apply le_0_n.
auto.
destruct H as [tso H0].
destruct H0 as [tso' H1].
destruct H1 as [varg H2].
destruct H2 as [c''' H3].
decompose [and] H3.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
clear J1.
rewrite J in f0.
dependent destruction f0.
destruct H8 as [T H8].
destruct H8 as [H10 H11].
assert (H12 := VaridMapWFacts.MapsTo_fun H0 H10).
rewrite ← H12 in H11.
rewrite ← H2 in H11.
rewrite H5 in H11.
red in H11.
assumption.
destruct H as [tso H0].
destruct H0 as [tso′ H0].
destruct H0 as [varg H0].
destruct H0 as [c″ H0].
decompose [and] H0.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
clear J1.
rewrite J in f0.
dependent destruction f0.
destruct H8 as [T H8].
destruct H8 as [H10 H11].
assert (H12 := VaridMapWFacts.MapsTo_fun H1 H10).
rewrite ← H12 in H11.
rewrite ← H3 in H11.
rewrite H5 in H11.
red in H11.
decompose [and] H11.
simpl in H13.
red.
unfold addR.
eapply EssoERIncl; eauto.
assert (r′ <r= ri) as H17.
transitivity x.
assumption.
rewrite ← rincl.
assumption.
rewrite H17.
assert (TssoGenDataT (existT ti (minC ti)) =TC=
   TssoGenDataT (existT t′ (minC t′))) as H18.
red.
apply CT_PD_T_eq_pot_eq.
intros u u′.
simpl.
rewrite minCMin.
rewrite minCMin.
reflexivity.
rewrite H18.
reflexivity.
destruct H as [tso H0].
destruct H0 as [tso′ H0].
destruct H0 as [varg H0].
destruct H0 as [c″ H0].
decompose [and] H0.
assert (H6 := (“(‘ (projT2 wt)))).
red in H6.
assert (H7 := H6 v f J1).
destruct H7 as [H7 H8].
destruct f.
red in f0.
destruct f0.
simpl in ×.
clear J1.
rewrite J in f0.
```

```
    dependent destruction f0.
    destruct H8 as [T H8].
    destruct H8 as [H10 H11].
    assert (H12 := VaridMapWFacts.MapsTo_fun H1 H10).
    rewrite ← H12 in H11.
    rewrite ← H3 in H11.
    rewrite H5 in H11.
    red in H11.
    decompose [and] H11.
    simpl in H11.
    red in H2.
    apply CT_PD_T_lt_pot_eq.
    rewrite CT_PD_T_le_pot_eq in H16.
    intros u u′.
    simpl in ×.
    assert (H17 := H16 u u′).
    assert (H18 := H2 uorig u′).
    rewrite ← H18.
    red.
    rewrite plus_comm.
    replace (S (CT_PD_interp to (‘toc) u (“toc))) with
      ((CT_PD_interp to (‘toc) u (“toc)) + 1).
    apply plus_le_compat.
    assumption.
    apply CT_PD_interp_prf.
    apply plus_comm.
  Defined.

End FuncExtractAssistS.

Definition funcExtractionPred(r : RssoI)(t t′ : ProtoT)(c : sig (CTDTP t))
  (c′ : sig (CTDTP t′))(v : Varid) :=
  (∃ tso : TssoGen CTDT CTDTP,
    ∃ tso′ : TssoGen CTDT CTDTP,
      ∃ varg : Varid,
        ∃ c″ : sig (CTDTP t),
          VaridMapMod.MapsTo v tso r ∧
          tso′ =TC= tso ∧
          CTCombAppPrf ICostDT c′ c″ c ∧
          tso′ = TssoGenFunc varg t′ (existT t c″)).

Definition extractFuncFromEnv(r : RssoI)(t t′ : ProtoT)(c : sig (CTDTP t))
  (c′ : sig (CTDTP t′))(v : Varid)(wc : sigWssoClos minC r)
  (newRed : ∀ v′ r′ t″ c″, existT t″ c″ <tc< existT t c →
    sigWssoClos minC (addR v′ t′ r′) →
    sig (EssoPI (addR v′ t′ r′) t″ c″) → UPotI t c) :
  funcExtractionPred r t t′ c c′ v →
  UPotI t′ c′ → UPotI t c.
intros H upin.
Check FuncExtractAssist.
refine (
  let fix extractFuncInner(ri : RssoI)
    (wcir : WssoCpltSIL minC)(wcp : WssoCpltClos ri wcir)


    (H′ : funcExtractionPred ri t t′ c c′ v) :=
    match wcir as wcp return _ = wcp → _ with
      | nil ⇒ fun J0 : wcir = nil ⇒ !
      | (rn, wt) :: wpl ⇒ fun J0 ⇒
        match (VaridMapMod.find v (‘ (‘ (projT2 wt)))) as fo
          return _ = fo → _ with
          | Some f ⇒ fun J1 ⇒ _




          | None ⇒ fun J1 ⇒
            extractFuncInner rn wpl _ _
        end eq_refl
    end eq_refl
    in extractFuncInner r (‘wc) (“wc) H
).
Proof.
  destruct wcp; discriminate.
  refine (FuncExtractAssist t t′ c c′ ri
    (exist (WssoCpltClos ri) wcir wcp) wt f v _ upin _ _ newRed).
  apply VaridMapWFacts.find_mapsto_iff in J1.
  assumption.
  destruct wcp.
  replace wt with (existT r0 w).
  simpl.
  reflexivity.
  injection J0; auto.
  replace (projT1 wt) with r′.
  apply updateROvwrt.
  injection J0; intros H1 H2 H3.
  rewrite ← H2.
  auto.
  destruct H′ as [tso H′].
  destruct H′ as [tso′ H′].
  destruct H′ as [varg H′].
  destruct H′ as [c″ H′].
  decompose [and] H′.
  ∃ tso.
  ∃ tso′.
  ∃ varg.
  ∃ c″.
  split.
  apply VaridMapWFacts.find_mapsto_iff in J1.
  simpl in ×.
  dependent destruction wcp.
  replace (projT1 wt) with r0.
  assumption.
  injection J0; intros H5 H6 H7.
  rewrite ← H6; auto.
  apply VaridMapWPties.update_mapsto_iff in H0.
  destruct H0.
  replace (projT1 wt) with r′.
  assumption.
  injection J0; intros H5 H6 H7.
  auto.
  destruct wt.
  simpl in ×.
  injection H6; auto.
  destruct wt.
  assert (VaridMapMod.In v r′) as H5a.
  destruct w as [w].
  simpl in ×.
  destruct w0.
  destruct x0.
  simpl in ×.
  red in w.
  destruct w.
  simpl in ×.
  red in f0.
  assert (H5 := f0 v f J1).
  destruct H5 as [H5 H6].
  destruct H6 as [T H6].
  destruct H6 as [H6 H7].
  red.
  red.
  ∃ T.
  replace r′ with x.
  assumption.
  injection J0; auto.
  destruct H0 as [H0 H0a].
```

```
contradict H0a.
assumption.
split; try assumption.
split; assumption.
apply VaridMapWFacts.not_find_in_iff in J1.
simpl in ×.
rewrite J0 in wcp.
dependent destruction wcp.
red in w.
red in w.
destruct w.
simpl in ×.
red in w.
contradict J1.
apply w.
destruct H′.
destruct H0 as [tso′ H0].
destruct H0 as [varg H0].
destruct H0 as [c″ H0].
decompose [and] H0.
red.
red.
∃ x0.
assumption.
assumption.
red.
destruct H′ as [tso H′].
destruct H′ as [tso′ H′].
destruct H′ as [varg H′].
destruct H′ as [c″ H′].
decompose [and] H′.
∃ tso.
∃ tso′.
∃ varg.
∃ c″.
split.
apply VaridMapWFacts.not_find_in_iff in J1.
simpl in ×.
dependent destruction wcp.
replace r0 with (projT1 wt) in H0.
destruct wt.
destruct w0.
red in w0.
simpl in ×.
contradict J1.
apply w0.
red.
red.
∃ tso.
assumption.
injection J0; intros H5 H6 H7.
rewrite ← H6; auto.
apply VaridMapWPties.update_mapsto_iff in H0.
destruct H0.
contradict J1.
replace wt with (existT r′ w).
simpl.
destruct w.
red in w.
simpl in ×.
apply w.
red.
red.
∃ tso.
assumption.
simpl.
injection J0; auto.
```

```
  destruct H0 as [H0 H0a].
  replace rn with r0.
  assumption.
  injection J0; auto.
  split; try assumption.
  split; assumption.
Defined.

End UAppMakeSect.

Local Close Scope program_scope.
```

Listing D.37: The expression language reduction functions

```
Require Import Coq.Setoids.Setoid.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Classes.SetoidDec.
Require Import Coq.Lists.List.
Require Import Coq.Program.Utils.
Require Import Coq.Program.Basics.
Require Import Coq.Program.Equality.
Require Import Coq.Classes.RelationClasses.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.Le.
Require Import Coq.Arith.Lt.
Require Import Coq.Arith.Plus.
Require Import Coq.Classes.Morphisms.
Require Import Coq.Arith.Wf_nat.
Require Import Coq.Wellfounded.Inverse_Image.
Require Import Coq.Wellfounded.Inclusion.

Require Coq.Lists.SetoidList.
Require Coq.FSets.FMapWeakList.
Require Coq.FSets.FMapFacts.

Require Import HBCL.Util.ListLemmas.
Require Import HBCL.Util.ArithLemmas.
Require Import HBCL.Util.sigTypes.
Require Import HBCL.HBCL_0_1.BaseLibs.Ids.Ids_S.
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.costAbstract.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.TypeSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.ExprSSO.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.RedTupBranch.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.RedRecBranch.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.RedPattBranch.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.reduction.RedAppBranch.

Import HBCL_0_1_Id_S.
Import HBCL_0_1_L_UTS.

Local Open Scope program_scope.

Section reduceSect.

  Variables (CTDT CTDT_TUP CTDT_REC : Type).
  Variable (CTDTP : (ProtoT → CTDT → Prop)).
  Variable (CTDTP_TUP : ((sigT LTypesPS) → CTDT_TUP → Prop)).
  Variable (CTDTP_REC : ((sigT LRTypesPS) → CTDT_REC → Prop)).

  Context ‘{ICostDT : CostDT CTDT CTDTP}.
  Context ‘{ICostDTupT : CostDTupT CTDT_TUP CTDTP_TUP}.
  Context ‘{ICostDRecT : CostDRecT CTDT_REC CTDTP_REC}.

  Hypothesis ICostDTInProtoT :
    (CT_PD_T_eqrel (CostBase := ICostDT)) = ProtoEqTSigT.
  Hypothesis ICostDTupTInSigTTsEq :
    (CT_PD_T_eqrel (CostBase := ICostDTupT)) = LTypesPSEqSigT.
  Hypothesis ICostDRecTInSigTTsEq :
   (CT_PD_T_eqrel (CostBase := ICostDRecT)) = LRTypesPSEqSigT.
Variable minC : ∀ t : ProtoT, sig (CTDTP t).
Hypothesis minCMin : ∀ t u, CT_PD_interp(CostBase := ICostDT)
   t (‘ (minC t)) u (“ (minC t)) = 0.

Let RssoI := Rsso ICostDT.
Let EssoPI := EssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let CssoPI := CssoP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let PattPI := PattP CTDT CTDTP.

Let EssoTupPI := EssoTupP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let EssoRecPI := EssoRecP CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let EssoRawI := EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
Let UPotI := UPot CTDT CTDTP ICostDT.

Implicit Arguments getEssoRawTC [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
CTDTP_REC].
Implicit Arguments EssoRawCeilingL [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC ICostDT ICostDTupT].
Implicit Arguments EssoRawCeilingR [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC ICostDT ICostDRecT].
Implicit Arguments sig2ifyExprList [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC ICostDT ICostDTupT ICostDRecT].
Implicit Arguments sig2ifyProj [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC ICostDT ICostDTupT ICostDRecT].
Implicit Arguments ListCostLeTQuant [CTDT CTDT_TUP CTDTP CTDTP_TUP
   ICostDT ICostDTupT].
Implicit Arguments UDatListFoldFunc [CTDT CTDTP ICostDT].
Implicit Arguments UCostTupSigifyList [CTDT CTDT_TUP CTDTP CTDTP_TUP].
Implicit Arguments Build_UPot [T uraw u CTDT CTDTP ].
Implicit Arguments uPPot [T uraw u CTDT CTDTP].
Implicit Arguments uPDat [T uraw u CTDT CTDTP].

Implicit Arguments WssoCpltS [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments sigWssoClos [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments extractDatFromEnv [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments extractFuncFromEnv [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments UTupleMake [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments URecordMake [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments extractDatFromEnv [CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP
   CTDTP_REC].
Implicit Arguments addR [CTDT CTDTP].

Let WssoCpltSI := WssoCpltS ICostDT ICostDTupT ICostDRecT minC.

Let sigWssoClosI := sigWssoClos ICostDT ICostDTupT ICostDRecT minC.

Let ExprDQualP(r : RssoI)(ts : sigT LTypesPS)(cs : sig (CTDTP_TUP ts)) :=
   sig2 (EssoRawCeilingL ts cs)
   (fun erp : EssoRawI ⇒
      EssoPI r (projT1 (getEssoRawTC erp)) (projT2 (getEssoRawTC erp)) erp).

Let ExprDQualPR(r : RssoI)(tr : sigT LRTypesPS)(cr : sig (CTDTP_REC tr)) :=
   sig2 (EssoRawCeilingR tr cr)
   (fun erp : EssoRawI ⇒
      EssoPI r (projT1 (getEssoRawTC erp)) (projT2 (getEssoRawTC erp)) erp).

Let EssoRawCeilingT(r : RssoI)(t : sigT LTypePS)(c : sig (CTDTP t))
(t′ : sigT LTypePS)(c′ : sig (CTDTP t′))(ep : sig (EssoPI r t′ c′)) :=
existT t′ c′ <tc< existT t c.

Let extractDatFromEnvI := extractDatFromEnv ICostDT ICostDTupT ICostDRecT
   ICostDTInProtoT minC minCMin.

Let extractFuncFromEnvI := extractFuncFromEnv ICostDT ICostDTupT ICostDRecT
   minC minCMin.

Let redfuncP(t : sigT LTypePS)(c : sig (CTDTP t)) :=
   ∀ (t′ : sigT LTypePS)(c′ : sig (CTDTP t′)) r,
      (sigWssoClosI r) → sig (EssoRawCeilingT r t c t′ c′) →
      UPotI t′ c′.
  Let addRI := addR ICostDT minC.
  Let redfuncA(t t′ : sigT LTypePS)(c : sig (CTDTP t)) :=
    ∀ (v : Varid) (r′ : Rsso ICostDT)
      (t″ : sigT LTypePS)
      (c″ : sig (CTDTP t″)),
      existT t″ c″ <tc< existT t c →
      sigWssoClosI (addRI v t′ r′) →
      sig (EssoPI (addRI v t′ r′) t″ c″) → UPotI t c.
  Let redfuncTup(r : RssoI)(ts : sigT LTypesPS)(cs : sig (CTDTP_TUP ts)) :=
    ∀ na : ExprDQualP r ts cs,
      (UPotI (projT1 (getEssoRawTC (proj1_sig2 na)))
        (projT2 (getEssoRawTC (proj1_sig2 na)))).
  Let redfuncRec(r : RssoI)(tr : sigT LRTypesPS)(cr : sig (CTDTP_REC tr)) :=
    ∀ na : ExprDQualPR r tr cr,
      (UPotI (projT1 (getEssoRawTC (proj1_sig2 na)))
        (projT2 (getEssoRawTC (proj1_sig2 na)))).

Definition maxCostPred(pmax : Potential)
  (t : {t : ProtoT & sig (CTDTP t)}) : Prop :=
  (∀ u, CT_PD_interp (projT1 t) (proj1_sig (projT2 t)) u
    (proj2_sig (projT2 t)) ≤ pmax) ∧
  (∃ v, CT_PD_interp (projT1 t) (proj1_sig (projT2 t)) v
    (proj2_sig (projT2 t)) = pmax).
Definition maxCostSig(p : Potential) := (sig (maxCostPred p)).
Definition maxCostSigLt(t t′ : sigT maxCostSig) := projT1 t < projT1 t′.
Definition maxCostSigWF := well_founded_ltof (sigT maxCostSig)
  (projT1 (P := maxCostSig)).
Definition costSigDat(t : sigT maxCostSig) := (‘ (projT2 t)).

Record ExprExArgs : Type := {
  recEnv : RssoI;
  wcEnv : sigWssoClosI recEnv;
  exprType : ProtoT;
  exprCost : sig (CTDTP exprType);
  expr : sig (EssoPI recEnv exprType exprCost)
}.
Definition potOfExprArgs(eargs : ExprExArgs ) :=
  CT_PD_interp (exprType eargs).
Definition exprArgsMeas(eargs : ExprExArgs ) : nat :=
  CT_PD_max (exprType eargs) (exprCost eargs).
Definition ExprExArgsWF(eargs : ExprExArgs ) :=
well_founded_ltof
  (ExprExArgs ) (exprArgsMeas ) eargs.
Definition maxCostSigSigT(t : {t : ProtoT & sig (CTDTP t)}) : sigT maxCostSig :=
  existT (P := maxCostSig) (CT_PD_max (projT1 t) (projT2 t))
  (exist (maxCostPred (CT_PD_max (projT1 t) (projT2 t))) t (maxCost_prf t)).
Definition maxCostSigSigTWF := wf_inverse_image
  {t : ProtoT & sig (CTDTP t)} (sigT maxCostSig)
  maxCostSigLt maxCostSigSigT maxCostSigWF.
Lemma maxCostIncl : inclusion {t : ProtoT & sig (CTDTP t)}
  CT_PD_T_lt_pot
  (fun x y : {t : ProtoT & sig (CTDTP t)} ⇒
    maxCostSigLt (maxCostSigSigT x) (maxCostSigSigT y)).
Definition costLtWF := wf_incl {t : ProtoT & sig (CTDTP t)} CT_PD_T_lt_pot
  (fun x y : {t : ProtoT & sig (CTDTP t)} ⇒
    maxCostSigLt (maxCostSigSigT x) (maxCostSigSigT y))
maxCostIncl maxCostSigSigTWF.
Check UBaseDataP.
Lemma CssoPIndentTImpl : ∀ r t t′ c c′ cr,
  EssoPI r t c (essoConstr CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC
    t′ c′ cr) →
  CssoPI r t c cr → CssoPI r t′ c′ cr.
Lemma LTypeDiscrimBaseTup : ∀ ts blt,
  ¬ LTypePSEq (projT1 ts) 1 (buildLTypePSFromTS (projT1 ts) (projT2 ts))
  (BuildBaseTypePS 1 blt eq_refl).
Lemma LTypeDiscrimBaseRec : ∀ tr blt,
  ¬ LTypePSEq (projT1 tr) 1 (buildLTypePSFromRT (projT1 tr) (projT2 tr))
  (BuildBaseTypePS 1 blt eq_refl).
Lemma LTypeDiscrimTupRec : ∀ ts tr, ¬ LTypePSEq (projT1 ts) (projT1 tr)
  (buildLTypePSFromTS (projT1 ts) (projT2 ts))
  (buildLTypePSFromRT (projT1 tr) (projT2 tr)).
Check EssoERIncl.
Implicit Arguments EssoERIncl [r r′ t c er].
Lemma EssoTupInfer : ∀ r ts cs ert t c t′ c′ er cr,
  EssoPI r t c er → er = essoConstr CTDT CTDT_TUP CTDT_REC CTDTP
  CTDTP_TUP CTDTP_REC t′ c′ cr →
  cr = cssoTuple CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC ts cs ert →
  EssoTupPI r ts cs ert.
Lemma EssoRecInfer : ∀ r tr cr erm t c t′ c′ er csr,
  EssoPI r t c er → er = essoConstr CTDT CTDT_TUP CTDT_REC CTDTP
  CTDTP_TUP CTDTP_REC t′ c′ csr →
  csr = cssoRecord CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC tr cr erm
  → EssoRecPI r tr cr erm.

Section redTupS.
  Variable t : ProtoT.
  Variable c : sig (CTDTP t).
  Variable r : RssoI.
  Variable e : sig (EssoPI r t c).
  Variable wc : sigWssoClosI r.
  Let ea := {|
        recEnv := r;
        wcEnv := wc;
        exprType := t;
        exprCost := c;
        expr := e |} : ExprExArgs.
  Variable args : ExprExArgs.
  Variable reduce′ : ∀ args′ : ExprExArgs,
          ltof ExprExArgs exprArgsMeas args′ args →
          UPotI (exprType args′) (exprCost args′).
  Variable recEnv′ : RssoI.
  Variable wcEnv′ : sigWssoClosI recEnv′.
  Variable exprType′ : ProtoT.
  Variable exprCost′ : sig (CTDTP exprType′).
  Variable expr′ : sig (EssoPI recEnv′ exprType′ exprCost′).
  Hypothesis J : args =
      {|
      recEnv := recEnv′;
      wcEnv := wcEnv′;
      exprType := exprType′;
      exprCost := exprCost′;
      expr := expr′ |}.
  Variable er : EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Variable t′ : ProtoT.
  Variable c′ : sig (CTDTP t′).
  Variable cstrct : CssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Hypothesis ep : EssoPI recEnv′ exprType′ exprCost′ er.
  Hypothesis J0 : er =
       essoConstr CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC t′ c′
         cstrct.
  Hypothesis H : CssoPI recEnv′ exprType′ exprCost′ cstrct.
  Section MakeTupS.
    Variable ts : sigT LTypesPS.
    Variable cs : sig (CTDTP_TUP ts).
    Variable ert :
      list (EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC).
    Hypothesis J1 : cstrct =
      cssoTuple CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC ts cs ert.

    Variable newRed : ∀ na : ExprDQualP recEnv′ ts cs,
      UPotI (projT1 (getEssoRawTC (proj1_sig2 na)))
      (projT2 (getEssoRawTC (proj1_sig2 na))).

    Definition MakeTup : UPotI exprType′ exprCost′.

  End MakeTupS.

  Section MakeRecS.

    Variable tr : sigT LRTypesPS.
    Variable cr : sig (CTDTP_REC tr).
    Variable erm : VaridMapModRaw.t
      (EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC).
    Hypothesis J1 : cstrct =
      cssoRecord CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC tr cr erm.
    Variable newRed : ∀ na : ExprDQualPR recEnv′ tr cr,
      UPotI (projT1 (getEssoRawTC (proj1_sig2 na)))
      (projT2 (getEssoRawTC (proj1_sig2 na))).

    Definition MakeRec : UPotI exprType′ exprCost′.

  End MakeRecS.

End redTupS.

Print exprArgsMeas.
Check Acc.

Definition cMeasLt(tc tc′ : {t : ProtoT & sig (CTDTP t)}) :=
  CT_PD_max (projT1 tc)(projT2 tc) < CT_PD_max (projT1 tc′)(projT2 tc′).

    Definition newRedConst(tc tc′ : {t : ProtoT & sig (CTDTP t)})
(reduce′ : ∀ (tc : {t : ProtoT & sig (CTDTP t)}) (r : RssoI),
              sig (EssoPI r (projT1 tc) (projT2 tc)) →
              sigWssoClosI r → Acc cMeasLt tc → UPotI (projT1 tc) (projT2 tc))
(wfp : Acc cMeasLt tc) :
      redfuncA (projT1 tc) (projT1 tc′) (projT2 tc).

    Defined.

Section redHelpersS.

  Variable reduce′ : ∀ (tc : {t : ProtoT & sig (CTDTP t)}) (r : RssoI),
              sig (EssoPI r (projT1 tc) (projT2 tc)) →
              sigWssoClosI r → Acc cMeasLt tc → UPotI (projT1 tc) (projT2 tc).
  Variable tc : {t : ProtoT & sig (CTDTP t)}.
  Variable r : RssoI.
  Variable expr′ : sig (EssoPI r (projT1 tc) (projT2 tc)).
  Variable wc : sigWssoClosI r.
  Hypothesis wfp : Acc cMeasLt tc.
  Variable er : EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Hypothesis ep : EssoPI r (projT1 tc) (projT2 tc) er.
  Hypothesis J : expr′ = exist (EssoPI r (projT1 tc) (projT2 tc)) er ep.

  Section redConstrS.

  Variable t′ : ProtoT.
  Variable c′ : sig (CTDTP t′).
  Variable cstrct : CssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
  Hypothesis J0 : er =
        essoConstr CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC t′ c′
          cstrct.

  Definition redConstructors : UPotI (projT1 tc) (projT2 tc).
  Defined.

  End redConstrS.

  Section redPattS.

    Variable t′ : ProtoT.
    Variable c′ : sig (CTDTP t′).
    Variable p : Patt.
    Hypothesis J0 :
      er = essoPatt CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC t′ c′ p.

    Definition redPatts : UPotI (projT1 tc) (projT2 tc).
```

```
      Defined.
    End redPattS.
    Section redAppS.
      Variable t' : ProtoT.
      Variable c' : sig (CTDTP t').
      Variable er0 : EssoRaw CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC.
      Variable v : String.string.
      Variable J0 : er =
        essoApp CTDT CTDT_TUP CTDT_REC CTDTP CTDTP_TUP CTDTP_REC t' c' er0 v.
      Let expr'' :
        sig (EssoPI r (projT1 (getEssoRawTC er0)) (projT2 (getEssoRawTC er0))).
      Defined.
      Let applicandRed : UPot CTDT CTDTP ICostDT (projT1 (getEssoRawTC er0))
        (projT2 (getEssoRawTC er0)).
      Defined.
      Definition newRed :
        redfuncA (projT1 tc) (projT1 (getEssoRawTC er0)) (projT2 tc).
      Defined.
      Definition redApp : UPotI (projT1 tc) (projT2 tc).
      Defined.
    End redAppS.
  End redHelpersS.
  Fixpoint reduce'(tc : {t : ProtoT & sig (CTDTP t)})(r : RssoI)
    (expr' : sig (EssoPI r (projT1 tc) (projT2 tc)))
    (wc : sigWssoClosI r)(wfp : Acc cMeasLt tc) {struct wfp}
    : (UPotI (projT1 tc) (projT2 tc)) :=
    match expr' as expr' return _ = expr' → _ with
      | exist er ep ⇒ fun J : expr' = exist _ er ep ⇒
        match er as er return _ = er →
          (UPotI (projT1 tc) (projT2 tc)) with
          | essoConstr t' c' cstrct ⇒ fun J0 ⇒
            redConstructors reduce' tc r expr' wc wfp er ep J t' c' cstrct J0
          | essoPatt t' c' p ⇒ fun J0 ⇒
            redPatts reduce' tc r wc wfp er ep t' c' p J0
          | essoApp t' c' er0 v ⇒ fun J0 ⇒
            redApp reduce' tc r expr' wc wfp er ep
              J t' c' er0 v J0
        end eq_refl
    end eq_refl.
  Lemma cMeasLtWf : well_founded cMeasLt.
  Definition reduce(t : ProtoT)(c : sig (CTDTP t))
        (r : RssoI)(e : sig (EssoPI r t c))(wc : sigWssoClosI r) : (UPotI t c).
  Defined.
  End reduceSect.
  Local  Close Scope program_scope.
```

Appendix E

Listings of further Harmonic Box Coordination Language examples

E.1 Gate and fan-out instance library

Listing E.1: The Boolean box library

```
llib boolBoxLib {

  linst andGateInst {

    mem(fb) inAMem :  commonDat.tDatBool64 [64, tfl(0)];
    mem(fb) inBMem :  commonDat.tDatBool64 [64, tfl(0)];
    mem(bf) andOut : commonDat.tDatBool64 [64, tfl(0)];

    observe {
      inAMem;
      inBMem;
    }

    manifest {
      andOut;
    }

    hbox andGate : (inAMem, inBMem) -> andOut [64]
    {
      main : { memInA : (bool); memInB : (bool) }
       -> { andOut : (bool) } :=
         { andOut = (and memInA.0 memInB.0) };
    }
  }

  linst orGateInst {

    mem(fb) inAMem :  commonDat.tDatBool64 [64, tfl(0)];
    mem(fb) inBMem :  commonDat.tDatBool64 [64, tfl(0)];
    mem(bf) orOut : commonDat.tDatBool64 [64, tfl(0)];

    observe {
      inAMem;
      inBMem;
  }

  manifest {
    orOut;
  }

  hbox orGate : (inAMem, inBMem) -> orOut [64]
  {
    main : { memInA : (bool); memInB : (bool) }
     -> { orOut : (bool) } :=
       { orOut = (or memInA.0 memInB.0) };
  }
}

linst xorGateInst {

  mem(fb) inAMem :  commonDat.tDatBool64 [64, tfl(0)];
  mem(fb) inBMem :  commonDat.tDatBool64 [64, tfl(0)];
  mem(bf) xorOut : commonDat.tDatBool64 [64, tfl(0)];

  observe {
    inAMem;
    inBMem;
  }

  manifest {
    xorOut;
  }

  hbox xorGate : (inAMem, inBMem) -> xorOut [64]
  {
    main : { memInA : (bool); memInB : (bool) }
     -> { xorOut : (bool) } :=
       { xorOut = (xor memInA.0 memInB.0) };
  }
}

linst fanout2Inst {

  mem(fb) inMem :  commonDat.tDatBool64 [64, tfl(0)];
  mem(bf) fanoutA : commonDat.tDatBool64 [64, tfl(0)];
  mem(bf) fanoutB : commonDat.tDatBool64 [64, tfl(0)];

  observe {
    inMem;
  }

  manifest {
    fanoutA;
    fanoutB;
  }

  hbox fanout2 : (inMem) -> (fanoutA, fanoutB) [64]
  {
    main : { memIn : (bool) }
     -> { fanoutA : (bool); fanoutB : (bool) } :=
       { fanoutA = (memIn.0); fanoutB = (memIn.0) };
  }
  }

  linst fanout3Inst {

    mem(fb) inMem :  commonDat.tDatBool64 [64, tfl(0)];
    mem(bf) fanoutA : commonDat.tDatBool64 [64, tfl(0)];
    mem(bf) fanoutB : commonDat.tDatBool64 [64, tfl(0)];
    mem(bf) fanoutC : commonDat.tDatBool64 [64, tfl(0)];

    observe {
      inMem;
    }

    manifest {
      fanoutA;
      fanoutB;
      fanoutC;
    }

    hbox fanout3 : (inMem) -> (fanoutA, fanoutB, fanoutC) [64]
    {
      main : { memIn : (bool) }
       -> { fanoutA : (bool); fanoutB : (bool); fanoutC : (bool) } :=
         { fanoutA = (memIn.0); fanoutB = (memIn.0); fanoutC = (memIn.0) };
    }
  }
}
```

E.2 Boolean source and sink instance library

Listing E.2: The source and sink library

```
llib sourceSinkLib {

  linst source0Inst {

    mem(bf) bOut : commonDat.tDatBool64 [64, tfl(0)];

    observe {

    }

    manifest {
      bOut;
    }

    hbox source0 : () -> bOut [64]
    {
      main : { }
       -> { bOut : (bool) } := { bOut = (false) };
    }
  }

  linst source1Inst {

    mem(bf) bOut : commonDat.tDatBool64 [64, tfl(0)];

    observe {

    }

    manifest {
      bOut;
    }

    hbox source1 : () -> bOut [64]
    {
      main : { }
       -> { bOut : (bool) } := { bOut = (true) };
    }
  }

  linst sinkInst {

    mem(fb) inMem :  commonDat.tDatBool64 [64, tfl(0)];

    observe {
      inMem;
    }

    manifest {

    }

    hbox orGate : (inMem) -> () [64]
    {
      main : { memIn : (bool) }
       -> { } := { };
    }
  }
}
```

E.3 Boolean voter library

Listing E.3: The voter library

```
llib votersLib {

  linst voter3Inst {

    mem(fb) inMemA :  commonDat.tDatBool64 [64, tfl(0)];
    mem(fb) inMemB :  commonDat.tDatBool64 [64, tfl(0)];
    mem(fb) inMemC :  commonDat.tDatBool64 [64, tfl(0)];
    mem(bf) voter3Out : commonDat.tDatBool64 [64, tfl(0)];

    observe {
      inMemA;
      inMemB;
      inMemC;
    }

    manifest {
      voter3Out;
    }

    hbox voter3 : (inMemA, inMemB, inMemC) -> (voter3Out) [64]
    {
      main : { memInA : (bool); memInB : (bool); memInC : (bool); }
       -> { voter3Out : (bool) } :=
         { voter3Out = (or
         (or (and memInA.0 memInB.0) (and memInB.0 memInC.0))
         (and memInA.0 memInC.0 )) };
    }
  }
}
```

E.4 Coq code for negator example

We now provide the complete hand-compiled code for the negator example. The harmonic box interface file is necessary to avoid a universe inconsistency, which is a limitation of Coq's current type and module system variant.

Listing E.4: The negator harmonic box interface object

```
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HBox_SBLT.


Require Import HBCL.Util.Freq.
Require Import HBCL.HBCL_0_1.Examples.libs.commonDat.
Import String.
Definition freqMapIn : Freq := freq64.
Definition freqMapOut : Freq := freq64.
Print HBCL_0_1_L_HBox.MDataInst.MDatBoxElt.
Print HBCL_0_1_L_HBox.MDataInst.MDatBoxFreqEltBase.
Program Definition posInMemOid : HBCL_0_1_Oid_S.HBCL_OidMemFB :=
  ((”posIn”)%string :: (”MemFB”)%string :: nil)%list.
Obligation 1.
Admitted.
Program Definition posInMemVarid : HBCL_0_1_Id_S.Varid :=
  (”posIn”)%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.
Require Import Coq.ZArith.ZArith_base.
  Definition posInMem : HBCL_0_1_L_HBox.MDataInst.MDatBoxElt :=
    HBCL_0_1_L_HBox.MDataInst.Build_MDatBoxFreqEltBase
    (proj1_sig posInMemOid) freqMapIn tDatDoublePair (0)%Z (0)%N (2)%positive.
  Definition negatorBoxInMemMap :
    HBCL_0_1_Id_S.VaridMapMod.t HBCL_0_1_L_HBox.MDataInst.MDatBoxElt
    := HBCL_0_1_Id_S.VaridMapMod.add posInMemVarid posInMem
    (HBCL_0_1_Id_S.VaridMapMod.empty _).

Program Definition negOutMemOid : HBCL_0_1_Oid_S.HBCL_OidMemBF :=
  ((”negOut”)%string :: (”MemBF”)%string :: nil)%list.
Obligation 1.
Admitted.

Program Definition negOutMemVarid : HBCL_0_1_Id_S.Varid :=
  (”negOut”)%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

  Definition negOutMem : HBCL_0_1_L_HBox.MDataInst.MDatBoxElt :=
    HBCL_0_1_L_HBox.MDataInst.Build_MDatBoxFreqEltBase
    (proj1_sig negOutMemOid) freqMapOut tDatTriple (-2)%Z (0)%N (2)%positive.

  Definition negatorBoxOutMemMap :
    HBCL_0_1_Id_S.VaridMapMod.t HBCL_0_1_L_HBox.MDataInst.MDatBoxElt
    := HBCL_0_1_Id_S.VaridMapMod.add negOutMemVarid negOutMem
    (HBCL_0_1_Id_S.VaridMapMod.empty _).

Print HBCL_0_1_L_HBox.HBoxSSO.

Definition negatorBoxOverallFreq := freq64.
```

Listing E.5: The negator untimed box object

```
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UBoxEmptyEnc_SB.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.exprLang.
Require Import HBCL.HBCL_0_1.Examples.libs.commonDat.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorDatTempify.
Require Import HBCL.HBCL_0_1.BaseLibs.ExprLangs.bitLang.builtIn.

Import sigTypes.
Import String.

Program Definition posInId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  (”posIn”)%string.
Obligation 1.
Admitted.

Obligation 2.
Qed.

Program Definition negOutId : sig HBCL_0_1_Id_S.varidPredType.Pred :=
  (”negOut”)%string.
Obligation 1.
Admitted.

Obligation 2.
Qed.

Definition posInCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang doublePairProt).
Defined.

Definition negOutCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang uDatTripleProt).
Defined.

Definition negSSOInMap : HBCL_0_1_Id_S.VaridMapMod.t
     ({t : HBCL_0_1_L_UTS.ProtoT &
     {o : HBCL_0_1_Oid_S.HBCL_OidUT & HBCL_0_1_L_UTSOid.T o t} &
     sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)} × nat × nat) :=
let posIn :=
((existT2 doublePairProt
  (existT (P := fun o ⇒ HBCL_0_1_L_UTSOid.T o doublePairProt)
    (projTD1 uDatDoublePairOidT) (projTD3 uDatDoublePairOidT))
  posInCost), 1, 1)
in
HBCL_0_1_Id_S.VaridMapMod.add posInId posIn (HBCL_0_1_Id_S.VaridMapMod.empty _).

Definition negSSOOutMap : HBCL_0_1_Id_S.VaridMapMod.t
    ({t : HBCL_0_1_L_UTS.ProtoT &
    {o : HBCL_0_1_Oid_S.HBCL_OidUT & HBCL_0_1_L_UTSOid.T o t} &
    sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)} × nat × nat)
:=
let negOut :=
((existT2 uDatTripleProt
  (existT (P := fun o ⇒ HBCL_0_1_L_UTSOid.T o uDatTripleProt)
    (projTD1 uDatTripleOidT) (projTD3 uDatTripleOidT))
  negOutCost), 1, 1)
in
HBCL_0_1_Id_S.VaridMapMod.add negOutId negOut
(HBCL_0_1_Id_S.VaridMapMod.empty _).

Definition posInTypeBucketTuple := bitPairPairStrong.

Definition posInTypeBucket : HBCL_0_1_L_UTS.ProtoT := doublePairProt.

Definition posInBucketTrainTupRaw := (existT (projT1 posInTypeBucket)
    (proj1_sig (projT2 posInTypeBucket)) :: nil)%list.

Program Definition posInTypeBucketTupleStrong :
  HBCL_0_1_L_UTS.LTypesPS 3 := (existT (projT1 posInTypeBucket)
    (proj1_sig (projT2 posInTypeBucket)) :: nil)%list.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

Program Definition posInTypeBucketTrain : HBCL_0_1_L_UTS.ProtoT :=
  existT 3 (HBCL_0_1_L_UTS.LTupleType 3 posInBucketTrainTupRaw).
Obligation 1.
Admitted.

Program Definition mainInType : HBCL_0_1_L_UTS.ProtoT :=
  existT 4 (HBCL_0_1_L_UTS.LRecordType 4
    (HBCL_0_1_Id_S.VaridMapMod.this
      (
        HBCL_0_1_Id_S.VaridMapMod.add posInId
        (existT (projT1 posInTypeBucketTrain)
          (proj1_sig (projT2 posInTypeBucketTrain)))
        (HBCL_0_1_Id_S.VaridMapMod.empty _)
      )
    )
  ).
Obligation 1.
Admitted.

Definition mainOutTypeBucketTuple := uDatTripleTupStrong.

Definition mainOutTypeBucket : HBCL_0_1_L_UTS.ProtoT := uDatTripleProt.

Program Definition mainOutTypeBucketC : {t : HBCL_0_1_L_UTS.ProtoT &
      sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)} :=
existT mainOutTypeBucket 10.
Obligation 1.
Admitted.

Definition bucketTrainTupRaw := (existT (projT1 mainOutTypeBucket)
    (proj1_sig (projT2 mainOutTypeBucket)) :: nil)%list.

Program Definition mainOutTypeTupleStrong :
  HBCL_0_1_L_UTS.LTypesPS 3 := (existT (projT1 mainOutTypeBucket)
    (proj1_sig (projT2 mainOutTypeBucket)) :: nil)%list.
Obligation 1.
Admitted.
Obligation 2.
Admitted.

Program Definition mainOutTypeBucketTrain : HBCL_0_1_L_UTS.ProtoT :=
  existT 3 (HBCL_0_1_L_UTS.LTupleType 3 bucketTrainTupRaw).
```

```
Obligation 1.
Admitted.

Program Definition mainOutTypeBucketTrainC : {t : HBCL_0_1_L_UTS.ProtoT &
          sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)} :=
existT mainOutTypeBucketTrain 12.
Obligation 1.
Admitted.

Program Definition mainOutTypeRecord :
  HBCL_0_1_L_UTS.LRTypesPS 4 :=
 HBCL_0_1_Id_S.VaridMapMod.this
 (
    HBCL_0_1_Id_S.VaridMapMod.add negOutId
    (existT (projT1 mainOutTypeBucketTrain)
       (proj1_sig (projT2 mainOutTypeBucketTrain)))
    (HBCL_0_1_Id_S.VaridMapMod.empty _)
 ).
Obligation 1.
Admitted.

Obligation 2.
Admitted.

Check HBCL_0_1_L_UTS.LRecordType.
Check proj1_sig mainOutTypeRecord.
Program Definition mainOutType : HBCL_0_1_L_UTS.ProtoT :=
  existT 4 (HBCL_0_1_L_UTS.LRecordType 4
     (HBCL_0_1_Id_S.VaridMapMod.this
        (
           HBCL_0_1_Id_S.VaridMapMod.add negOutId
           (existT (projT1 mainOutTypeBucketTrain)
              (proj1_sig (projT2 mainOutTypeBucketTrain)))
           (HBCL_0_1_Id_S.VaridMapMod.empty _)
        )
     )
  ).
Obligation 1.
Admitted.

Program Definition mainOutTypeC : {t : HBCL_0_1_L_UTS.ProtoT &
          sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang t)} :=
existT mainOutType 13.
Obligation 1.
Admitted.

Definition mainTsso :=
  TypeSSO.TssoGenFunc (CTDTP := HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang)
  mainId mainInType mainOutTypeC.

Definition negEssoR : TypeSSO.Rsso instBTSCostBase :=
  HBCL_0_1_Id_S.VaridMapMod.add mainId mainTsso RssoWithNotAndXorFunc.

Program Definition costIn : sig (CTDTPtriv mainInType) := 1.
Obligation 1.
Admitted.


Print ExprSSO.PattEl.
Print ExprSSO.PattElP.

Program Definition pattElsA : sig
  (ExprSSO.PattElsP (projT1 mainInType) 1 (projT2 mainInType) BaseTypeBool) :=
  (ExprSSO.pattVarid posInId ::
     ExprSSO.pattPosParam 0 :: ExprSSO.pattPosParam 0 :: nil)%list.
Obligation 1.
Admitted.

Program Definition pattElsB : sig
  (ExprSSO.PattElsP (projT1 mainInType) 1 (projT2 mainInType) BaseTypeBool) :=
  (ExprSSO.pattVarid posInId ::
     ExprSSO.pattPosParam 0 :: ExprSSO.pattPosParam 1 ::
     ExprSSO.pattPosParam 0 :: nil)%list.
Obligation 1.
```

```
Admitted.
Program Definition pattElsC : sig
  (ExprSSO.PattElsP (projT1 mainInType) 1 (projT2 mainInType) BaseTypeBool) :=
  (ExprSSO.pattVarid posInId ::
    ExprSSO.pattPosParam 0 :: ExprSSO.pattPosParam 1 ::
    ExprSSO.pattPosParam 1 :: nil)%list.
Obligation 1.
Admitted.
Program Definition BoolLookupCost :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang (existT 1 BaseTypeBool)) :=
  1.
Obligation 1.
Admitted.
Program Definition pattA : sig
  (ExprSSO.PattP _ _ (ICostDT := instBTSCostBase)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT (existT mainInType (minCtriv mainInType)))
      RssoWithNotFunc) (existT 1 BaseTypeBool) BoolLookupCost) :=
  ExprSSO.patt mainArgId (proj1_sig pattElsA).
Obligation 1.
Admitted.
Program Definition pattB : sig
  (ExprSSO.PattP _ _ (ICostDT := instBTSCostBase)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT (existT mainInType (minCtriv mainInType)))
      RssoWithNotFunc) (existT 1 BaseTypeBool) BoolLookupCost) :=
  ExprSSO.patt mainArgId (proj1_sig pattElsB).
Obligation 1.
Admitted.
Program Definition pattC : sig
  (ExprSSO.PattP _ _ (ICostDT := instBTSCostBase)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT (existT mainInType (minCtriv mainInType)))
      RssoWithNotFunc) (existT 1 BaseTypeBool) BoolLookupCost) :=
  ExprSSO.patt mainArgId (proj1_sig pattElsC).
Obligation 1.
Admitted.
Definition exprA : sig
  (ExprSSO.EssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT (existT mainInType (minCtriv mainInType)))
      RssoWithNotFunc) (existT 1 BaseTypeBool) BoolLookupCost) :=
  exist _ _
  (ExprSSO.essoPattP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (proj1_sig pattA)
    (existT 1 BaseTypeBool) BoolLookupCost
    (TypeSSO.R2Contains_refl _ _ _ _)
    (proj2_sig pattA)).
Definition exprB : sig
  (ExprSSO.EssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT (existT mainInType (minCtriv mainInType)))
      RssoWithNotFunc) (existT 1 BaseTypeBool) BoolLookupCost) :=
  exist _ _
  (ExprSSO.essoPattP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (proj1_sig pattB)
    (existT 1 BaseTypeBool) BoolLookupCost
    (TypeSSO.R2Contains_refl _ _ _ _)
    (proj2_sig pattB)).

Definition exprC : sig
  (ExprSSO.EssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT (existT mainInType (minCtriv mainInType)))
      RssoWithNotFunc) (existT 1 BaseTypeBool) BoolLookupCost) :=
  exist _ _
  (ExprSSO.essoPattP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (HBCL_0_1_Id_S.VaridMapMod.add mainArgId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (proj1_sig pattC)
    (existT 1 BaseTypeBool) BoolLookupCost
    (TypeSSO.R2Contains_refl _ _ _ _)
    (proj2_sig pattC)).

Program Definition BoolNotCostApp :
  sig (HBCL_0_1_L_UBoxEmtpy.CTDTP UBitLang (existT 1 BaseTypeBool)) :=
  3.
Obligation 1.
Admitted.

Program Definition notA :
  sig (ExprSSO.EssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add posInId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (existT 1 BaseTypeBool) BoolNotCostApp) :=
  ExprSSO.essoApp _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  (existT 1 BaseTypeBool) BoolNotCostApp (proj1_sig exprA)
  (proj1_sig (proj1_sig notId)).
Obligation 1.
Admitted.

Program Definition notB :
  sig (ExprSSO.EssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add posInId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (existT 1 BaseTypeBool) BoolNotCostApp) :=
  ExprSSO.essoApp _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  (existT 1 BaseTypeBool) BoolNotCostApp (proj1_sig exprB)
  (proj1_sig (proj1_sig notId)).
Obligation 1.
Admitted.

Program Definition notC :
  sig (ExprSSO.EssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    (HBCL_0_1_Id_S.VaridMapMod.add posInId
      (TypeSSO.TssoGenDataT
        (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
    (existT 1 BaseTypeBool) BoolNotCostApp) :=
  ExprSSO.essoApp _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  (existT 1 BaseTypeBool) BoolNotCostApp (proj1_sig exprC)
  (proj1_sig (proj1_sig notId)).
Obligation 1.
Admitted.

Program Definition outBucketConstructCost :
  sig (CTDTPtrivT (existT 3 mainOutTypeBucketTuple)) :=
  9.
Obligation 1.
Admitted.

Program Definition outBucketConstr :
  sig (ExprSSO.CssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  (HBCL_0_1_Id_S.VaridMapMod.add posInId
    (TypeSSO.TssoGenDataT
      (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
  mainOutTypeBucket (projT2 mainOutTypeBucketC))
  := ExprSSO.cssoTuple _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
(existT 3 mainOutTypeBucketTuple) outBucketConstructCost
((proj1_sig notA) :: (proj1_sig notB) :: (proj1_sig notC) :: nil)%list.
Obligation 1.
Admitted.

Program Definition outBucketExpr : sig (ExprSSO.EssoP _ _ _ _ _ _
  (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  (HBCL_0_1_Id_S.VaridMapMod.add posInId
    (TypeSSO.TssoGenDataT
      (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
  (projT1 mainOutTypeBucketC) (projT2 mainOutTypeBucketC)) :=
ExprSSO.essoConstr _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
(projT1 mainOutTypeBucketC) (projT2 mainOutTypeBucketC)
(proj1_sig outBucketConstr).
Obligation 1.
Admitted.

Program Definition OutBucketTrainConstructCost :
  sig (CTDTPtrivT (existT 3 mainOutTypeTupleStrong)) :=
  11.
Obligation 1.
Admitted.

Program Definition outBucketTrainConstr :
  sig (ExprSSO.CssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  (HBCL_0_1_Id_S.VaridMapMod.add posInId
    (TypeSSO.TssoGenDataT
      (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
  mainOutTypeBucket (projT2 mainOutTypeBucketTrainC))
  := ExprSSO.cssoTuple _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
(existT 3 mainOutTypeTupleStrong) OutBucketTrainConstructCost
((proj1_sig outBucketExpr) :: nil)%list.
Obligation 1.
Admitted.

Obligation 2.
Admitted.

Program Definition outBucketTrainExpr : sig (ExprSSO.EssoP _ _ _ _ _ _
  (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  (HBCL_0_1_Id_S.VaridMapMod.add posInId
    (TypeSSO.TssoGenDataT
      (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
  (projT1 mainOutTypeBucketTrainC) (projT2 mainOutTypeBucketTrainC)) :=
ExprSSO.essoConstr _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
(projT1 mainOutTypeBucketTrainC) (projT2 mainOutTypeBucketTrainC)
(proj1_sig outBucketTrainConstr).

Obligation 1.
Admitted.
```

```
Program Definition OutRecordConstructCost :
  sig (CTDTPtrivR (existT _ mainOutTypeRecord)) := 12.
Obligation 1.
Admitted.

Program Definition outConstr :
  sig (ExprSSO.CssoP _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  (HBCL_0_1_Id_S.VaridMapMod.add posInId
    (TypeSSO.TssoGenDataT
      (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
  (projT1 mainOutTypeC) (projT2 mainOutTypeC))
  := ExprSSO.cssoRecord _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  (existT _ mainOutTypeRecord) OutRecordConstructCost
  (HBCL_0_1_Id_S.VaridMapMod.this (HBCL_0_1_Id_S.VaridMapMod.add
    negOutId (proj1_sig outBucketTrainExpr)
    (HBCL_0_1_Id_S.VaridMapMod.empty _))).
Obligation 1.
Admitted.

Program Definition outExpr : sig (ExprSSO.EssoP _ _ _ _ _ _
  (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  (HBCL_0_1_Id_S.VaridMapMod.add posInId
    (TypeSSO.TssoGenDataT
      (existT mainInType (minCtriv mainInType))) RssoWithNotFunc)
  (projT1 mainOutTypeC) (projT2 mainOutTypeC)) :=
ExprSSO.essoConstr _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
(projT1 mainOutTypeC) (projT2 mainOutTypeC)
(proj1_sig outConstr).
Obligation 1.
Admitted.

Lemma rssoIncl : TypeSSO.R2ContainsR1 RssoWithNotFunc negEssoR.

Print ExprSSO.FssoRaw.
Print ExprSSO.FssoP.

Check ExprSSO.FssoEFunc _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
  (ICostDT := instBTSCostBase) minCtriv negEssoR posInId (projT1 mainOutTypeC)
  mainInType (minCtriv _) (proj1_sig outExpr).

Print ExprSSO.FssoEFunc.
Check ExprSSO.FssoDat.

Definition mainFssoRaw :=
(ExprSSO.FssoEFunc _ _ _ CTDTPtriv CTDTPtrivT CTDTPtrivR
(ICostDT := instBTSCostBase) minCtriv negEssoR mainArgId (projT1 mainOutTypeC)
mainInType (minCtriv _) (proj1_sig outExpr)).

Print ExprSSO.Fsso.

Program Definition mainFsso :=
  sigTypes.existTD (P := ExprSSO.Fsso _ _ _ _ _ _ _)
  negEssoR mainTsso
  (exist (ExprSSO.FssoP _ _ _ _ _ _
    (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord) minCtriv
    negEssoR mainTsso)
  mainFssoRaw
  _).
Obligation 1.
Admitted.

Definition FssoMapMainFunc := HBCL_0_1_Id_S.VaridMapMod.add mainId mainFsso
  FssoMapNotFunc.

Program Definition negEssoWFInR : sig
  (ExprSSO.FssoMapWFInR _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv negEssoR) := FssoMapMainFunc.
Obligation 1.
Admitted.

Program Definition negEssoWC :
  ExprSSO.sigWssoClos _ _ _ _ _ _ (ICostDT := instBTSCostBase)
  (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
  minCtriv negEssoR := ((negEssoR, existT negEssoR negEssoWFInR) :: nil)%list.
Obligation 1.
Admitted.
Obligation 2.
Admitted.
Obligation 3.
Admitted.
Lemma funcExtractPredMain : RedAppBranch.funcExtractionPred
  CTDTtriv CTDTPtriv negEssoR mainOutType mainInType
 (projT2 mainOutTypeC) costIn mainId.
Lemma inIOTypeMatch :
  exprLang.OidTypeIOMatch CTDTtriv CTDTPtriv negSSOInMap mainInType.
Lemma outIOTypeMatch :
  exprLang.OidTypeIOMatch CTDTtriv CTDTPtriv negSSOOutMap mainOutType.
Check exprLang.Make_sso.
Definition negSSO :
  HBCL_0_1_L_UBoxEmtpy.sso UBitLang negSSOInMap negSSOOutMap :=
  (exprLang.Make_sso (ICostDT := instBTSCostBase)
    (ICostDTupT := instBTSCostTuple) (ICostDRecT := instBTSCostRecord)
    _ _ _ _ _ _ minCtriv negSSOInMap negSSOOutMap mainOutType mainInType
    (projT2 mainOutTypeC) costIn mainId negEssoR negEssoWC)
  funcExtractPredMain
  inIOTypeMatch
  outIOTypeMatch.
Definition computeFuncInst :=
  HBCL_0_1_L_UBoxEmtpy.reduce UBitLang negSSOInMap negSSOOutMap negSSO.
Check computeFuncInst.

Check proj1_sig (projT2 doublePairProt).
Program Definition posInBucketInstance :
  sig (HBCL_0_1_L_UTS.UDataP doublePairProt ) :=
  outerPair (true, (true, true)).
Obligation 1.
Admitted.
Definition posInBucketTrainTuple :=
  HBCL_0_1_L_UTS.UTupleData _ posInTypeBucketTuple
  (proj1_sig posInBucketInstance :: nil).
Program Definition samplePosInBucketData :
  sig (HBCL_0_1_L_UTS.UDataP posInTypeBucketTrain) := posInBucketTrainTuple.
Obligation 1.
Admitted.
Definition udatIn :
  HBCL_0_1_Id_S.VaridMapMod.t (sigT HBCL_0_1_L_UTS.UDataPST) :=
  HBCL_0_1_Id_S.VaridMapMod.add posInId
  (existT posInTypeBucketTrain samplePosInBucketData)
  (HBCL_0_1_Id_S.VaridMapMod.empty _).
Check HBCL_0_1_L_UBoxEmtpy.UDataPSTMatchesInpOutpTypes.
Program Definition sampleDatMatches :
  sig (HBCL_0_1_L_UBoxEmtpy.UDataPSTMatchesInpOutpTypes
    CTDTtriv CTDTPtriv negSSOInMap) :=
  udatIn.
Obligation 1.
Admitted.
Check computeFuncInst.
```

Listing E.6: The negator harmonic box data binder object

```
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HBox_SBLT.
Require Import HBCL.Util.Freq.
Require Import HBCL.HBCL_0_1.Examples.libs.commonDat.
Import HBCL_0_1_L_HBox.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorHIface.

Program Definition baseTypeGen(b : bool) :
  sig (HBCL_0_1_L_UTS.UDataP (existT 1 BaseTypeBool)) :=
  HBCL_0_1_L_UTS.UBaseData HBCL_0_1_L_UTS.BasetypeBool
  (HBCL_0_1_L_UTS.UBTBool HBCL_0_1_L_UTS.BasetypeBool eq_refl b).
Obligation 1.
Admitted.

Section InDatSliceEmbedS.

  Variable indatCoq : (bool × (bool × bool)).
  Variable tTimeTDatDoublePair : TTime (sigTypes.projTT2 tDatDoublePair).
  Variable posInMemOid : HBCL_0_1_Oid_S.HBCL_OidMemFB.

  Definition innerPair := HBCL_0_1_L_UTS.UTupleData 2 pairTupStrong
    (proj1_sig (baseTypeGen (fst (snd indatCoq))) ::
      proj1_sig (baseTypeGen (snd (snd indatCoq))) :: nil)%list.

  Definition outerPair := HBCL_0_1_L_UTS.UTupleData 3 bitPairPairStrong
    (proj1_sig (baseTypeGen (fst indatCoq)) :: innerPair :: nil)%list.

  Definition posInBucketInstance :
    sig (HBCL_0_1_L_UTS.UDataP doublePairProt).
  Defined.

  Definition posInTimedVInstance : HBCL_0_1_L_HTS.TimedV
    tDatDoublePair tTimeTDatDoublePair :=
    HBCL_0_1_L_HTS.MakeTimedV tDatDoublePair tTimeTDatDoublePair
    (Some posInBucketInstance).

  Definition posInTimeDatInstance :
    HBCL_0_1_L_HBox.MDataInst.MDatTimeElt :=
    ((sigTypes.existTD _ _ posInTimedVInstance :: nil)%list).

  Definition timeSliceRaw : HBCL_0_1_L_HBox.InMemModBox.MDatTimeMapRaw :=
  HBCL_0_1_L_HBox.InMemModBox.otm.add posInMemOid posInTimeDatInstance
  (HBCL_0_1_L_HBox.InMemModBox.otm.empty _).

End InDatSliceEmbedS.
```

Listing E.7: The negator harmonic box object

```
Require Import HBCL.Util.Freq.
Require Import HBCL.HBCL_0_1.BaseLibs.BoxLangs.bitExprBoxLang.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HBox_SBLT.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorUbox.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorHIface.

Definition negHBoxRaw :=
  HBCL_0_1_L_HBox.HBoxSSORaw_make HBoxUBitLang negSSOInMap negSSOOutMap negSSO.


Program Definition HBoxSSOMemSpec : HBCL_0_1_L_HBox.HBoxSSO
  negatorBoxOverallFreq freqMapIn freqMapOut
  negatorBoxInMemMap negatorBoxOutMemMap :=
  negHBoxRaw.
Obligation 1.
Admitted.
```

Listing E.8: The negator coordination object

```
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HBox_SBLT.
```

```
Import HBCL_0_1_L_HBox.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.CoordInterp1_SBLT.
Import HBCL_0_1_L_Coord.
Require Import HBCL.HBCL_0_1.Examples.libs.commonDat.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorHIface.
Require Import HBCL.Util.Freq.
Require Import Coq.QArith.QArith_base.
Close Scope Q_scope.
Require Import Coq.Program.Program.
Import String.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UTypeSysOid_SB.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HTypeSys_SB.

Section negCoordS.
  Variable HBoxSSOMemSpec : HBoxSSO
  negatorBoxOverallFreq freqMapIn freqMapOut
  negatorBoxInMemMap negatorBoxOutMemMap.
  Definition inMemMapIO : InMemModInst.MDatFreqMapIO freqMapIn.
  Defined.
  Definition outMemMapIO : OutMemModInst.MDatFreqMapIO freqMapOut.
  Defined.
  Definition negatorLInstFreq := freq64.
  Definition negatorClosSigRaw :
    HBCL_0_1_Oid_S.LInstMapMod.t LInstSignatureRaw :=
    HBCL_0_1_Oid_S.LInstMapMod.empty _.
  Program Definition negatorLInstSig : LInstSignature negatorLInstFreq :=
    {| InstSigFreqMemIn := freqMapIn;
      InstSigFreqMemOut := freqMapIn;
      InstSigInputMems := inMemMapIO;
      InstSigOutputMems := outMemMapIO
    |}.
  Obligation 1.
  Admitted.
  Program Definition freqNul : Freq := MakeFreq (0 # 1)%Q.
Obligation 1.
Admitted.
  Program Definition EmptyNestMapIn : InMemModInst.MDatFreqMap freqNul :=
    InMemModBox.otm.empty MDataTypeInst.MDatFreqElt.
Obligation 1.
Admitted.
  Program Definition EmptyNestMapOut : OutMemModInst.MDatFreqMap freqNul :=
    OutMemModBox.otm.empty MDataTypeInst.MDatFreqElt.
Obligation 1.
Admitted.
 Program Definition negatorBoxid : HBCL_0_1_Id_S.Boxid :=
    ("negator")%string.
Obligation 1.
Admitted.
Obligation 2.
Admitted.
  Definition negatorHBoxMap : HBCL_0_1_L_HBox.BoxTypeIdMapMod.t
    (HBoxSSONonDep × (InMemModBox.otm.t
      HBCL_0_1_Id_S.Varid)
      × (HBCL_0_1_Id_S.VaridMapMod.t HBCL_0_1_Oid_S.HBCL_OidMemBF)) :=
    let negatorNonDep : HBoxSSONonDep :=
      {|
    HBoxSSONonDep_f := _;
   HBoxSSONonDep_fi := _;
    HBoxSSONonDep_fo := _;
    HBoxSSONonDep_ttmfIn := _;
    HBoxSSONonDep_ttmfOut := _;
    HBoxSSONonDep_HBoxSSO := HBoxSSOMemSpec
    |}
      in
```

```
          let inmemvarmap : InMemModBox.otm.t
            HBCL_0_1_Id_S.Varid
            := InMemModBox.otm.add posInMemOid posInMemVarid
              (InMemModBox.otm.empty _)
              in
              let varoutmemmap :
                HBCL_0_1_Id_S.VaridMapMod.t HBCL_0_1_Oid_S.HBCL_OidMemBF
                := HBCL_0_1_Id_S.VaridMapMod.add negOutMemVarid negOutMemOid
                (HBCL_0_1_Id_S.VaridMapMod.empty _)

            in
            BoxTypeIdMapMod.add negatorBoxid
            (negatorNonDep, inmemvarmap, varoutmemmap)
            (BoxTypeIdMapMod.empty _).
      Definition negatorObsMap :
        InMemModBox.otm.t
    HBCL_0_1_Oid_S.HBCL_OidMemFB :=
      InMemModBox.otm.add posInMemOid posInMemOid

        (InMemModBox.otm.empty _).
      Definition negatorManifMap :
        OutMemModBox.otm.t
        HBCL_0_1_Oid_S.HBCL_OidMemBF :=
      OutMemModBox.otm.add negOutMemOid negOutMemOid

        (OutMemModBox.otm.empty _).
      Definition negatorFIFOMap : HBCL_0_1_Id_S.VaridMapMod.t
        (HBCL_0_1_Oid_S.HBCL_OidMemBF × HBCL_0_1_Oid_S.HBCL_OidMemFB) :=
        HBCL_0_1_Id_S.VaridMapMod.empty _.
      Definition negatorNestedEmbeddedLInstMap : HBCL_0_1_Oid_S.LInstMapMod.Raw.t
        LInstSSORaw := HBCL_0_1_Oid_S.LInstMapMod.this
        ( HBCL_0_1_Oid_S.LInstMapMod.empty LInstSSORaw).
      Definition negatorNestedLLibLInstMap :
         HBCL_0_1_Oid_S.LInstMapMod.t HBCL_0_1_Oid_S.HBCL_OidLInst :=
         HBCL_0_1_Oid_S.LInstMapMod.empty _.
     Definition negatorNestedExternalLLibInstEnvMap :
       HBCL_0_1_Oid_S.LInstMapMod.t
       (sigT LInstSignature × HBCL_0_1_Oid_S.LInstMapMod.t LInstSignatureRaw) :=
       HBCL_0_1_Oid_S.LInstMapMod.empty _.
     Definition negatorLocalLibMap : HBCL_0_1_Oid_S.LLibMapMod.Raw.t LibSSORaw :=
       HBCL_0_1_Oid_S.LLibMapMod.this (HBCL_0_1_Oid_S.LLibMapMod.empty _).
     Definition negatorLocalLibEnv : HBCL_0_1_Oid_S.LLibMapMod.t
             (HBCL_0_1_Oid_S.LInstMapMod.t LInstSignatureRaw) :=
             HBCL_0_1_Oid_S.LLibMapMod.empty _.
     Definition negatorNestedInstTShift : HBCL_0_1_Oid_S.LInstMapMod.t TTFL :=
       HBCL_0_1_Oid_S.LInstMapMod.empty _.
      Program Definition negatorLInstSSO : LInstSSO negatorLInstFreq negatorLInstSig
        negatorClosSigRaw :=
        LInstSSORaw_make freqMapIn freqMapOut freqMapIn freqMapOut
      freqNul freqNul negatorLInstFreq freqNul negatorLInstFreq
      (' inMemMapIO)
      (' outMemMapIO)
      EmptyNestMapIn
      EmptyNestMapOut
      negatorHBoxMap
      negatorObsMap
      negatorManifMap
      negatorFIFOMap
      negatorNestedEmbeddedLInstMap
      negatorNestedLLibLInstMap
      negatorNestedExternalLLibInstEnvMap
      negatorLocalLibMap
      negatorNestedInstTShift
      negatorLocalLibEnv.
```

```
  Obligation 1.
    Admitted.
  Program Definition negatorInstOid : HBCL_0_1_Oid_S.HBCL_OidLInst :=
    (("negatorInst")%string :: ("LInst")%string :: nil)%list.
  Obligation 1.
  Admitted.
  Definition negatorLLibInstSigMap := HBCL_0_1_Oid_S.LInstMapMod.add
    negatorInstOid ('negatorLInstSig) (HBCL_0_1_Oid_S.LInstMapMod.empty _).
  Definition negatorLLibInstMap := HBCL_0_1_Oid_S.LInstMapMod.add
    negatorInstOid ('negatorLInstSSO) (HBCL_0_1_Oid_S.LInstMapMod.empty _).
  Definition negatorLibInstSigMap :
  HBCL_0_1_Oid_S.LInstMapMod.t
    (sigT LInstSignature × HBCL_0_1_Oid_S.LInstMapMod.t LInstSignatureRaw) :=
    HBCL_0_1_Oid_S.LInstMapMod.add negatorInstOid
    (existT _ negatorLInstSig, negatorClosSigRaw)
    (HBCL_0_1_Oid_S.LInstMapMod.empty _).
  Definition negatorLibLibInstSigMap : HBCL_0_1_Oid_S.LLibMapMod.t
    (HBCL_0_1_Oid_S.LInstMapMod.t LInstSignatureRaw) :=
     HBCL_0_1_Oid_S.LLibMapMod.empty _.
  Program Definition negatorLLibSSO : LLibSSO negatorLLibInstSigMap :=
    LLibSSORaw_make
    (HBCL_0_1_Id_S.VaridMapMod.empty _)
    (HBCL_0_1_Id_S.VaridMapMod.empty _)
    negatorLLibInstMap
    negatorLibInstSigMap
    negatorLocalLibMap
    negatorLibLibInstSigMap.
  Obligation 1.
  Admitted.
  Program Definition negatorLibMap : LibClos negatorClosSigRaw := nil.
  Obligation 1.
  Admitted.
End negCoordS.
```

Listing E.9: The negator initial state object

```
Require Import HBCL.HBCL_0_1.BaseLibs.UTypeSystems.bitTSys.BFUTypeSys.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HBox_SBLT.
Require Import HBCL.Util.Freq.
Require Import HBCL.HBCL_0_1.Examples.libs.commonDat.
Import HBCL_0_1_L_HBox.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorCoord.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorHIface.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.Coord_SBLT.
Import HBCL_0_1_L_Coord.
Definition tZero : TTime negatorLInstFreq :=
  C_Time _ 0.
Definition posInMemEl : MDataInst.MDatTimeElt := nil.
Definition negOutMemEl : MDataInst.MDatTimeElt := nil.
Definition inMemMapNul : InMemModBox.MDatTimeMapRaw :=
InMemModBox.otm.add posInMemOid posInMemEl
(InMemModBox.otm.empty _).
Definition outMemMapNul : OutMemModBox.MDatTimeMapRaw :=
OutMemModBox.otm.add negOutMemOid negOutMemEl
(OutMemModBox.otm.empty _).
Definition initStateRaw : CoordStateRaw :=
  CoordStateRaw_make inMemMapNul outMemMapNul
  (HBCL_0_1_Oid_S.LInstMapMod.this
    (HBCL_0_1_Oid_S.LInstMapMod.empty _)).
Section negatorCStateInitS.
  Variable HBoxSSOMemSpec : HBoxSSO
    negatorBoxOverallFreq freqMapIn freqMapOut
    negatorBoxInMemMap negatorBoxOutMemMap.
  Let negatorLInstSSOLoc := negatorLInstSSO HBoxSSOMemSpec.
  Definition initState : sig
    (CoordStateInnerFIFOsEnabled negatorLInstFreq tZero
      negatorLInstSig negatorClosSigRaw negatorLibMap
      negatorLInstSSOLoc).
  Defined.
End negatorCStateInitS.
```

Listing E.10: The negator test harness

```
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HBox_SBLT.
Import HBCL_0_1_L_HBox.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.CoordInterp1_SBLT.
Import HBCL_0_1_L_Coord.
Import HBCL_0_1_L_CoordInterp1.
Require Import HBCL.HBCL_0_1.Examples.libs.commonDat.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorDatTempify.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorInitState.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorHIface.
Require Import HBCL.Util.Freq.
Require Import Coq.QArith.QArith_base.
Close Scope Q_scope.
Require Import Coq.Program.Program.
Import String.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.UTypeSysOid_SB.
Require Import HBCL.HBCL_0_1.Instances.bitLangRFreq1.HTypeSys_SB.
Require Import HBCL.HBCL_0_1.Examples.libs.negator.negatorCoord.

Section negCoordS.
  Variable HBoxSSOMemSpec : HBoxSSO
  negatorBoxOverallFreq freqMapIn freqMapOut
  negatorBoxInMemMap negatorBoxOutMemMap.
  Let negatorLInstSSOLoc := negatorLInstSSO HBoxSSOMemSpec.
  Section calcResultS.
    Definition tZero : TTime negatorLInstFreq :=
      C_Time _ 0.
    Definition tiZero : TTime (InstSigFreqMemIn (‘negatorLInstSig)) :=
      C_Time _ 0.
    Lemma tTiZeroEq : TTseq tZero tiZero.
    Lemma tTiNextCeil : nextCeil (proj1 (proj2_sig negatorLInstSig)) tZero
      tiZero.
    Variable inpStream : InputStream (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) tiZero.
    Definition resultTrace := traceFIFOsGenOutpInit _ _ _ negatorLibMap
      negatorLInstSSOLoc tZero tiZero tTiZeroEq inpStream tTiNextCeil
      (initState HBoxSSOMemSpec).
  End calcResultS.
Print InMemModInst.MDatMapFreqTimePred.
Print HBCL_0_1_L_HBox.InMemModBox.MDatTimeMapRaw.
Print HBCL_0_1_L_HBox.MDataInst.MDatTimeElt.
Print HBCL_0_1_L_HBox.MDataInst.MemDatListRaw.
Print HBCL_0_1_L_HTS.TimedV.
Print HBCL_0_1_L_HTS.TimedVLocal.
```

```
  Definition sampleSlice0FormOK :
    sig (InMemModInst.MDatMapFreqTimePred (InstSigFreqMemIn (‘negatorLInstSig))
    (‘ (InstSigInputMems (‘negatorLInstSig)))).
  Defined.

  Definition sampleSlice0 :
    sig (InMemModInst.MDatMapModeReadPred (InstSigFreqMemIn (‘negatorLInstSig))
     (C_Time _ 0) (‘ (InstSigInputMems (‘negatorLInstSig)))).
  Defined.

  Definition sampleSlice1FormOK :
    sig (InMemModInst.MDatMapFreqTimePred (InstSigFreqMemIn (‘negatorLInstSig))
    (‘ (InstSigInputMems (‘negatorLInstSig)))).
  Defined.

  Definition sampleSlice1 :
    sig (InMemModInst.MDatMapModeReadPred (InstSigFreqMemIn (‘negatorLInstSig))
     (C_Time _ 1) (‘ (InstSigInputMems (‘negatorLInstSig)))).
  Defined.

  Definition sampleSlice2FormOK :
    sig (InMemModInst.MDatMapFreqTimePred (InstSigFreqMemIn (‘negatorLInstSig))
    (‘ (InstSigInputMems (‘negatorLInstSig)))).
  Defined.

  Definition sampleSlice2 :
    sig (InMemModInst.MDatMapModeReadPred (InstSigFreqMemIn (‘negatorLInstSig))
     (C_Time _ 2) (‘ (InstSigInputMems (‘negatorLInstSig)))).
  Defined.

  Definition InStream3 : InputStream (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 3) :=
    InputStreamFinal _ _ _.

  Lemma tnext2_3 : TTseq
          (tNext (InstSigFreqMemIn (‘negatorLInstSig))
              (C_Time (InstSigFreqMemIn (‘negatorLInstSig)) 2))
          (C_Time (InstSigFreqMemIn (‘negatorLInstSig)) 3).

  Definition InStream2 : InputStream (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 2) :=
    InputStreamInd (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 2) (C_Time _ 3)
    tnext2_3 sampleSlice2 InStream3.

  Lemma tnext1_2 : TTseq
          (tNext (InstSigFreqMemIn (‘negatorLInstSig))
              (C_Time (InstSigFreqMemIn (‘negatorLInstSig)) 1))
          (C_Time (InstSigFreqMemIn (‘negatorLInstSig)) 2).

  Definition InStream1 : InputStream (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 1) :=
    InputStreamInd (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 1) (C_Time _ 2)
    tnext1_2 sampleSlice1 InStream2.

  Lemma tnext0_1 : TTseq
          (tNext (InstSigFreqMemIn (‘negatorLInstSig))
              (C_Time (InstSigFreqMemIn (‘negatorLInstSig)) 0))
          (C_Time (InstSigFreqMemIn (‘negatorLInstSig)) 1).

  Definition InStream0 : InputStream (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 0) :=
    InputStreamInd (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) (C_Time _ 0) (C_Time _ 1)
    tnext0_1 sampleSlice0 InStream1.

  Definition sampleInStream : InputStream (InstSigFreqMemIn (‘negatorLInstSig))
    (InstSigInputMems (‘negatorLInstSig)) tiZero
    := InStream0.

End negCoordS.

Check resultTrace.
```

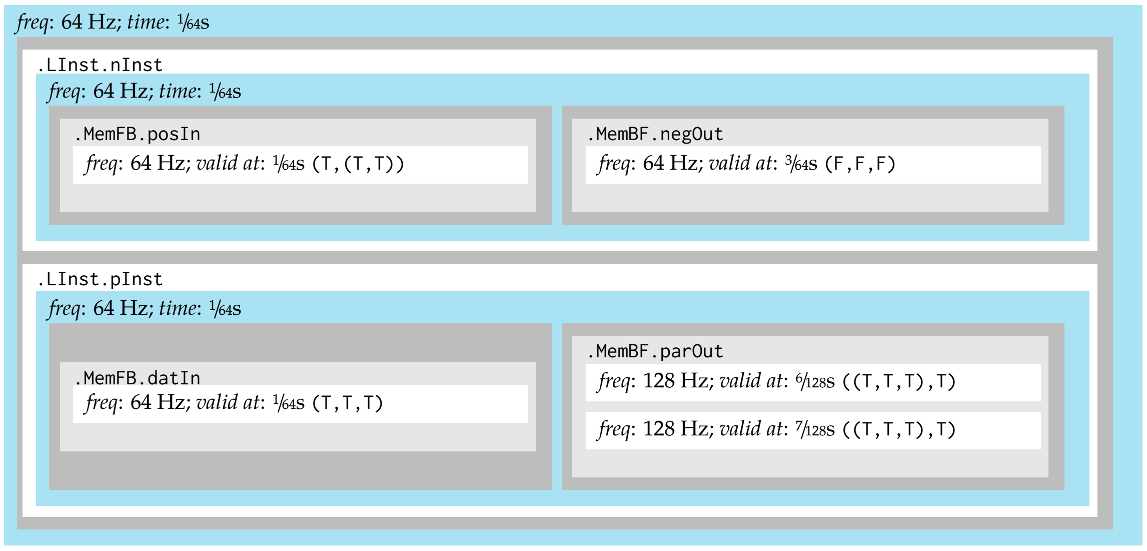
Appendix F

Further Harmonic Box Coordination Language execution traces

F.1 Parallel composition

Listing F.1 is the continuation of Listing 6.3.

Listing F.1(i): The parallel composition of a negator and parity box



freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.nInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.posIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s (T,(T,T))
.MemBF.negOut
freq: 64 Hz; *valid at*: $\frac{3}{64}$s (F,F,F)
.LInst.pInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.datIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s (T,T,T)
.MemBF.parOut
freq: 128 Hz; *valid at*: $\frac{6}{128}$s ((T,T,T),T)
freq: 128 Hz; *valid at*: $\frac{7}{128}$s ((T,T,T),T)

Listing F.1(ii): The parallel composition of a negator and parity box

freq: 64 Hz; *time*: 1/64s

```
.LInst.nInst
  freq: 64 Hz; time: 1/64s
    .MemFB.posIn
      freq: 64 Hz; valid at: 1/64s (T,(T,T))
    .MemBF.negOut
      freq: 64 Hz; valid at: 3/64s (F,F,F)
.LInst.pInst
  freq: 64 Hz; time: 1/64s
    .MemFB.datIn
      freq: 64 Hz; valid at: 1/64s (T,T,T)
    .MemBF.parOut
      freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
      freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
```

Listing F.1(iii): The parallel composition of a negator and parity box

freq: 64 Hz; *time*: 1/64s

```
.LInst.nInst
  freq: 64 Hz; time: 1/64s
    .MemFB.posIn
      freq: 64 Hz; valid at: 1/64s (T,(T,T))
    .MemBF.negOut
      freq: 64 Hz; valid at: 3/64s (F,F,F)
.LInst.pInst
  freq: 64 Hz; time: 1/64s
    .MemFB.datIn
      freq: 64 Hz; valid at: 1/64s (T,T,T)
    .MemBF.parOut
      freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
      freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
```

Listing F.1(iv): The parallel composition of a negator and parity box

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.LInst.nInst`

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,(F,F))`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,T,T)`

`.LInst.pInst`

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.MemFB.datIn`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,F,F)`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,T,T)`

`.MemBF.parOut`

freq: 128 Hz; *valid at*: $^{6}/_{128}$s `((T,T,T),T)`

freq: 128 Hz; *valid at*: $^{7}/_{128}$s `((T,T,T),T)`

freq: 128 Hz; *valid at*: $^{8}/_{128}$s `((T,F,F),T)`

freq: 128 Hz; *valid at*: $^{9}/_{128}$s `((T,F,F),T)`

Listing F.1(v): The parallel composition of a negator and parity box

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.LInst.nInst`

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,(F,F))`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,T,T)`

`.LInst.pInst`

freq: 64 Hz; *time*: $^{2}/_{64}$s

`.MemFB.datIn`

freq: 64 Hz; *valid at*: $^{2}/_{64}$s `(T,F,F)`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `(T,T,T)`

`.MemBF.parOut`

freq: 128 Hz; *valid at*: $^{6}/_{128}$s `((T,T,T),T)`

freq: 128 Hz; *valid at*: $^{7}/_{128}$s `((T,T,T),T)`

freq: 128 Hz; *valid at*: $^{8}/_{128}$s `((T,F,F),T)`

freq: 128 Hz; *valid at*: $^{9}/_{128}$s `((T,F,F),T)`

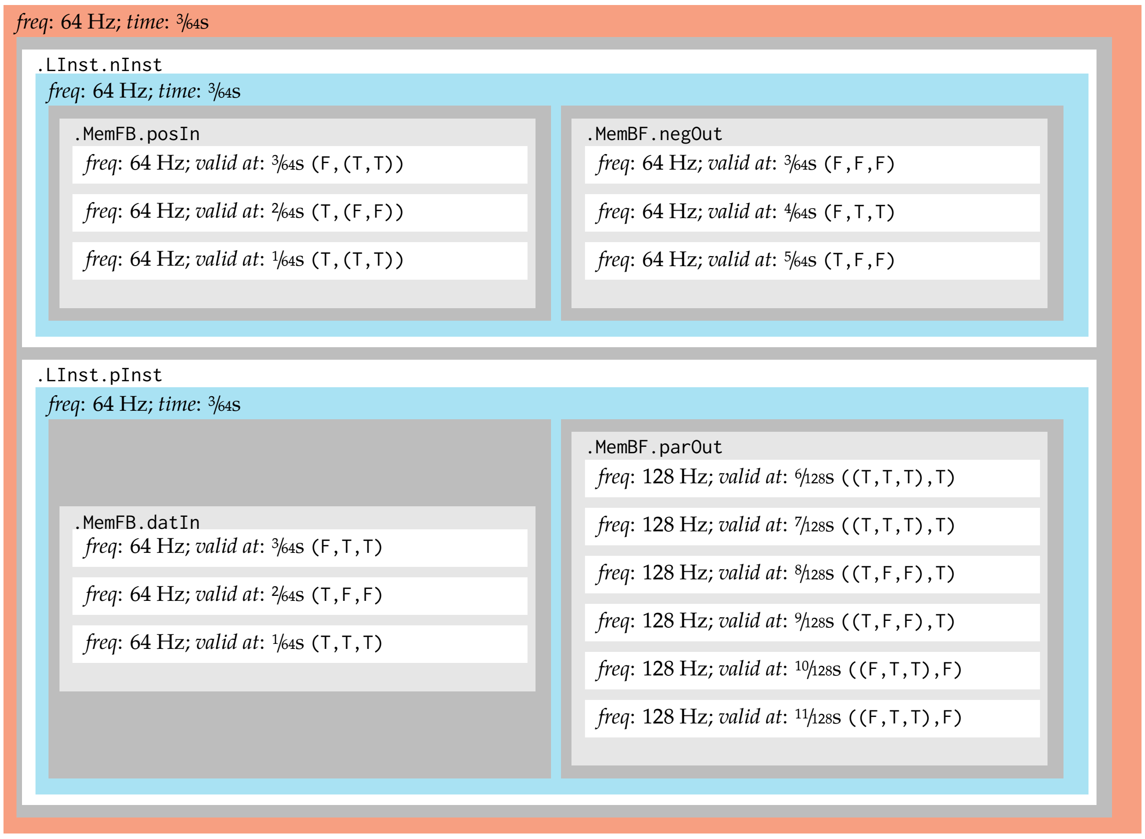
Listing F.1(vi): The parallel composition of a negator and parity box

```
freq: 64 Hz; time: 2/64s
  .LInst.nInst
    freq: 64 Hz; time: 2/64s
      .MemFB.posIn
        freq: 64 Hz; valid at: 2/64s (T,(F,F))
        freq: 64 Hz; valid at: 1/64s (T,(T,T))
      .MemBF.negOut
        freq: 64 Hz; valid at: 3/64s (F,F,F)
        freq: 64 Hz; valid at: 4/64s (F,T,T)
  .LInst.pInst
    freq: 64 Hz; time: 2/64s
      .MemFB.datIn
        freq: 64 Hz; valid at: 2/64s (T,F,F)
        freq: 64 Hz; valid at: 1/64s (T,T,T)
      .MemBF.parOut
        freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
        freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
        freq: 128 Hz; valid at: 8/128s ((T,F,F),T)
        freq: 128 Hz; valid at: 9/128s ((T,F,F),T)
```

Listing F.1(vii): The parallel composition of a negator and parity box

```
freq: 64 Hz; time: 2/64s
  .LInst.nInst
    freq: 64 Hz; time: 2/64s
      .MemFB.posIn
        freq: 64 Hz; valid at: 2/64s (T,(F,F))
        freq: 64 Hz; valid at: 1/64s (T,(T,T))
      .MemBF.negOut
        freq: 64 Hz; valid at: 3/64s (F,F,F)
        freq: 64 Hz; valid at: 4/64s (F,T,T)
  .LInst.pInst
    freq: 64 Hz; time: 2/64s
      .MemFB.datIn
        freq: 64 Hz; valid at: 2/64s (T,F,F)
        freq: 64 Hz; valid at: 1/64s (T,T,T)
      .MemBF.parOut
        freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
        freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
        freq: 128 Hz; valid at: 8/128s ((T,F,F),T)
        freq: 128 Hz; valid at: 9/128s ((T,F,F),T)
```

Listing F.1(viii): The parallel composition of a negator and parity box


freq: 64 Hz; *time*: $^{3}/_{64}$s
.LInst.nInst
freq: 64 Hz; *time*: $^{3}/_{64}$s
.MemFB.posIn
freq: 64 Hz; *valid at*: $^{3}/_{64}$s (F,(T,T))
freq: 64 Hz; *valid at*: $^{2}/_{64}$s (T,(F,F))
freq: 64 Hz; *valid at*: $^{1}/_{64}$s (T,(T,T))
.MemBF.negOut
freq: 64 Hz; *valid at*: $^{3}/_{64}$s (F,F,F)
freq: 64 Hz; *valid at*: $^{4}/_{64}$s (F,T,T)
freq: 64 Hz; *valid at*: $^{5}/_{64}$s (T,F,F)
.LInst.pInst
freq: 64 Hz; *time*: $^{3}/_{64}$s
.MemFB.datIn
freq: 64 Hz; *valid at*: $^{3}/_{64}$s (F,T,T)
freq: 64 Hz; *valid at*: $^{2}/_{64}$s (T,F,F)
freq: 64 Hz; *valid at*: $^{1}/_{64}$s (T,T,T)
.MemBF.parOut
freq: 128 Hz; *valid at*: $^{6}/_{128}$s ((T,T,T),T)
freq: 128 Hz; *valid at*: $^{7}/_{128}$s ((T,T,T),T)
freq: 128 Hz; *valid at*: $^{8}/_{128}$s ((T,F,F),T)
freq: 128 Hz; *valid at*: $^{9}/_{128}$s ((T,F,F),T)
freq: 128 Hz; *valid at*: $^{10}/_{128}$s ((F,T,T),F)
freq: 128 Hz; *valid at*: $^{11}/_{128}$s ((F,T,T),F)

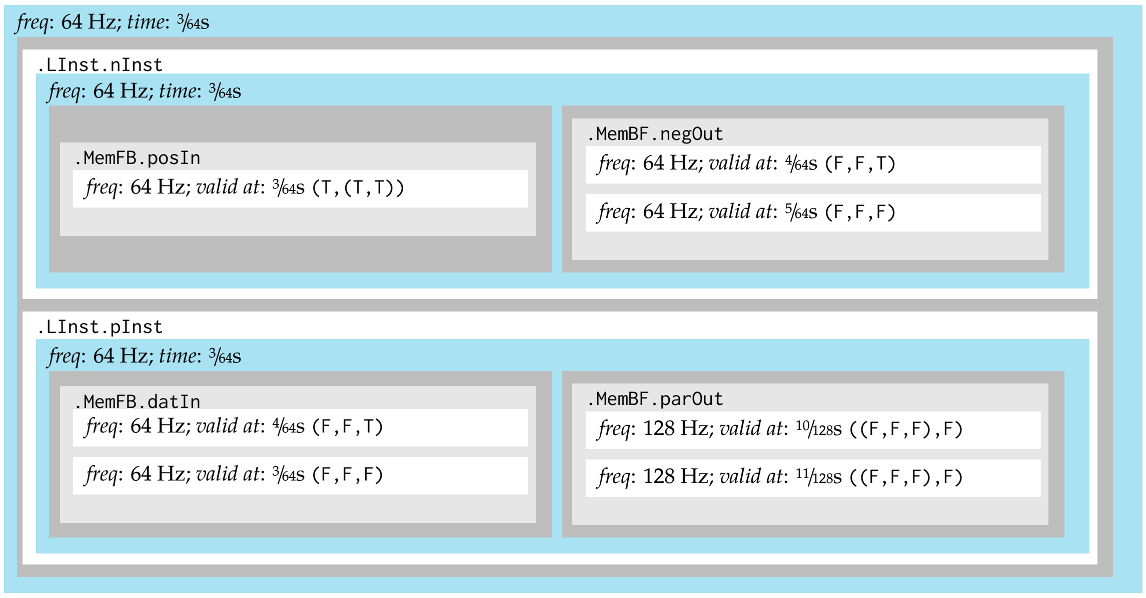
Listing F.1(ix): The parallel composition of a negator and parity box

```
freq: 64 Hz; time: 3/64s
  .LInst.nInst
  freq: 64 Hz; time: 3/64s
    .MemFB.posIn
      freq: 64 Hz; valid at: 3/64s (F,(T,T))
      freq: 64 Hz; valid at: 2/64s (T,(F,F))
      freq: 64 Hz; valid at: 1/64s (T,(T,T))
    .MemBF.negOut
      freq: 64 Hz; valid at: 3/64s (F,F,F)
      freq: 64 Hz; valid at: 4/64s (F,T,T)
      freq: 64 Hz; valid at: 5/64s (T,F,F)
  .LInst.pInst
  freq: 64 Hz; time: 3/64s
    .MemFB.datIn
      freq: 64 Hz; valid at: 3/64s (F,T,T)
      freq: 64 Hz; valid at: 2/64s (T,F,F)
      freq: 64 Hz; valid at: 1/64s (T,T,T)
    .MemBF.parOut
      freq: 128 Hz; valid at: 6/128s ((T,T,T),T)
      freq: 128 Hz; valid at: 7/128s ((T,T,T),T)
      freq: 128 Hz; valid at: 8/128s ((T,F,F),T)
      freq: 128 Hz; valid at: 9/128s ((T,F,F),T)
      freq: 128 Hz; valid at: 10/128s ((F,T,T),F)
      freq: 128 Hz; valid at: 11/128s ((F,T,T),F)
```

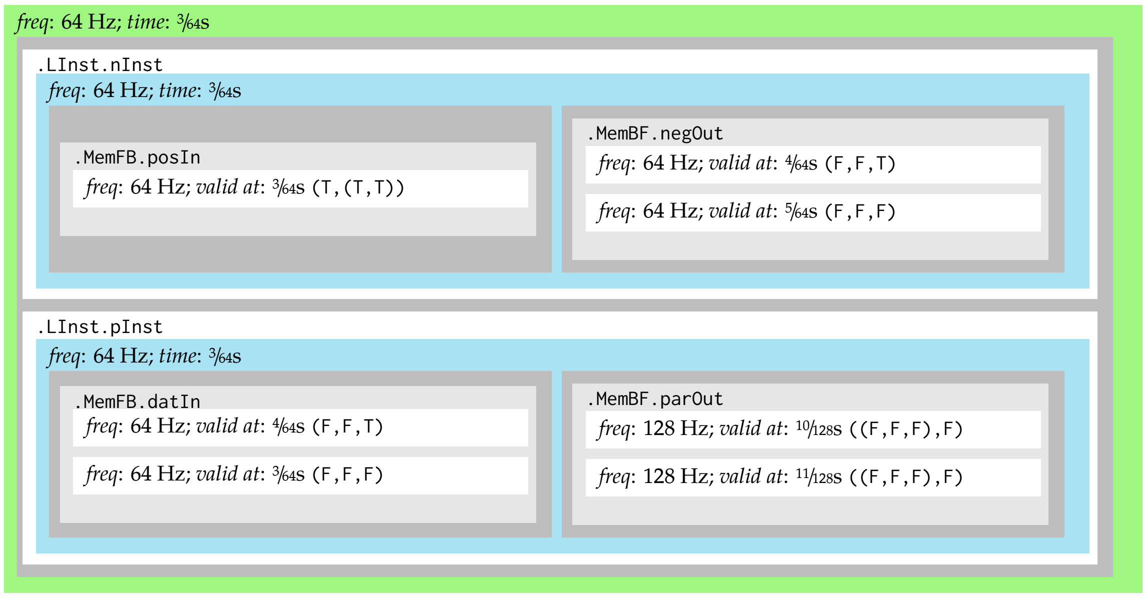
F.2 Simple pipeline

Listing F.2 is the continuation of Listing 6.4.

Listing F.2(i): The pipeline example


freq: 64 Hz; *time*: 3/64s
.LInst.nInst
freq: 64 Hz; *time*: 3/64s
.MemFB.posIn
freq: 64 Hz; *valid at*: 3/64s (T,(T,T))
.MemBF.negOut
freq: 64 Hz; *valid at*: 4/64s (F,F,T)
freq: 64 Hz; *valid at*: 5/64s (F,F,F)
.LInst.pInst
freq: 64 Hz; *time*: 3/64s
.MemFB.datIn
freq: 64 Hz; *valid at*: 4/64s (F,F,T)
freq: 64 Hz; *valid at*: 3/64s (F,F,F)
.MemBF.parOut
freq: 128 Hz; *valid at*: 10/128s ((F,F,F),F)
freq: 128 Hz; *valid at*: 11/128s ((F,F,F),F)

Listing F.2(ii): The pipeline example


freq: 64 Hz; *time*: 3/64s
.LInst.nInst
freq: 64 Hz; *time*: 3/64s
.MemFB.posIn
freq: 64 Hz; *valid at*: 3/64s (T,(T,T))
.MemBF.negOut
freq: 64 Hz; *valid at*: 4/64s (F,F,T)
freq: 64 Hz; *valid at*: 5/64s (F,F,F)
.LInst.pInst
freq: 64 Hz; *time*: 3/64s
.MemFB.datIn
freq: 64 Hz; *valid at*: 4/64s (F,F,T)
freq: 64 Hz; *valid at*: 3/64s (F,F,F)
.MemBF.parOut
freq: 128 Hz; *valid at*: 10/128s ((F,F,F),F)
freq: 128 Hz; *valid at*: 11/128s ((F,F,F),F)

Listing F.2(iii): The pipeline example

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.LInst.nInst`

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`

freq: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`

`.LInst.pInst`

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.MemFB.datIn`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

`.MemBF.parOut`

freq: 128 Hz; *valid at*: $^{10}/_{128}$s `((F,F,F),F)`

freq: 128 Hz; *valid at*: $^{11}/_{128}$s `((F,F,F),F)`

Listing F.2(iv): The pipeline example

freq: 64 Hz; *time*: $^{4}/_{64}$s

`.LInst.nInst`

freq: 64 Hz; *time*: $^{4}/_{64}$s

`.MemFB.posIn`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(T,(F,F))`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(T,(T,T))`

`.MemBF.negOut`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`

freq: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`

freq: 64 Hz; *valid at*: $^{6}/_{64}$s `(F,T,T)`

`.LInst.pInst`

freq: 64 Hz; *time*: $^{4}/_{64}$s

`.MemFB.datIn`

freq: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`

freq: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`

`.MemBF.parOut`

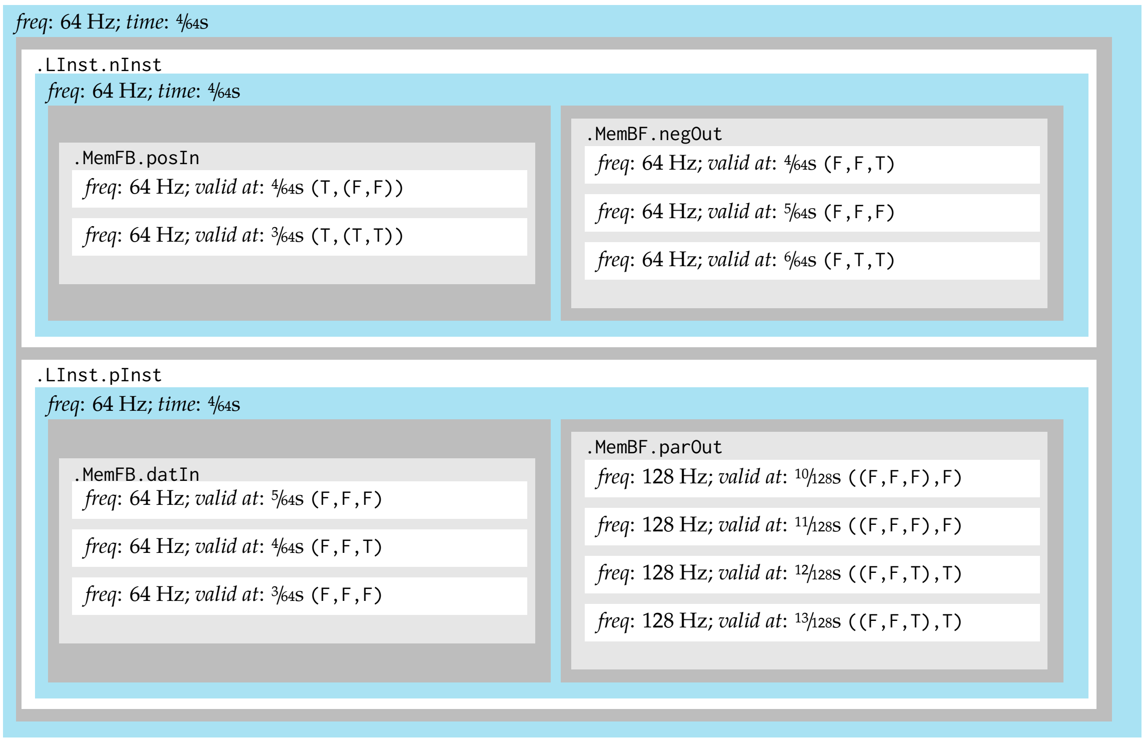
freq: 128 Hz; *valid at*: $^{10}/_{128}$s `((F,F,F),F)`

freq: 128 Hz; *valid at*: $^{11}/_{128}$s `((F,F,F),F)`

freq: 128 Hz; *valid at*: $^{12}/_{128}$s `((F,F,T),T)`

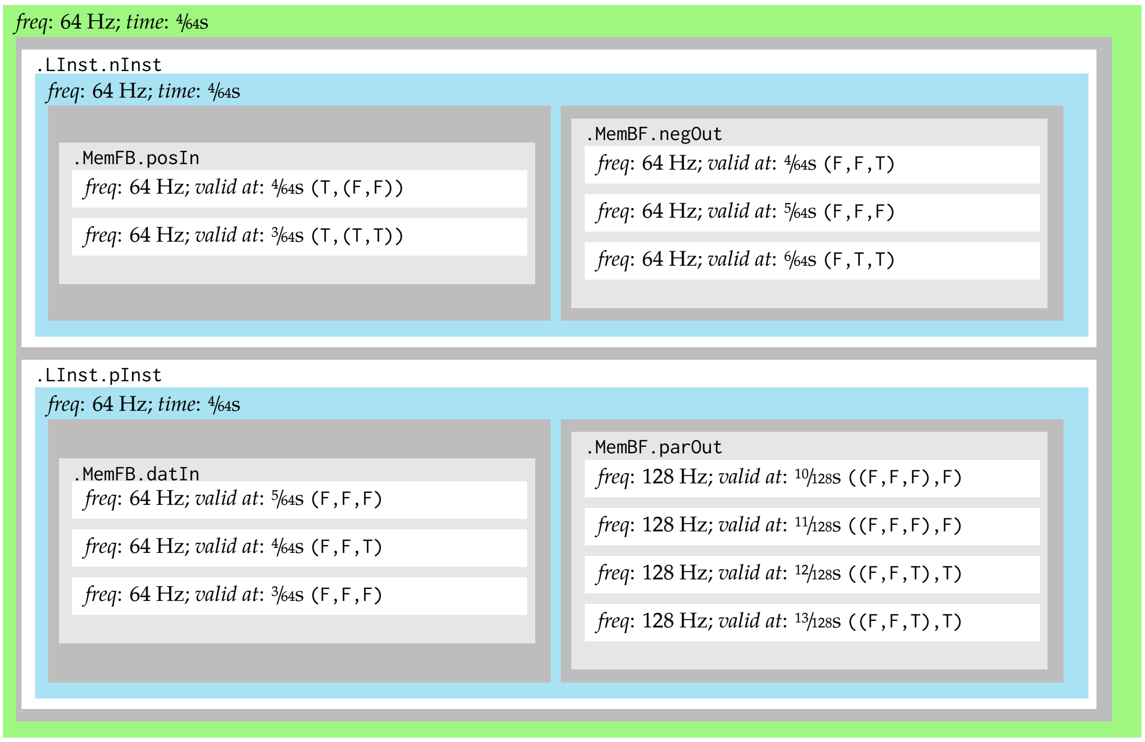
freq: 128 Hz; *valid at*: $^{13}/_{128}$s `((F,F,T),T)`

Listing F.2(v): The pipeline example



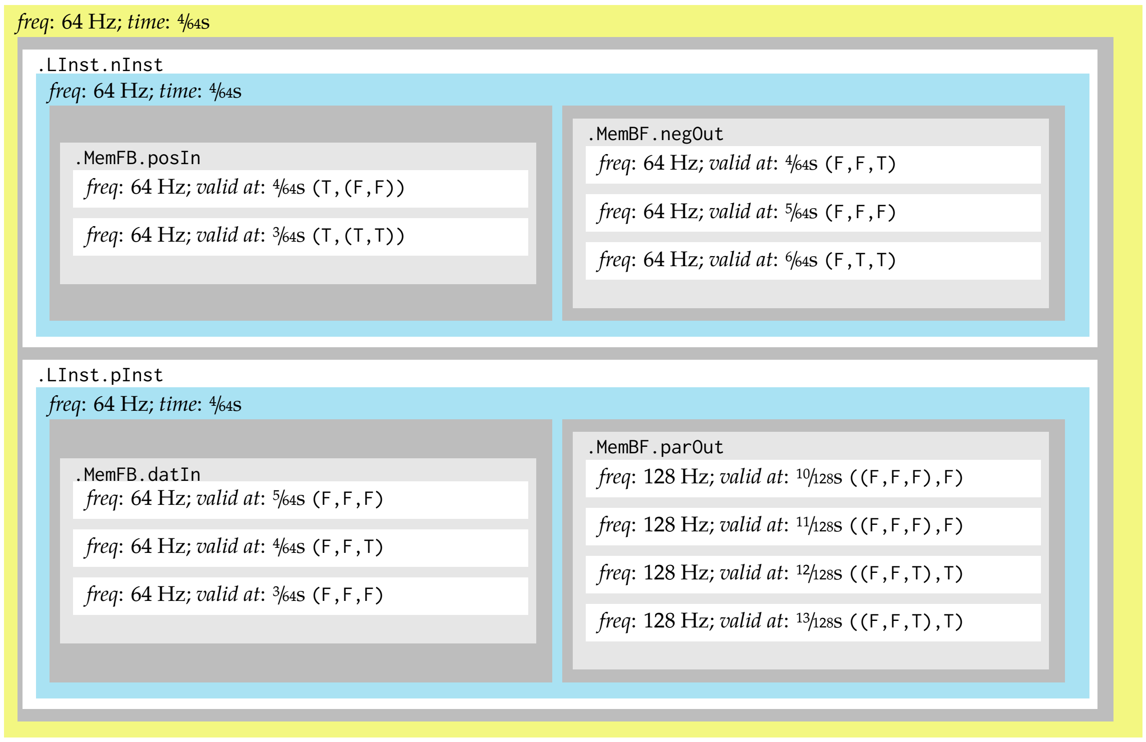
```
freq: 64 Hz; time: 4/64s
  .LInst.nInst
    freq: 64 Hz; time: 4/64s
      .MemFB.posIn
        freq: 64 Hz; valid at: 4/64s (T,(F,F))
        freq: 64 Hz; valid at: 3/64s (T,(T,T))
      .MemBF.negOut
        freq: 64 Hz; valid at: 4/64s (F,F,T)
        freq: 64 Hz; valid at: 5/64s (F,F,F)
        freq: 64 Hz; valid at: 6/64s (F,T,T)
  .LInst.pInst
    freq: 64 Hz; time: 4/64s
      .MemFB.datIn
        freq: 64 Hz; valid at: 5/64s (F,F,F)
        freq: 64 Hz; valid at: 4/64s (F,F,T)
        freq: 64 Hz; valid at: 3/64s (F,F,F)
      .MemBF.parOut
        freq: 128 Hz; valid at: 10/128s ((F,F,F),F)
        freq: 128 Hz; valid at: 11/128s ((F,F,F),F)
        freq: 128 Hz; valid at: 12/128s ((F,F,T),T)
        freq: 128 Hz; valid at: 13/128s ((F,F,T),T)
```

Listing F.2(vi): The pipeline example



```
freq: 64 Hz; time: 4/64s
  .LInst.nInst
    freq: 64 Hz; time: 4/64s
      .MemFB.posIn
        freq: 64 Hz; valid at: 4/64s (T,(F,F))
        freq: 64 Hz; valid at: 3/64s (T,(T,T))
      .MemBF.negOut
        freq: 64 Hz; valid at: 4/64s (F,F,T)
        freq: 64 Hz; valid at: 5/64s (F,F,F)
        freq: 64 Hz; valid at: 6/64s (F,T,T)
  .LInst.pInst
    freq: 64 Hz; time: 4/64s
      .MemFB.datIn
        freq: 64 Hz; valid at: 5/64s (F,F,F)
        freq: 64 Hz; valid at: 4/64s (F,F,T)
        freq: 64 Hz; valid at: 3/64s (F,F,F)
      .MemBF.parOut
        freq: 128 Hz; valid at: 10/128s ((F,F,F),F)
        freq: 128 Hz; valid at: 11/128s ((F,F,F),F)
        freq: 128 Hz; valid at: 12/128s ((F,F,T),T)
        freq: 128 Hz; valid at: 13/128s ((F,F,T),T)
```

Listing F.2(vii): The pipeline example



freq: 64 Hz; *time*: $^{4}/_{64}$s

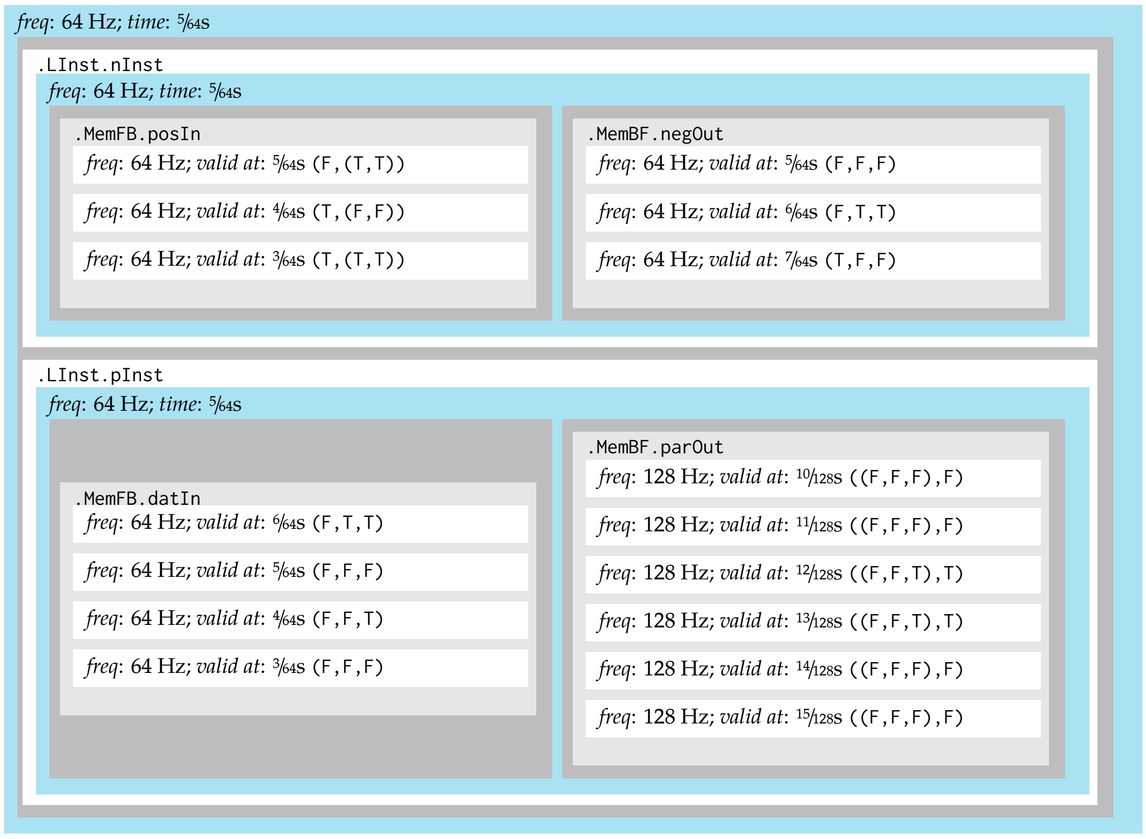
- `.LInst.nInst`
  - *freq*: 64 Hz; *time*: $^{4}/_{64}$s
    - `.MemFB.posIn`
      - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `(T,(F,F))`
      - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `(T,(T,T))`
    - `.MemBF.negOut`
      - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(F,T,T)`
- `.LInst.pInst`
  - *freq*: 64 Hz; *time*: $^{4}/_{64}$s
    - `.MemFB.datIn`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`
      - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`
    - `.MemBF.parOut`
      - *freq*: 128 Hz; *valid at*: $^{10}/_{128}$s `((F,F,F),F)`
      - *freq*: 128 Hz; *valid at*: $^{11}/_{128}$s `((F,F,F),F)`
      - *freq*: 128 Hz; *valid at*: $^{12}/_{128}$s `((F,F,T),T)`
      - *freq*: 128 Hz; *valid at*: $^{13}/_{128}$s `((F,F,T),T)`

Listing F.2(viii): The pipeline example

freq: 64 Hz; *time*: $^{5}/_{64}$s

- `.LInst.nInst`
  - *freq*: 64 Hz; *time*: $^{5}/_{64}$s
    - `.MemFB.posIn`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,(T,T))`
      - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `(T,(F,F))`
      - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `(T,(T,T))`
    - `.MemBF.negOut`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(F,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{7}/_{64}$s `(T,F,F)`
- `.LInst.pInst`
  - *freq*: 64 Hz; *time*: $^{5}/_{64}$s
    - `.MemFB.datIn`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(F,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `(F,F,T)`
      - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `(F,F,F)`
    - `.MemBF.parOut`
      - *freq*: 128 Hz; *valid at*: $^{10}/_{128}$s `((F,F,F),F)`
      - *freq*: 128 Hz; *valid at*: $^{11}/_{128}$s `((F,F,F),F)`
      - *freq*: 128 Hz; *valid at*: $^{12}/_{128}$s `((F,F,T),T)`
      - *freq*: 128 Hz; *valid at*: $^{13}/_{128}$s `((F,F,T),T)`
      - *freq*: 128 Hz; *valid at*: $^{14}/_{128}$s `((F,F,F),F)`
      - *freq*: 128 Hz; *valid at*: $^{15}/_{128}$s `((F,F,F),F)`

Listing F.2(ix): The pipeline example



```
freq: 64 Hz; time: 5/64s
  .LInst.nInst
  freq: 64 Hz; time: 5/64s
    .MemFB.posIn
      freq: 64 Hz; valid at: 5/64s (F,(T,T))
      freq: 64 Hz; valid at: 4/64s (T,(F,F))
      freq: 64 Hz; valid at: 3/64s (T,(T,T))
    .MemBF.negOut
      freq: 64 Hz; valid at: 5/64s (F,F,F)
      freq: 64 Hz; valid at: 6/64s (F,T,T)
      freq: 64 Hz; valid at: 7/64s (T,F,F)
  .LInst.pInst
  freq: 64 Hz; time: 5/64s
    .MemFB.datIn
      freq: 64 Hz; valid at: 6/64s (F,T,T)
      freq: 64 Hz; valid at: 5/64s (F,F,F)
      freq: 64 Hz; valid at: 4/64s (F,F,T)
      freq: 64 Hz; valid at: 3/64s (F,F,F)
    .MemBF.parOut
      freq: 128 Hz; valid at: 10/128s ((F,F,F),F)
      freq: 128 Hz; valid at: 11/128s ((F,F,F),F)
      freq: 128 Hz; valid at: 12/128s ((F,F,T),T)
      freq: 128 Hz; valid at: 13/128s ((F,F,T),T)
      freq: 128 Hz; valid at: 14/128s ((F,F,F),F)
      freq: 128 Hz; valid at: 15/128s ((F,F,F),F)
```

F.3 Simple checksum

Listing F.3 is the continuation of Listing 6.5.

Listing F.3(i): The simple checksum example

```
freq: 64 Hz; time: 6/64s
  .LInst.checksumInstConc
  freq: 64 Hz; time: 6/64s
    .MemFB.checkIn
      freq: 64 Hz; valid at: 9/64s (T,T,T)
      freq: 64 Hz; valid at: 8/64s (F,F,F)
      freq: 64 Hz; valid at: 7/64s (F,F,F)
      freq: 64 Hz; valid at: 6/64s (F,F,F)
      freq: 64 Hz; valid at: 5/64s (F,F,F)
    .MemFB.datIn
      freq: 64 Hz; valid at: 6/64s (T,F,F)
      freq: 64 Hz; valid at: 5/64s (T,T,T)
    .MemBF.checkOutDoublePair
      freq: 64 Hz; valid at: 9/64s (T,(T,T))
      freq: 64 Hz; valid at: 10/64s (T,(F,F))
    .MemBF.checkOutTriple
      freq: 64 Hz; valid at: 8/64s (F,F,F)
      freq: 64 Hz; valid at: 9/64s (T,T,T)
      freq: 64 Hz; valid at: 10/64s (T,F,F)
```

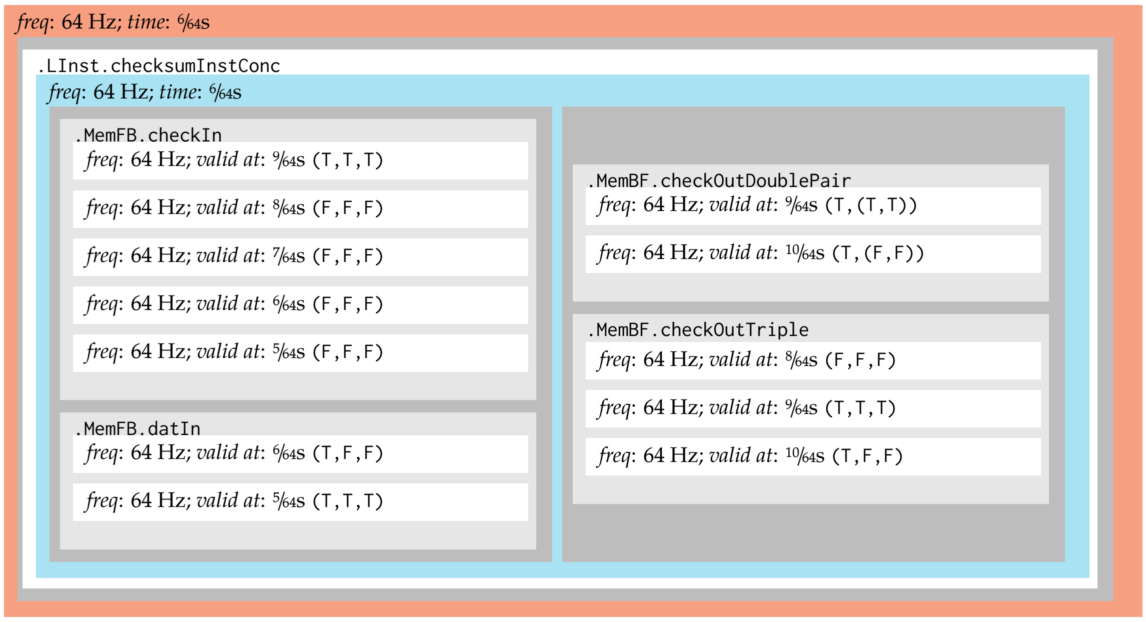
Listing F.3(ii): The simple checksum example

```
freq: 64 Hz; time: 7/64s
  .LInst.checksumInstConc
  freq: 64 Hz; time: 7/64s
    .MemFB.checkIn
      freq: 64 Hz; valid at: 10/64s (T,F,F)
      freq: 64 Hz; valid at: 9/64s (T,T,T)
      freq: 64 Hz; valid at: 8/64s (F,F,F)
      freq: 64 Hz; valid at: 7/64s (F,F,F)
      freq: 64 Hz; valid at: 6/64s (F,F,F)
      freq: 64 Hz; valid at: 5/64s (F,F,F)
    .MemFB.datIn
      freq: 64 Hz; valid at: 7/64s (F,T,T)
      freq: 64 Hz; valid at: 6/64s (T,F,F)
      freq: 64 Hz; valid at: 5/64s (T,T,T)
    .MemBF.checkOutDoublePair
      freq: 64 Hz; valid at: 9/64s (T,(T,T))
      freq: 64 Hz; valid at: 10/64s (T,(F,F))
      freq: 64 Hz; valid at: 11/64s (F,(T,T))
    .MemBF.checkOutTriple
      freq: 64 Hz; valid at: 8/64s (F,F,F)
      freq: 64 Hz; valid at: 9/64s (T,T,T)
      freq: 64 Hz; valid at: 10/64s (T,F,F)
      freq: 64 Hz; valid at: 11/64s (F,T,T)
```

F.4 Pipelined checksum

Listing F.4 is the continuation of Listing 6.6.

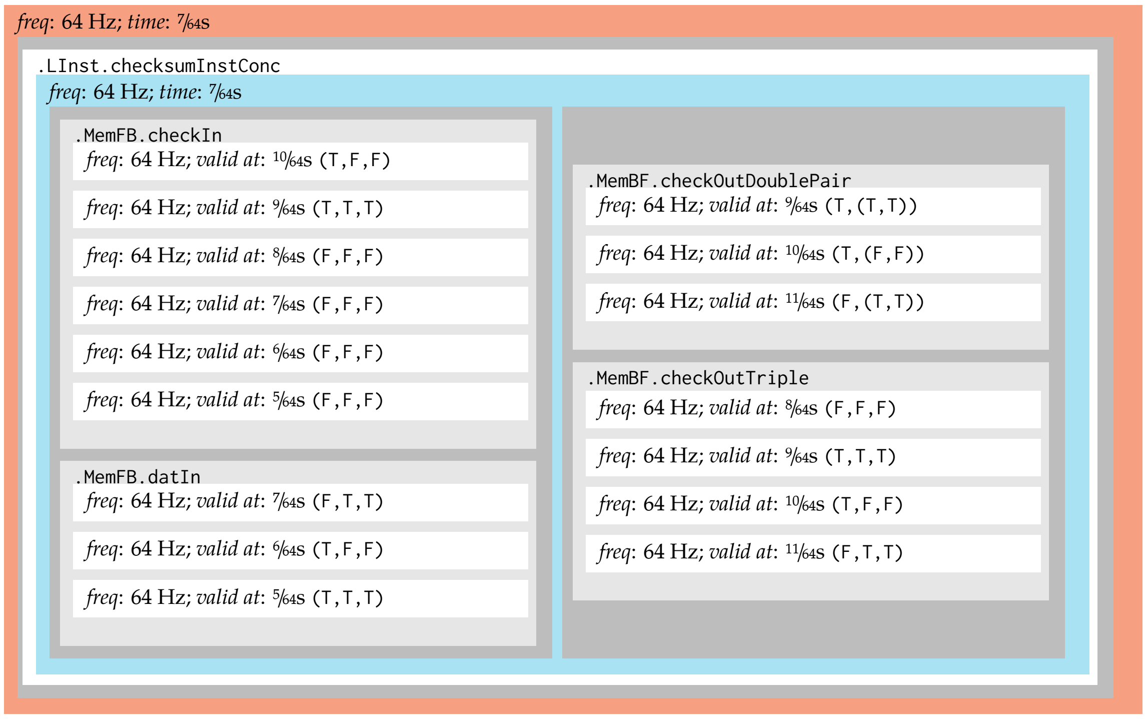
Listing F.4(i): The pipelined checksum example



freq: 64 Hz; *time*: $^{6}/_{64}$s

- `.LInst.checksumInstConc`
  - *freq*: 64 Hz; *time*: $^{6}/_{64}$s
    - `.MemFB.checkIn`
      - *freq*: 64 Hz; *valid at*: $^{9}/_{64}$s `(T,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{8}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{7}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`
    - `.MemFB.datIn`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(T,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(T,T,T)`
    - `.MemBF.checkOutDoublePair`
      - *freq*: 64 Hz; *valid at*: $^{9}/_{64}$s `(T,(T,T))`
      - *freq*: 64 Hz; *valid at*: $^{10}/_{64}$s `(T,(F,F))`
    - `.MemBF.checkOutTriple`
      - *freq*: 64 Hz; *valid at*: $^{8}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{9}/_{64}$s `(T,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{10}/_{64}$s `(T,F,F)`

Listing F.4(ii): The pipelined checksum example



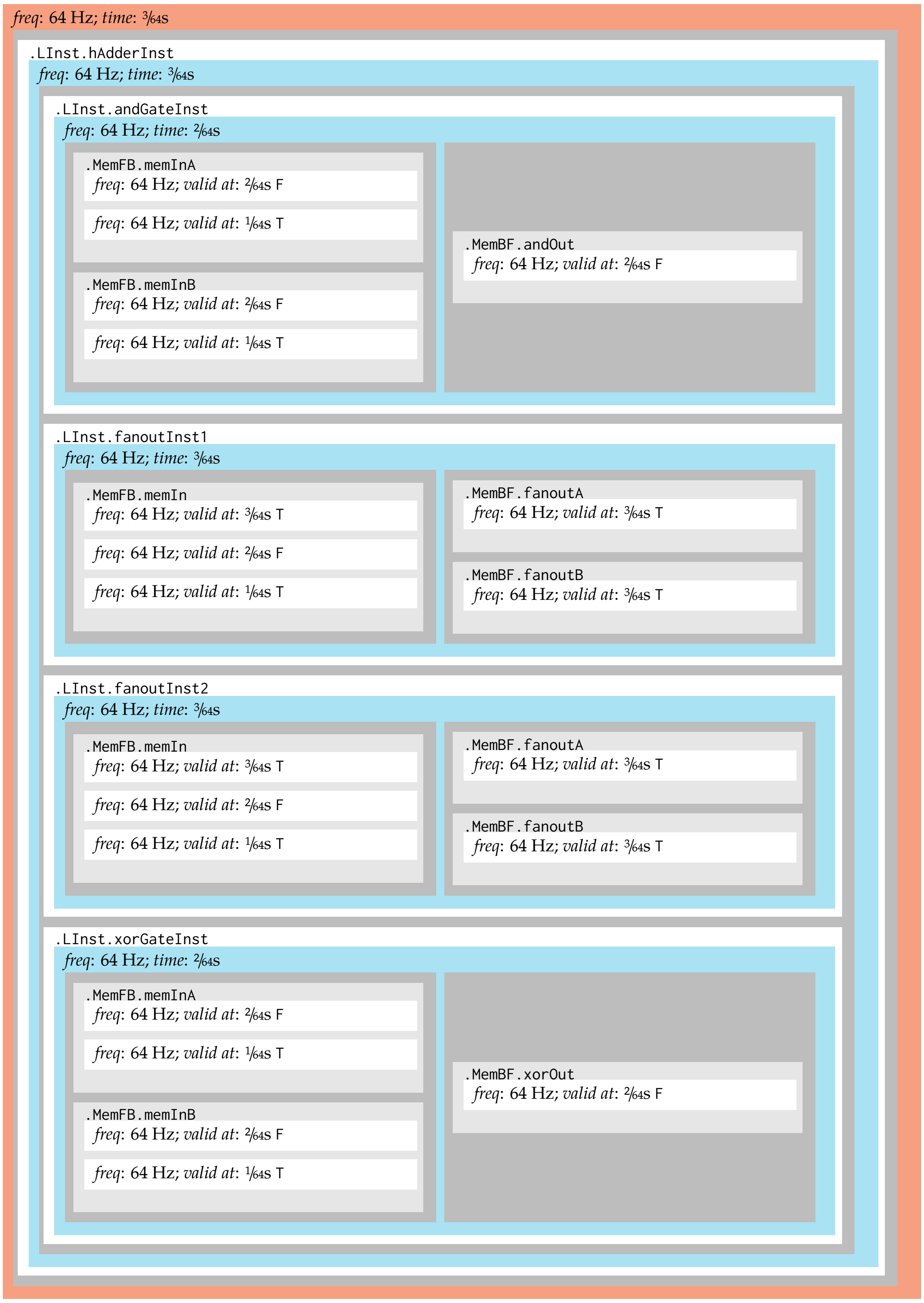
freq: 64 Hz; *time*: $^{7}/_{64}$s

- `.LInst.checksumInstConc`
  - *freq*: 64 Hz; *time*: $^{7}/_{64}$s
    - `.MemFB.checkIn`
      - *freq*: 64 Hz; *valid at*: $^{10}/_{64}$s `(T,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{9}/_{64}$s `(T,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{8}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{7}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(F,F,F)`
    - `.MemFB.datIn`
      - *freq*: 64 Hz; *valid at*: $^{7}/_{64}$s `(F,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{6}/_{64}$s `(T,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{5}/_{64}$s `(T,T,T)`
    - `.MemBF.checkOutDoublePair`
      - *freq*: 64 Hz; *valid at*: $^{9}/_{64}$s `(T,(T,T))`
      - *freq*: 64 Hz; *valid at*: $^{10}/_{64}$s `(T,(F,F))`
      - *freq*: 64 Hz; *valid at*: $^{11}/_{64}$s `(F,(T,T))`
    - `.MemBF.checkOutTriple`
      - *freq*: 64 Hz; *valid at*: $^{8}/_{64}$s `(F,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{9}/_{64}$s `(T,T,T)`
      - *freq*: 64 Hz; *valid at*: $^{10}/_{64}$s `(T,F,F)`
      - *freq*: 64 Hz; *valid at*: $^{11}/_{64}$s `(F,T,T)`

F.5 Half adder

Listing F.5 is the continuation of Listing 6.8.

Listing F.5(i): The half adder example



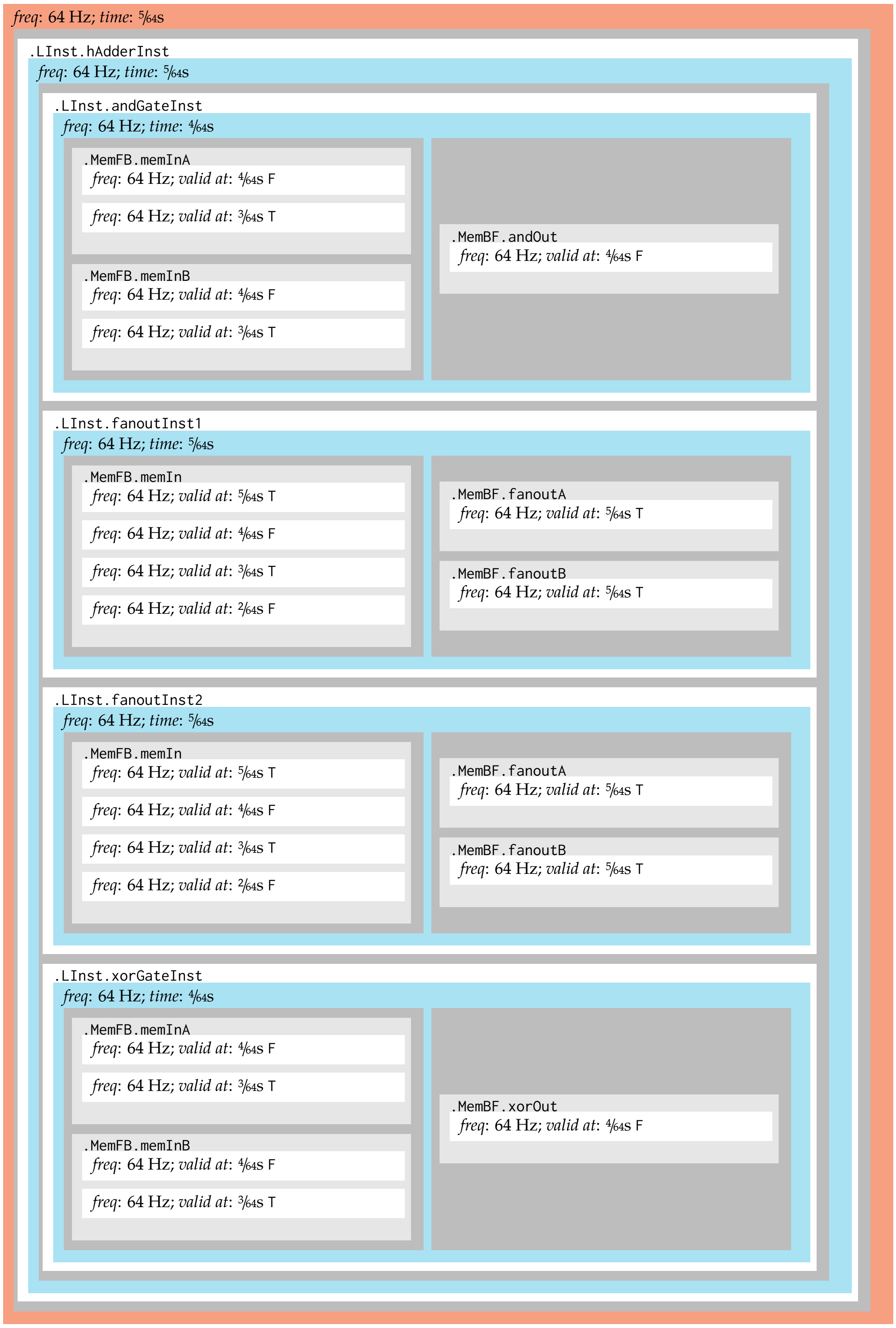
- *freq*: 64 Hz; *time*: $\frac{3}{64}$s
  - `.LInst.hAdderInst`
    - *freq*: 64 Hz; *time*: $\frac{3}{64}$s
      - `.LInst.andGateInst`
        - *freq*: 64 Hz; *time*: $\frac{2}{64}$s
          - `.MemFB.memInA`
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
            - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
          - `.MemFB.memInB`
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
            - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
          - `.MemBF.andOut`
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
      - `.LInst.fanoutInst1`
        - *freq*: 64 Hz; *time*: $\frac{3}{64}$s
          - `.MemFB.memIn`
            - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s T
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
            - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
          - `.MemBF.fanoutA`
            - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s T
          - `.MemBF.fanoutB`
            - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s T
      - `.LInst.fanoutInst2`
        - *freq*: 64 Hz; *time*: $\frac{3}{64}$s
          - `.MemFB.memIn`
            - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s T
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
            - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
          - `.MemBF.fanoutA`
            - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s T
          - `.MemBF.fanoutB`
            - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s T
      - `.LInst.xorGateInst`
        - *freq*: 64 Hz; *time*: $\frac{2}{64}$s
          - `.MemFB.memInA`
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
            - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
          - `.MemFB.memInB`
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
            - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
          - `.MemBF.xorOut`
            - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F

Listing F.5(ii): The half adder example

freq: 64 Hz; *time*: 4/64s

- `.LInst.hAdderInst`
  freq: 64 Hz; *time*: 4/64s
  - `.LInst.andGateInst`
    freq: 64 Hz; *time*: 3/64s
    - `.MemFB.memInA`
      - *freq*: 64 Hz; *valid at*: 3/64s T
      - *freq*: 64 Hz; *valid at*: 2/64s F
    - `.MemFB.memInB`
      - *freq*: 64 Hz; *valid at*: 3/64s T
      - *freq*: 64 Hz; *valid at*: 2/64s F
    - `.MemBF.andOut`
      - *freq*: 64 Hz; *valid at*: 3/64s T
  - `.LInst.fanoutInst1`
    freq: 64 Hz; *time*: 4/64s
    - `.MemFB.memIn`
      - *freq*: 64 Hz; *valid at*: 4/64s F
      - *freq*: 64 Hz; *valid at*: 3/64s T
      - *freq*: 64 Hz; *valid at*: 2/64s F
      - *freq*: 64 Hz; *valid at*: 1/64s T
    - `.MemBF.fanoutA`
      - *freq*: 64 Hz; *valid at*: 4/64s F
    - `.MemBF.fanoutB`
      - *freq*: 64 Hz; *valid at*: 4/64s F
  - `.LInst.fanoutInst2`
    freq: 64 Hz; *time*: 4/64s
    - `.MemFB.memIn`
      - *freq*: 64 Hz; *valid at*: 4/64s F
      - *freq*: 64 Hz; *valid at*: 3/64s T
      - *freq*: 64 Hz; *valid at*: 2/64s F
      - *freq*: 64 Hz; *valid at*: 1/64s T
    - `.MemBF.fanoutA`
      - *freq*: 64 Hz; *valid at*: 4/64s F
    - `.MemBF.fanoutB`
      - *freq*: 64 Hz; *valid at*: 4/64s F
  - `.LInst.xorGateInst`
    freq: 64 Hz; *time*: 3/64s
    - `.MemFB.memInA`
      - *freq*: 64 Hz; *valid at*: 3/64s T
      - *freq*: 64 Hz; *valid at*: 2/64s F
    - `.MemFB.memInB`
      - *freq*: 64 Hz; *valid at*: 3/64s T
      - *freq*: 64 Hz; *valid at*: 2/64s F
    - `.MemBF.xorOut`
      - *freq*: 64 Hz; *valid at*: 3/64s F

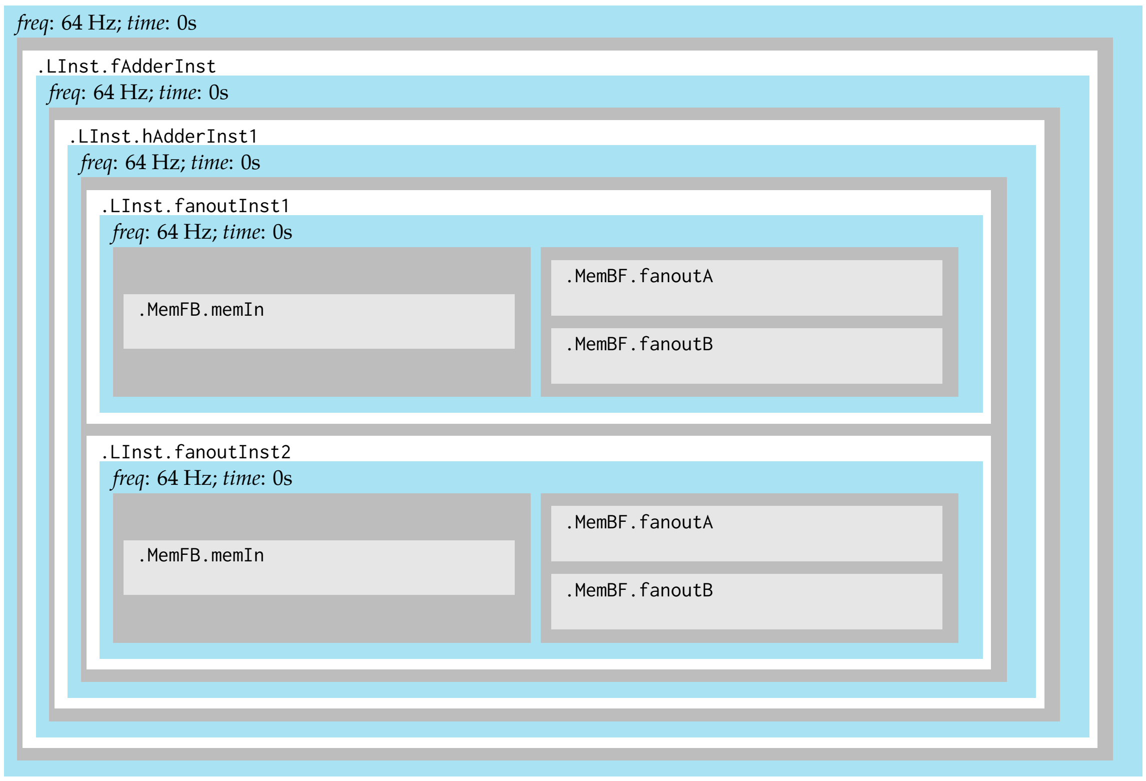
Listing F.5(iii): The half adder example



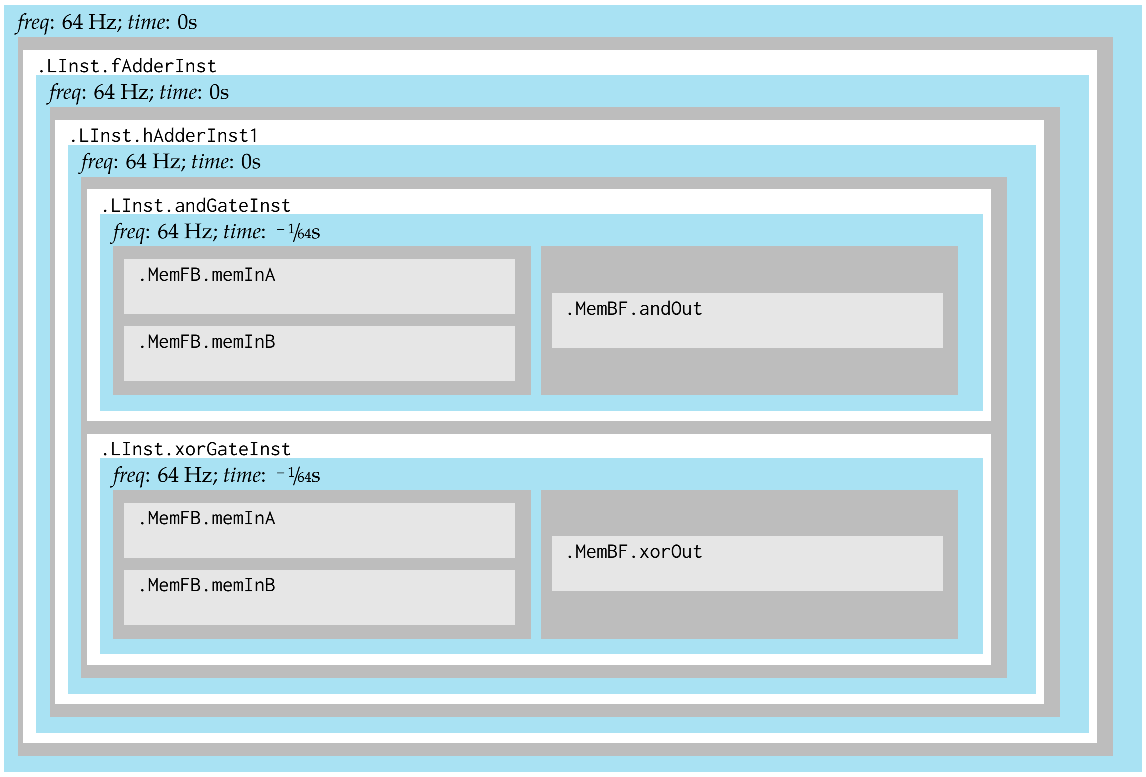
```
freq: 64 Hz; time: 5/64s
  .LInst.hAdderInst
  freq: 64 Hz; time: 5/64s
    .LInst.andGateInst
    freq: 64 Hz; time: 4/64s
      .MemFB.memInA
        freq: 64 Hz; valid at: 4/64s F
        freq: 64 Hz; valid at: 3/64s T
      .MemFB.memInB
        freq: 64 Hz; valid at: 4/64s F
        freq: 64 Hz; valid at: 3/64s T
      .MemBF.andOut
        freq: 64 Hz; valid at: 4/64s F
    .LInst.fanoutInst1
    freq: 64 Hz; time: 5/64s
      .MemFB.memIn
        freq: 64 Hz; valid at: 5/64s T
        freq: 64 Hz; valid at: 4/64s F
        freq: 64 Hz; valid at: 3/64s T
        freq: 64 Hz; valid at: 2/64s F
      .MemBF.fanoutA
        freq: 64 Hz; valid at: 5/64s T
      .MemBF.fanoutB
        freq: 64 Hz; valid at: 5/64s T
    .LInst.fanoutInst2
    freq: 64 Hz; time: 5/64s
      .MemFB.memIn
        freq: 64 Hz; valid at: 5/64s T
        freq: 64 Hz; valid at: 4/64s F
        freq: 64 Hz; valid at: 3/64s T
        freq: 64 Hz; valid at: 2/64s F
      .MemBF.fanoutA
        freq: 64 Hz; valid at: 5/64s T
      .MemBF.fanoutB
        freq: 64 Hz; valid at: 5/64s T
    .LInst.xorGateInst
    freq: 64 Hz; time: 4/64s
      .MemFB.memInA
        freq: 64 Hz; valid at: 4/64s F
        freq: 64 Hz; valid at: 3/64s T
      .MemFB.memInB
        freq: 64 Hz; valid at: 4/64s F
        freq: 64 Hz; valid at: 3/64s T
      .MemBF.xorOut
        freq: 64 Hz; valid at: 4/64s F
```

F.6 Full adder

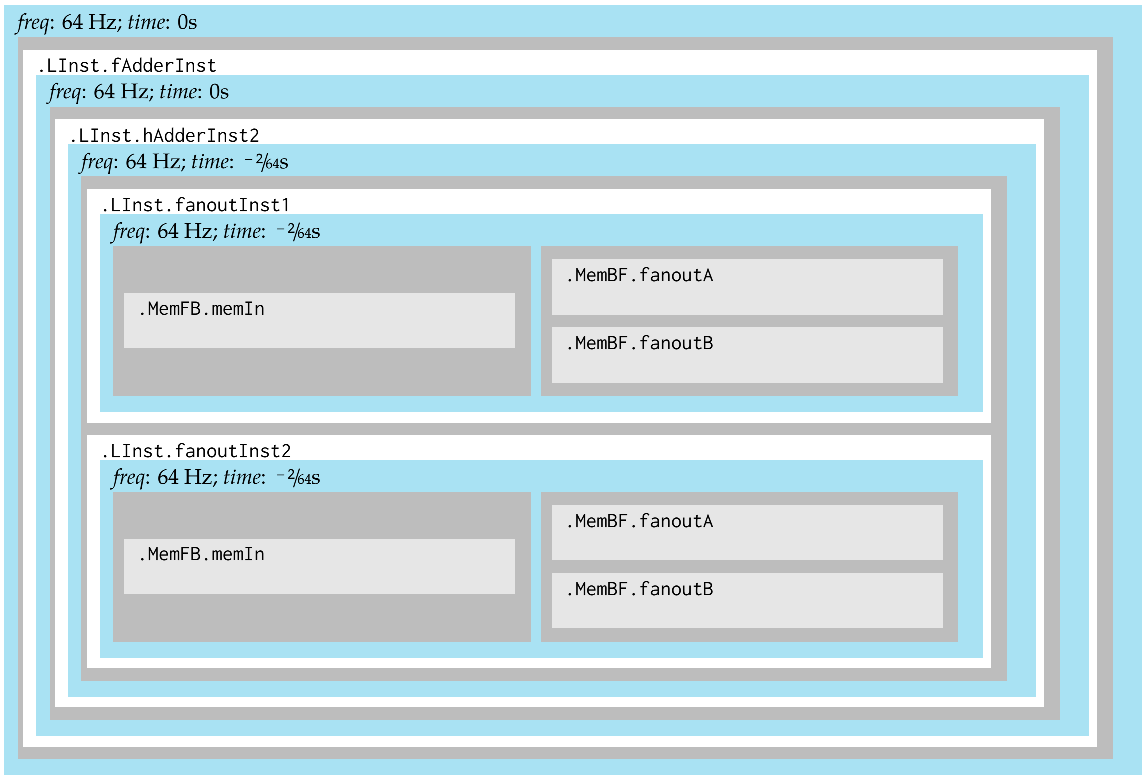
Listing F.6(i): The full adder example



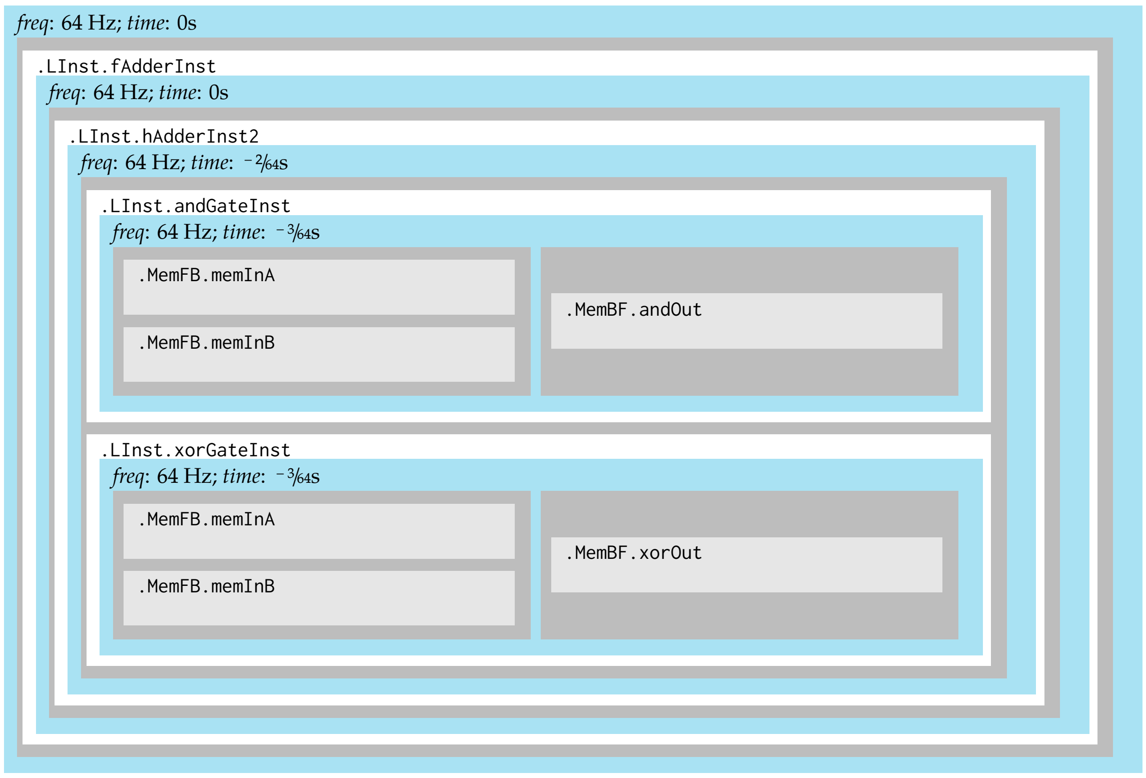
Listing F.6(ii): The full adder example



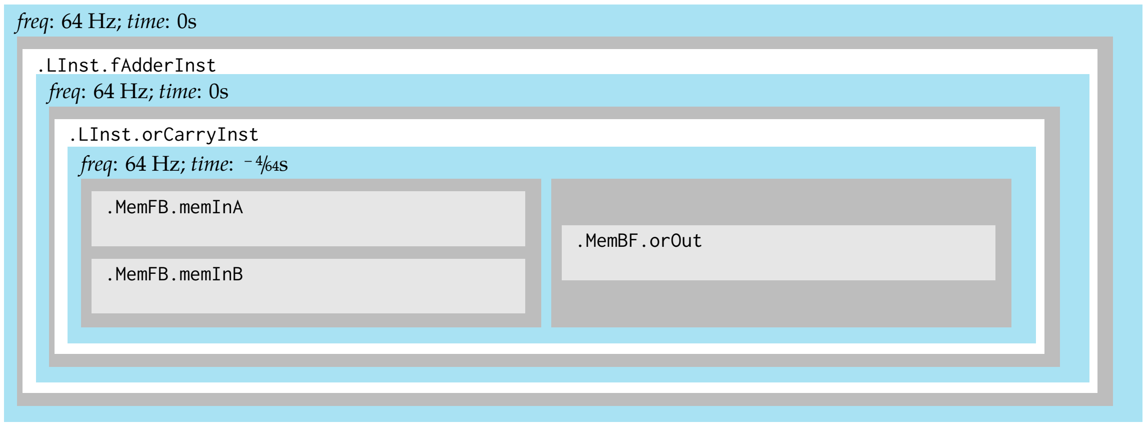
Listing F.6(iii): The full adder example



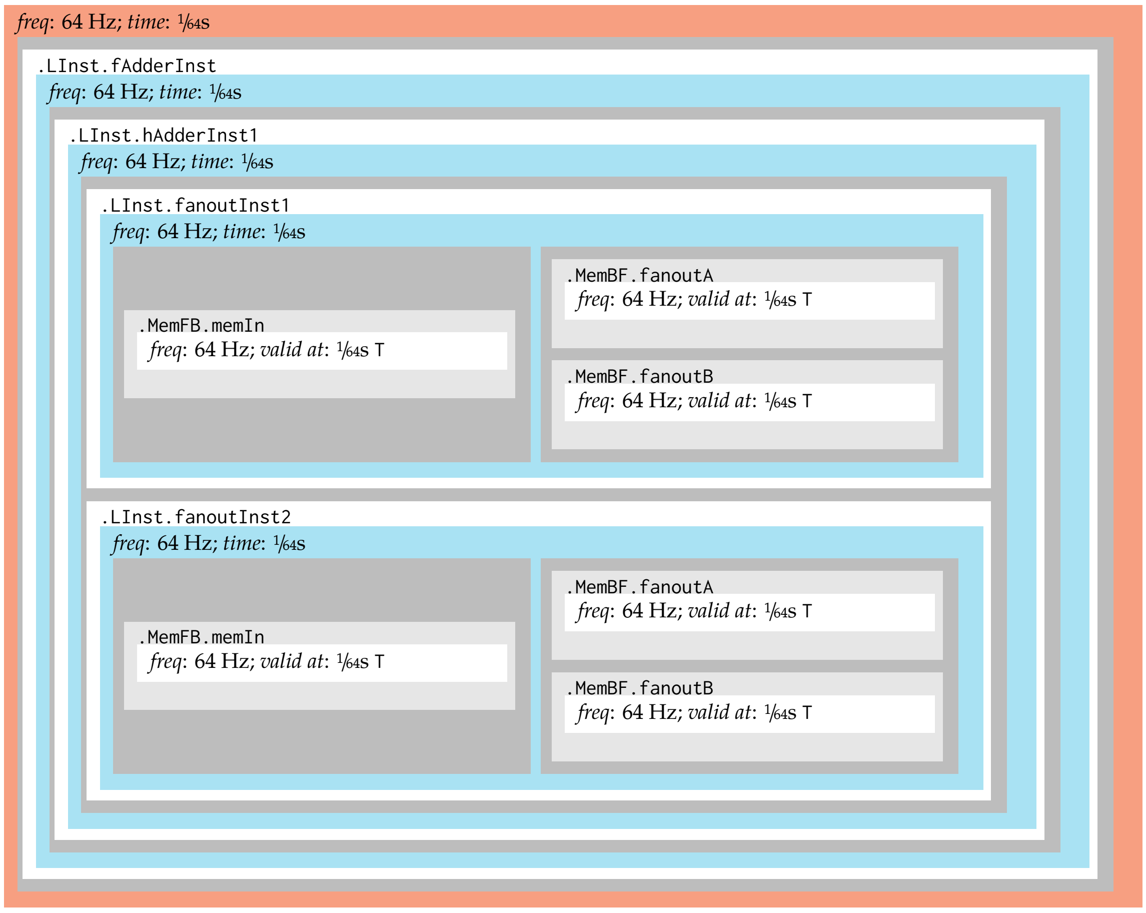
Listing F.6(iv): The full adder example


freq: 64 Hz; *time*: 0s
.LInst.fAdderInst
freq: 64 Hz; *time*: 0s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $-\frac{2}{64}$s
.LInst.andGateInst
freq: 64 Hz; *time*: $-\frac{3}{64}$s
.MemFB.memInA
.MemFB.memInB
.MemBF.andOut
.LInst.xorGateInst
freq: 64 Hz; *time*: $-\frac{3}{64}$s
.MemFB.memInA
.MemFB.memInB
.MemBF.xorOut

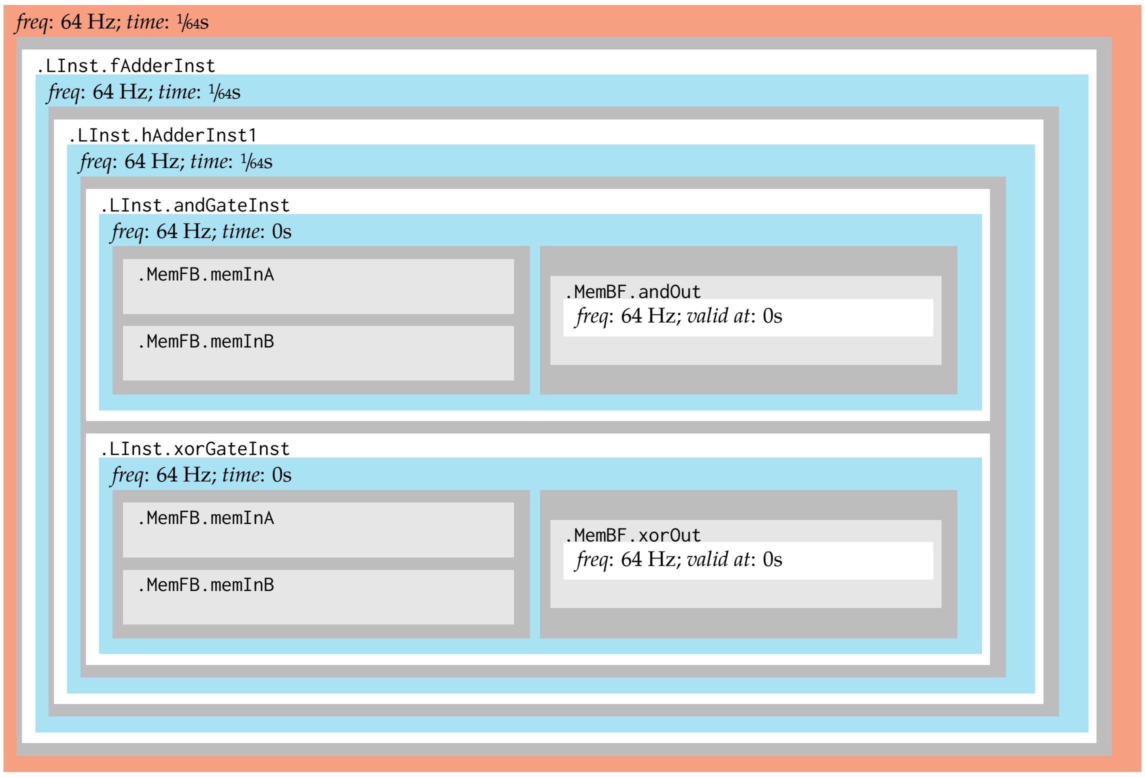
Listing F.6(v): The full adder example


freq: 64 Hz; *time*: 0s
.LInst.fAdderInst
freq: 64 Hz; *time*: 0s
.LInst.orCarryInst
freq: 64 Hz; *time*: $-\frac{4}{64}$s
.MemFB.memInA
.MemFB.memInB
.MemBF.orOut

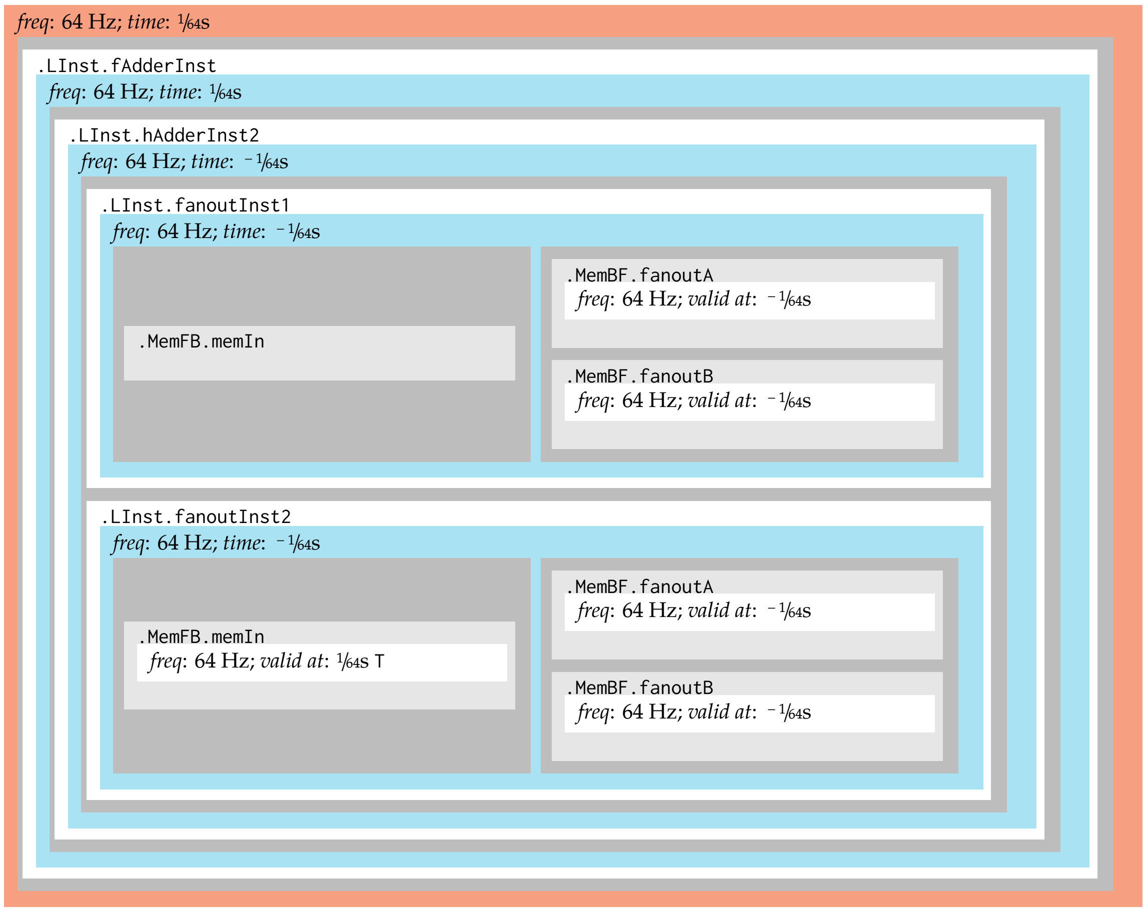
Listing F.6(vi): The full adder example


freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $^1/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.LInst.fanoutInst2
freq: 64 Hz; *time*: $^1/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $^1/_{64}$s T

Listing F.6(vii): The full adder example

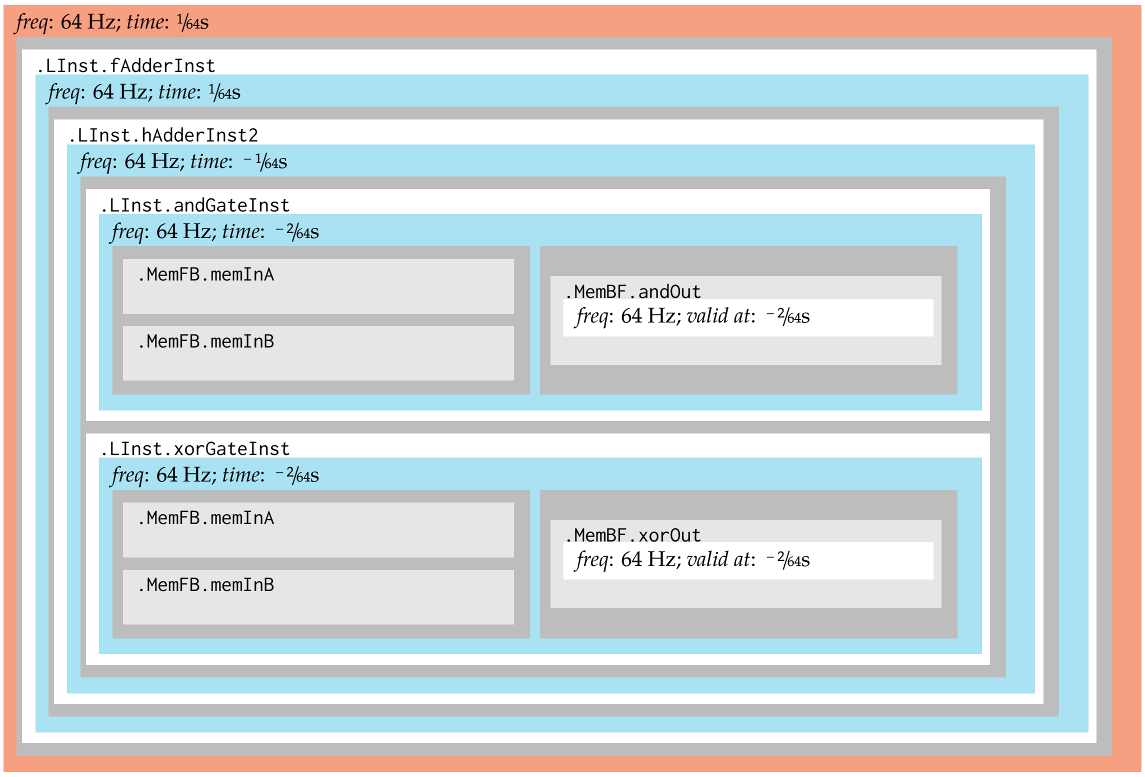

freq: 64 Hz; *time*: 1/64s
.LInst.fAdderInst
freq: 64 Hz; *time*: 1/64s
.LInst.hAdderInst1
freq: 64 Hz; *time*: 1/64s
.LInst.andGateInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
.MemFB.memInB
.MemBF.andOut
freq: 64 Hz; *valid at*: 0s
.LInst.xorGateInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
.MemFB.memInB
.MemBF.xorOut
freq: 64 Hz; *valid at*: 0s

Listing F.6(viii): The full adder example



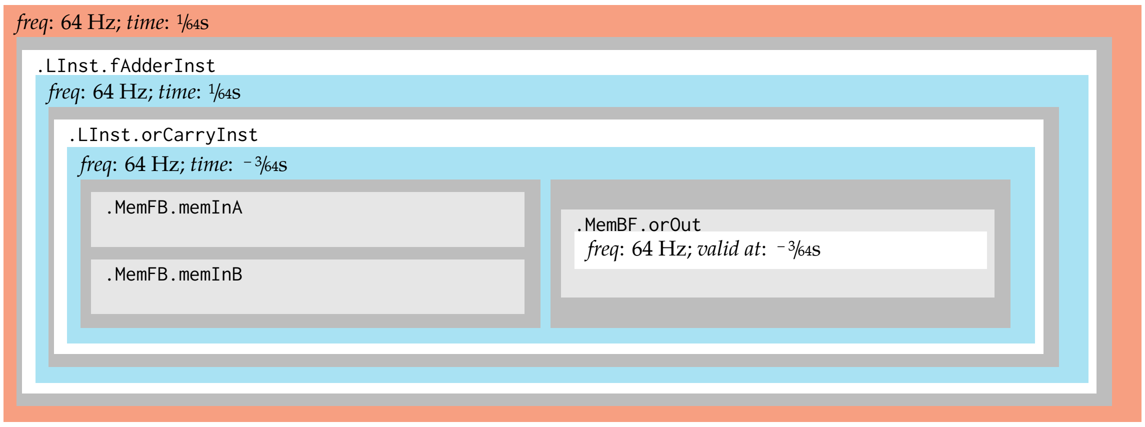
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $-^1/_{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $-^1/_{64}$s
.MemFB.memIn
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-^1/_{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-^1/_{64}$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $-^1/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-^1/_{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-^1/_{64}$s

Listing F.6(ix): The full adder example



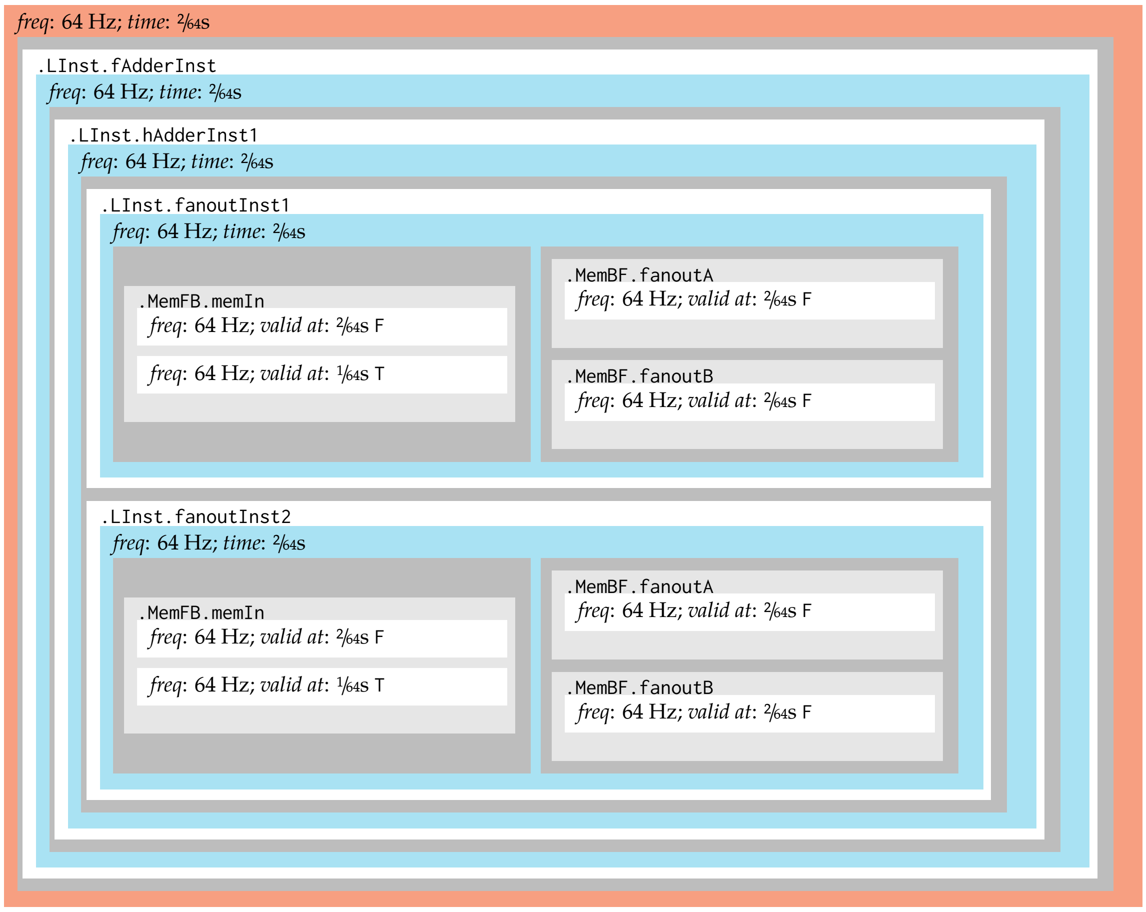
freq: 64 Hz; *time*: 1/64s
.LInst.fAdderInst
freq: 64 Hz; *time*: 1/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: −1/64s
.LInst.andGateInst
freq: 64 Hz; *time*: −2/64s
.MemFB.memInA
.MemFB.memInB
.MemBF.andOut
freq: 64 Hz; *valid at*: −2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: −2/64s
.MemFB.memInA
.MemFB.memInB
.MemBF.xorOut
freq: 64 Hz; *valid at*: −2/64s

Listing F.6(x): The full adder example



freq: 64 Hz; *time*: 1/64s
.LInst.fAdderInst
freq: 64 Hz; *time*: 1/64s
.LInst.orCarryInst
freq: 64 Hz; *time*: −3/64s
.MemFB.memInA
.MemFB.memInB
.MemBF.orOut
freq: 64 Hz; *valid at*: −3/64s

Listing F.6(xi): The full adder example



freq: 64 Hz; *time*: $^2/_{64}$s

`.LInst.fAdderInst`

freq: 64 Hz; *time*: $^2/_{64}$s

`.LInst.hAdderInst1`

freq: 64 Hz; *time*: $^2/_{64}$s

`.LInst.fanoutInst1`

freq: 64 Hz; *time*: $^2/_{64}$s

`.MemFB.memIn`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

freq: 64 Hz; *valid at*: $^1/_{64}$s `T`

`.MemBF.fanoutA`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

`.MemBF.fanoutB`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

`.LInst.fanoutInst2`

freq: 64 Hz; *time*: $^2/_{64}$s

`.MemFB.memIn`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

freq: 64 Hz; *valid at*: $^1/_{64}$s `T`

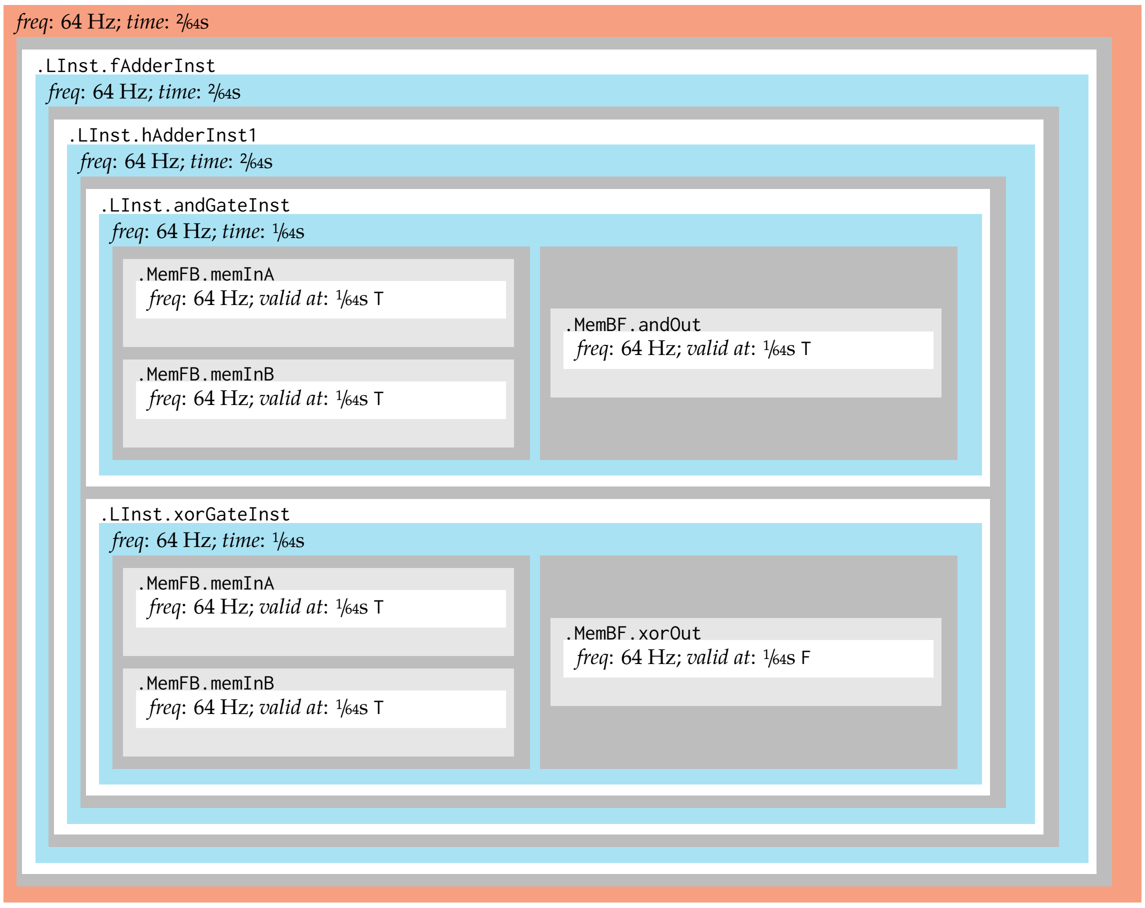
`.MemBF.fanoutA`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

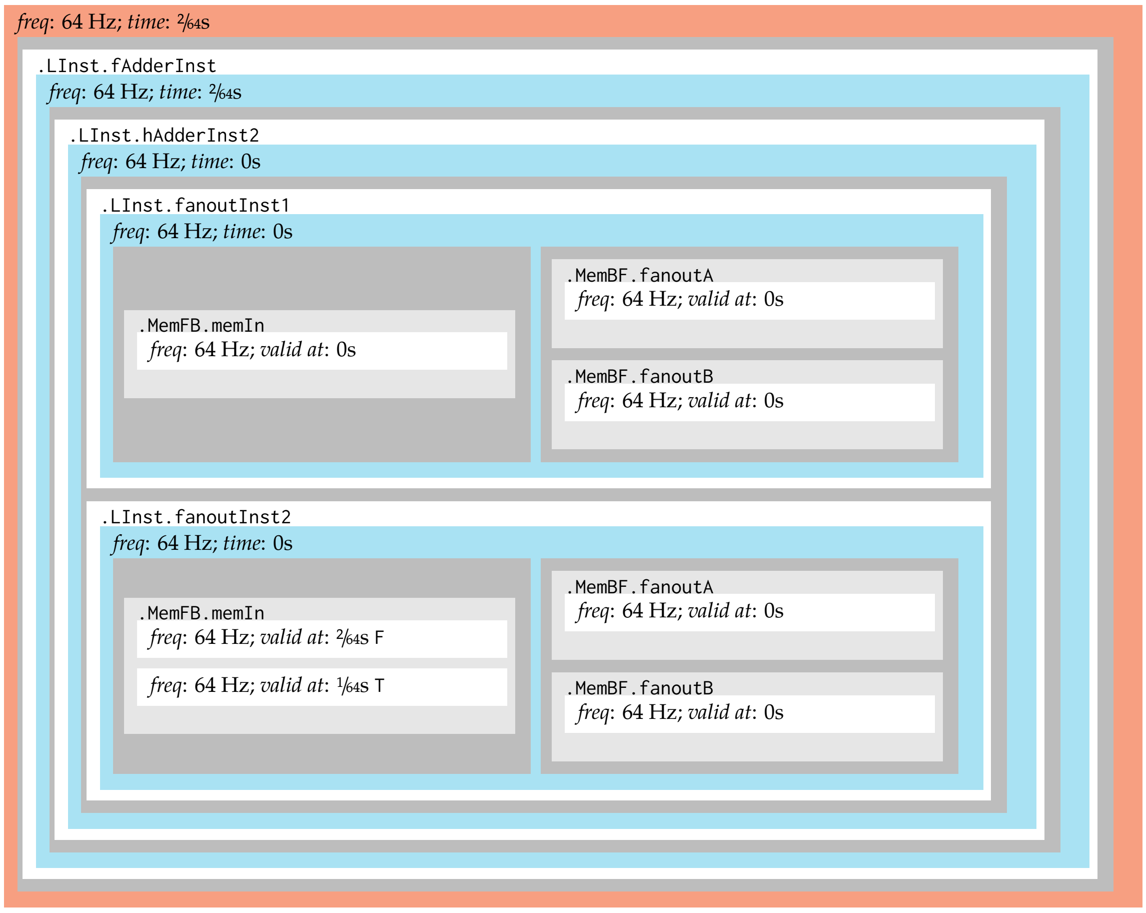
`.MemBF.fanoutB`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

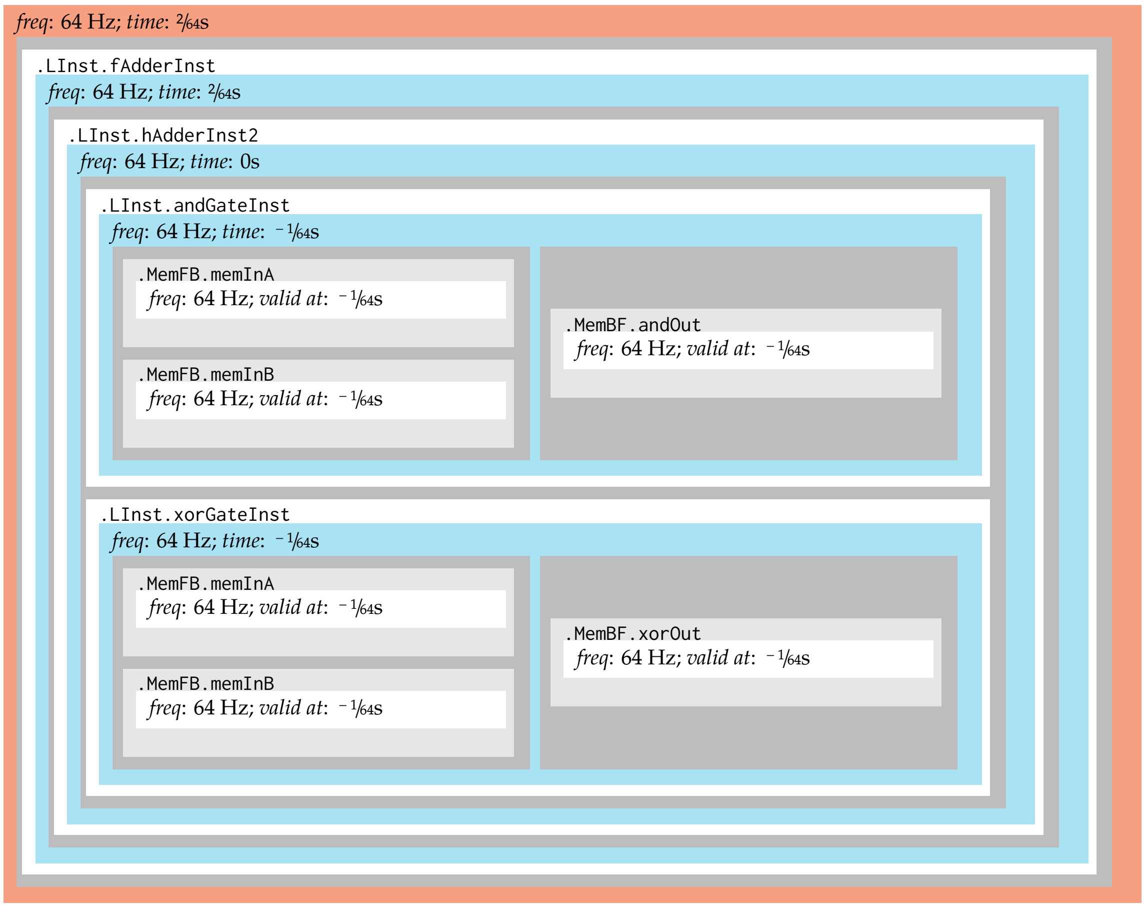
Listing F.6(xii): The full adder example


freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.andGateInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemBF.andOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.LInst.xorGateInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F

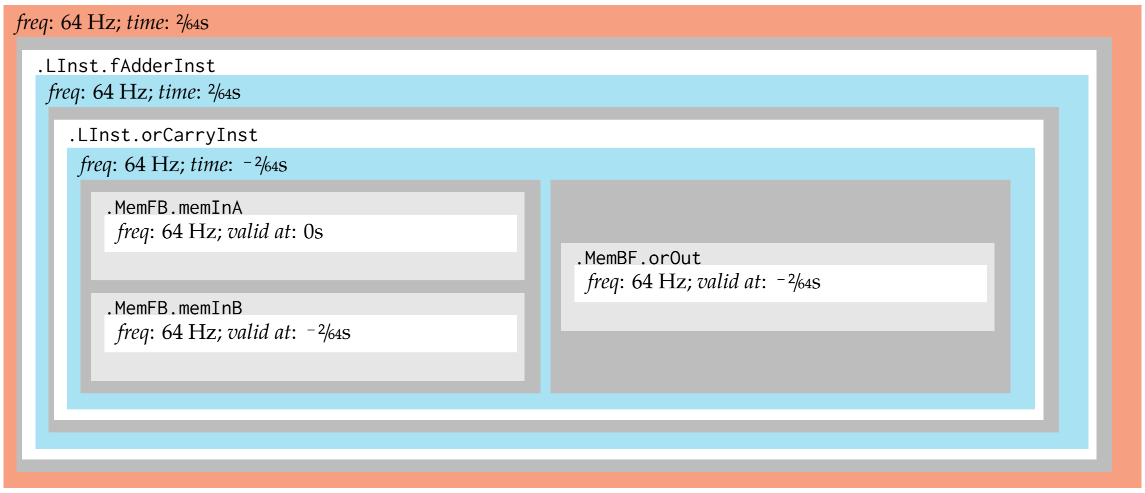
Listing F.6(xiii): The full adder example


freq: 64 Hz; *time*: $^2\!/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^2\!/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 0s
.LInst.fanoutInst1
freq: 64 Hz; *time*: 0s
.MemFB.memIn
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 0s
.LInst.fanoutInst2
freq: 64 Hz; *time*: 0s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^2\!/_{64}$s F
freq: 64 Hz; *valid at*: $^1\!/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 0s

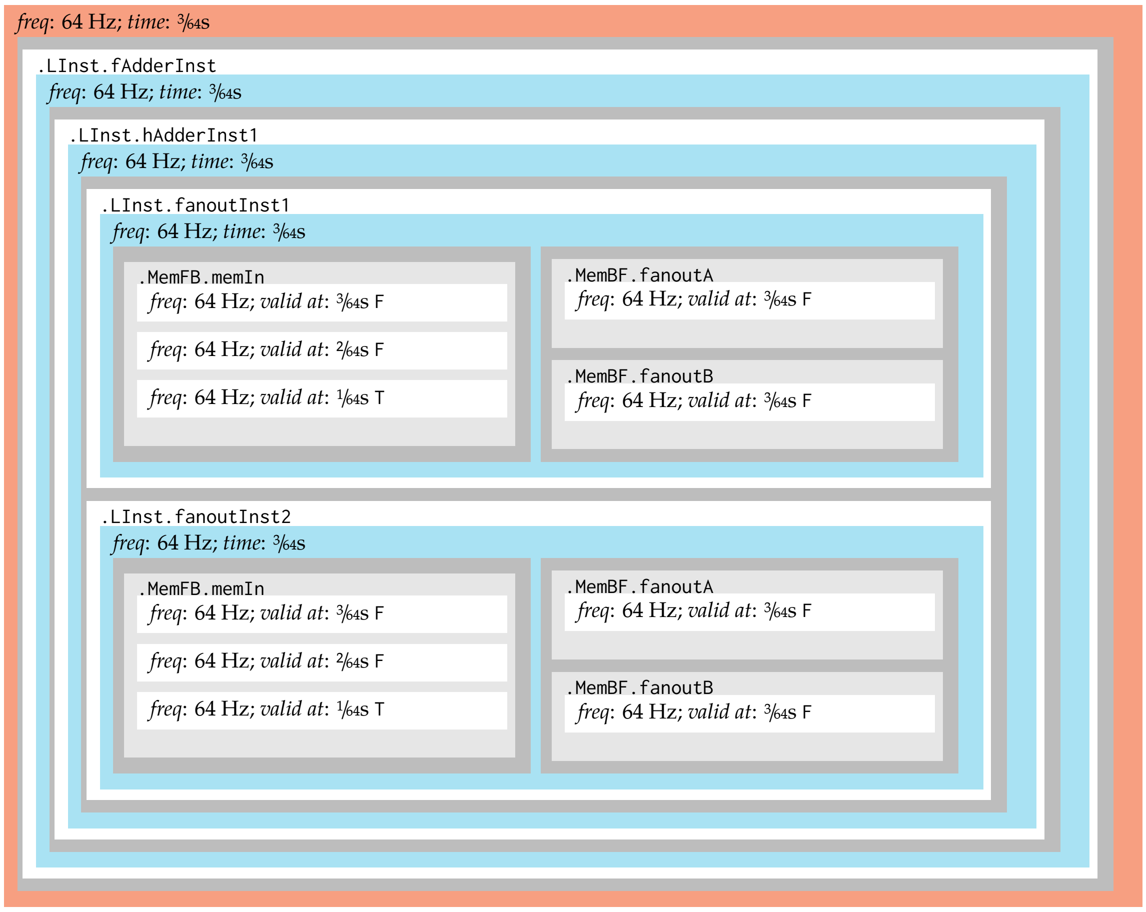
Listing F.6(xiv): The full adder example


freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 0s
.LInst.andGateInst
freq: 64 Hz; *time*: $-^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $-^{1}/_{64}$s
.MemFB.memInB
freq: 64 Hz; *valid at*: $-^{1}/_{64}$s
.MemBF.andOut
freq: 64 Hz; *valid at*: $-^{1}/_{64}$s
.LInst.xorGateInst
freq: 64 Hz; *time*: $-^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $-^{1}/_{64}$s
.MemFB.memInB
freq: 64 Hz; *valid at*: $-^{1}/_{64}$s
.MemBF.xorOut
freq: 64 Hz; *valid at*: $-^{1}/_{64}$s

Listing F.6(xv): The full adder example


freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $-^{2}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $-^{2}/_{64}$s
.MemBF.orOut
freq: 64 Hz; *valid at*: $-^{2}/_{64}$s

Listing F.6(xvi): The full adder example



freq: 64 Hz; *time*: $^{3}/_{64}$s

`.LInst.fAdderInst`

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.LInst.hAdderInst1`

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.LInst.fanoutInst1`

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.MemFB.memIn`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s F

freq: 64 Hz; *valid at*: $^{2}/_{64}$s F

freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

`.MemBF.fanoutA`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s F

`.MemBF.fanoutB`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s F

`.LInst.fanoutInst2`

freq: 64 Hz; *time*: $^{3}/_{64}$s

`.MemFB.memIn`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s F

freq: 64 Hz; *valid at*: $^{2}/_{64}$s F

freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

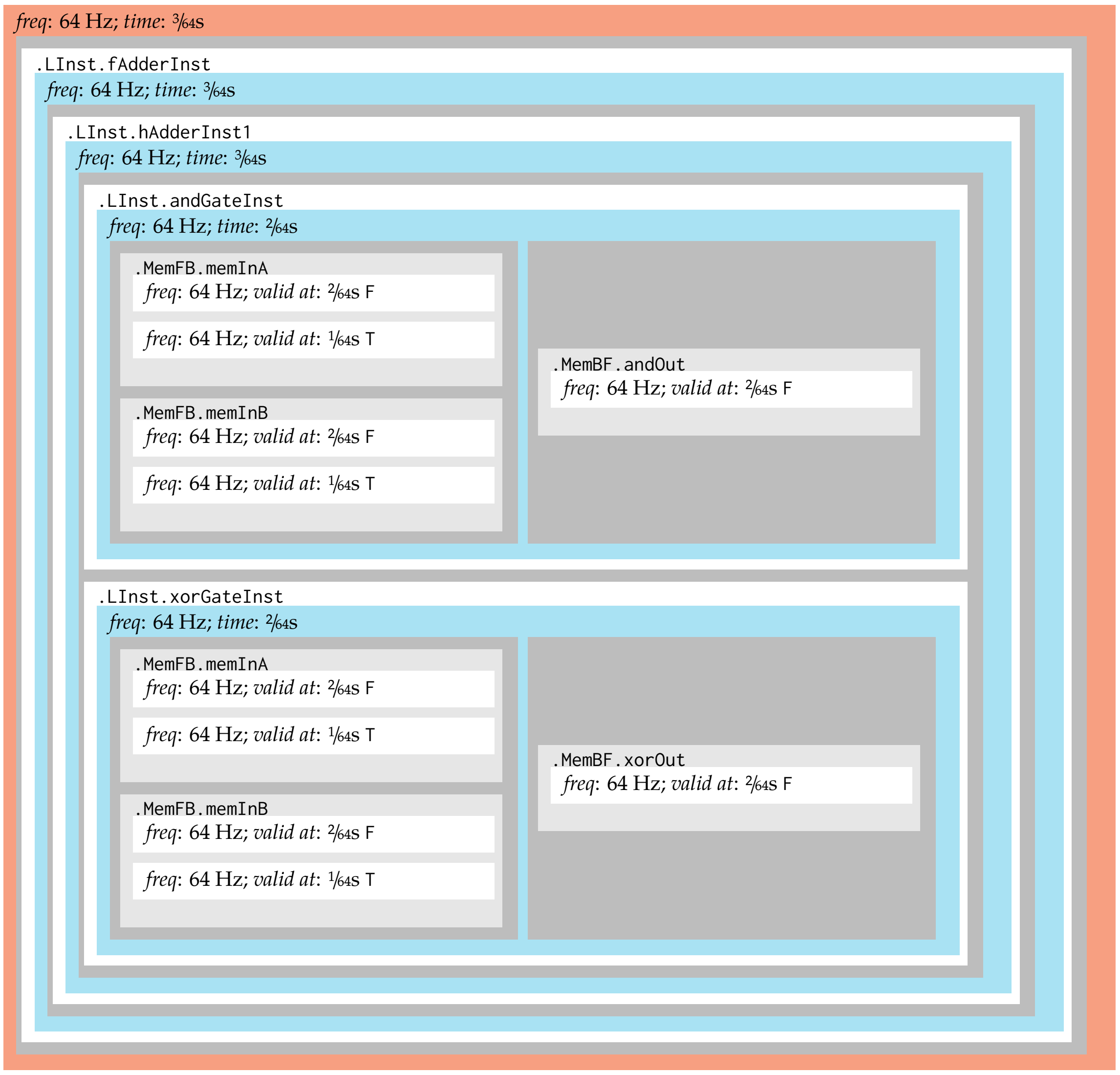
`.MemBF.fanoutA`

freq: 64 Hz; *valid at*: $^{3}/_{64}$s F

`.MemBF.fanoutB`

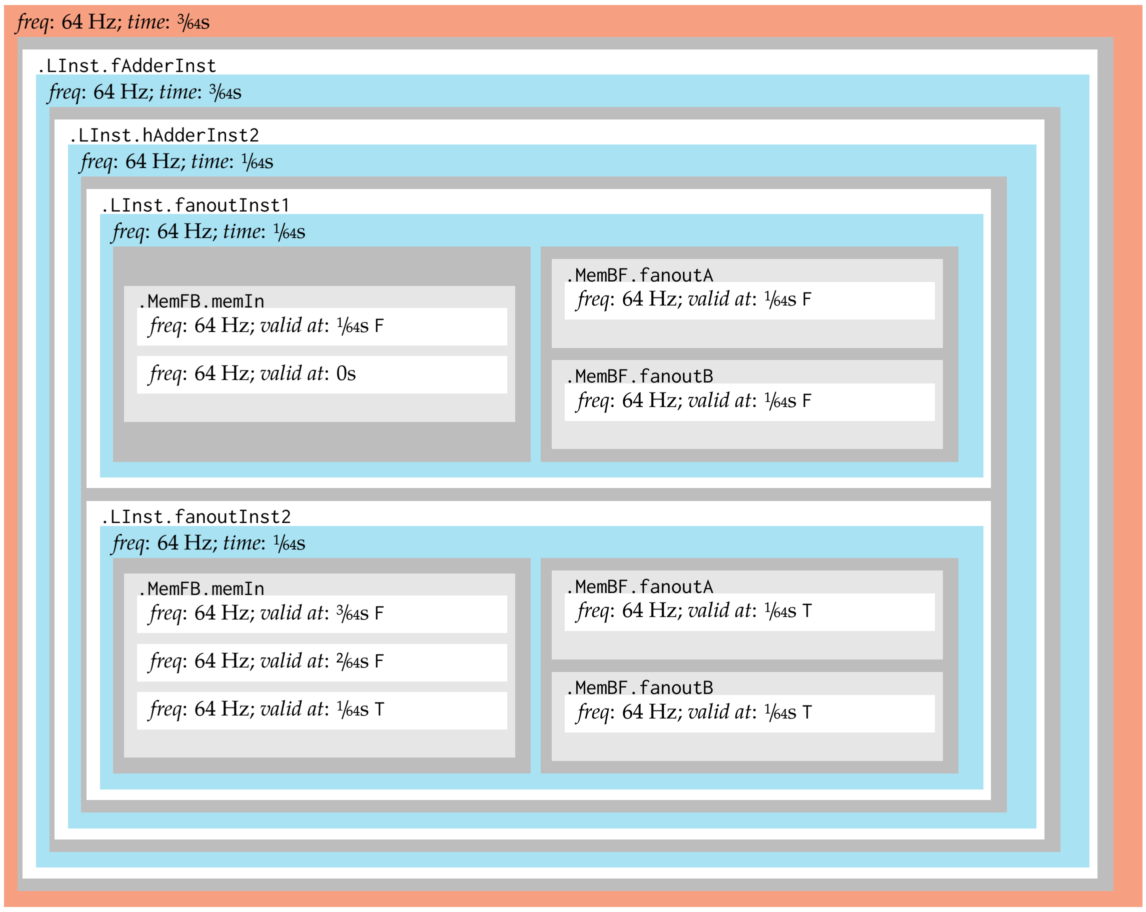
freq: 64 Hz; *valid at*: $^{3}/_{64}$s F

Listing F.6(xvii): The full adder example



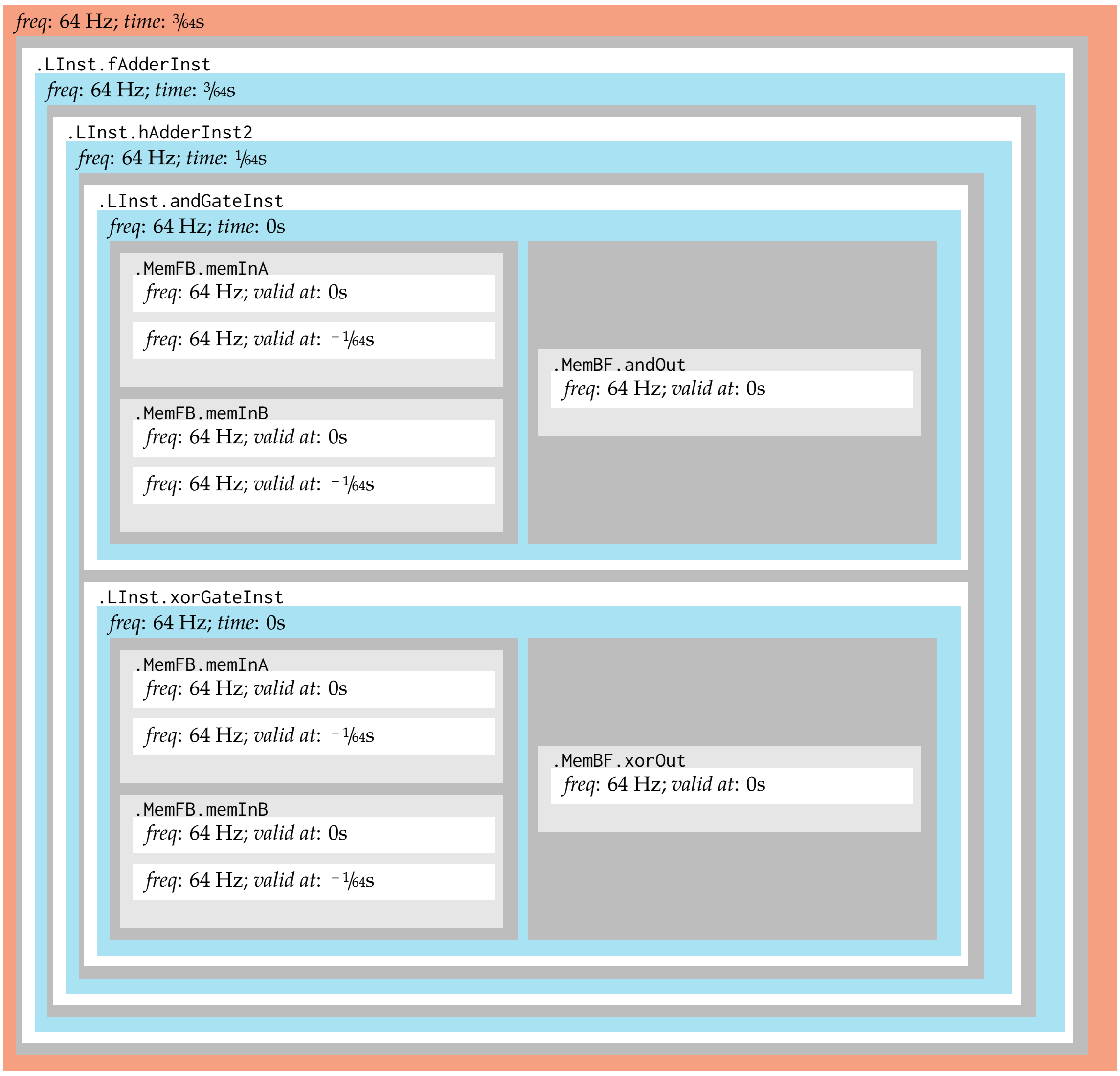
```
freq: 64 Hz; time: 3/64s
  .LInst.fAdderInst
  freq: 64 Hz; time: 3/64s
    .LInst.hAdderInst1
    freq: 64 Hz; time: 3/64s
      .LInst.andGateInst
      freq: 64 Hz; time: 2/64s
        .MemFB.memInA
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemFB.memInB
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemBF.andOut
          freq: 64 Hz; valid at: 2/64s F
      .LInst.xorGateInst
      freq: 64 Hz; time: 2/64s
        .MemFB.memInA
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemFB.memInB
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemBF.xorOut
          freq: 64 Hz; valid at: 2/64s F
```

Listing F.6(xviii): The full adder example



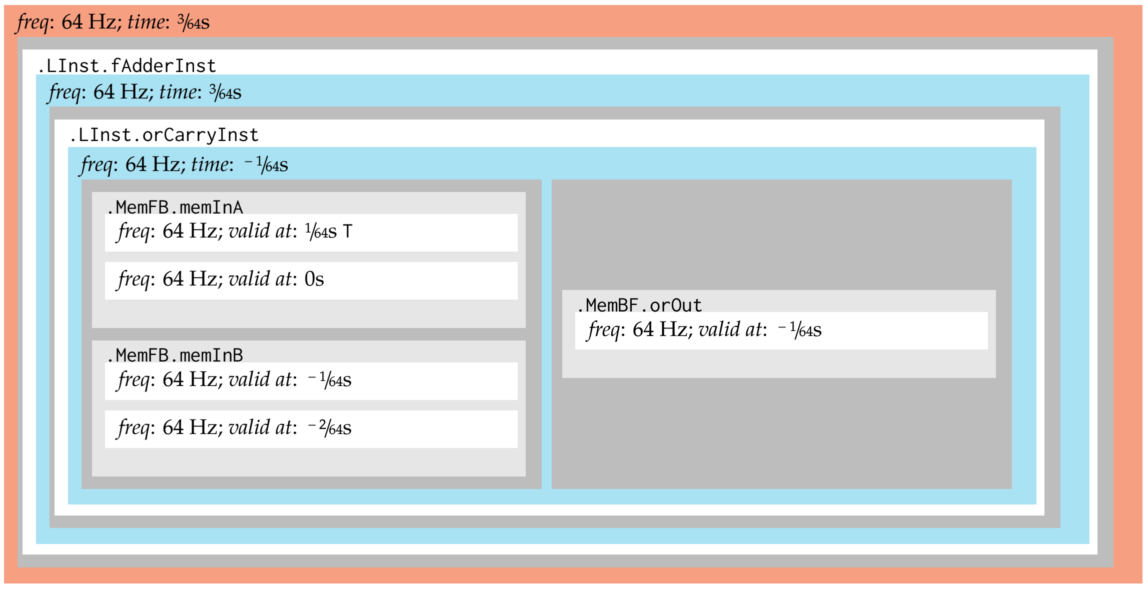
freq: 64 Hz; *time*: 3/64s
.LInst.fAdderInst
freq: 64 Hz; *time*: 3/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 1/64s
.LInst.fanoutInst1
freq: 64 Hz; *time*: 1/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1/64s F
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1/64s F
.LInst.fanoutInst2
freq: 64 Hz; *time*: 1/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 3/64s F
freq: 64 Hz; *valid at*: 2/64s F
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1/64s T

Listing F.6(xix): The full adder example



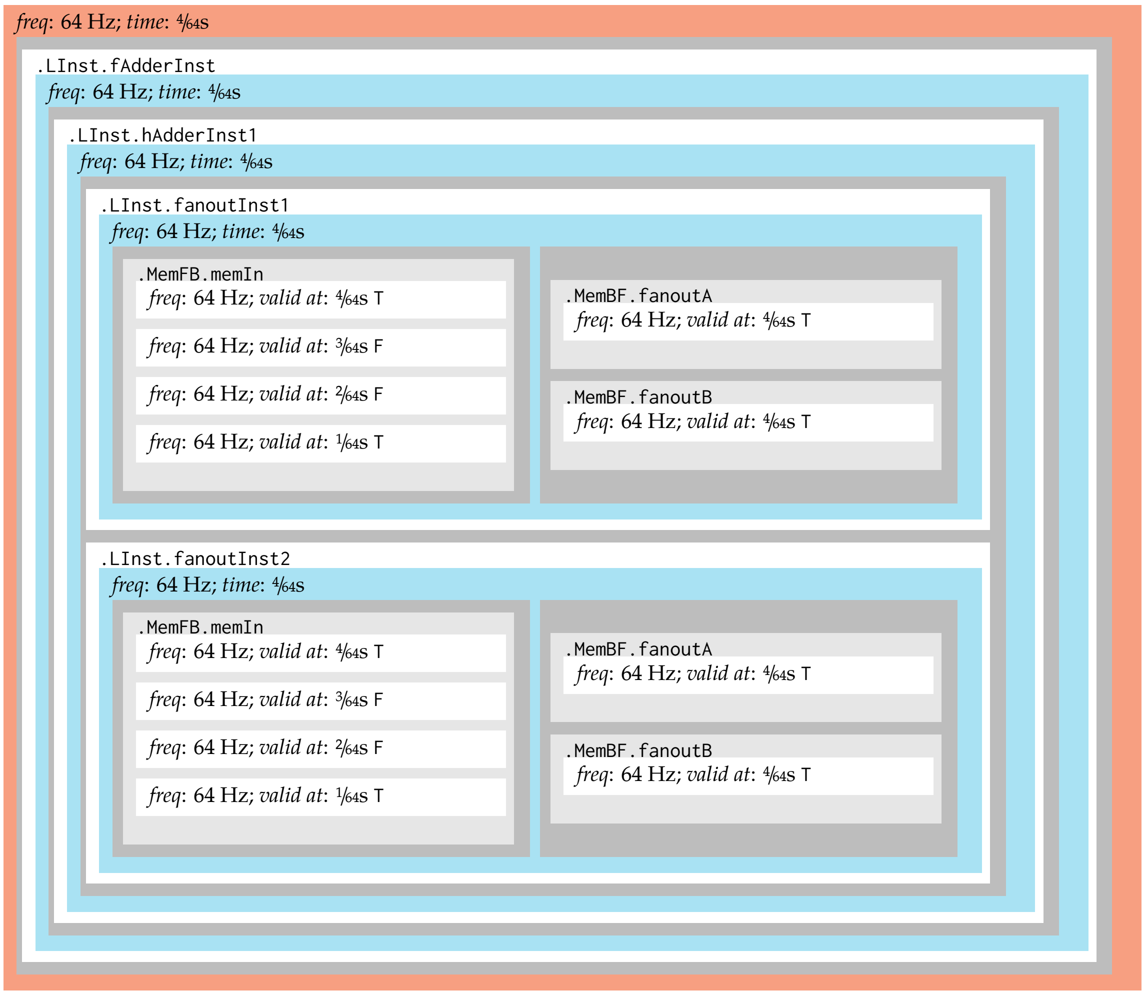
freq: 64 Hz; *time*: $\frac{3}{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $\frac{3}{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.andGateInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
freq: 64 Hz; *valid at*: 0s
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.MemFB.memInB
freq: 64 Hz; *valid at*: 0s
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.MemBF.andOut
freq: 64 Hz; *valid at*: 0s
.LInst.xorGateInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
freq: 64 Hz; *valid at*: 0s
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.MemFB.memInB
freq: 64 Hz; *valid at*: 0s
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 0s

Listing F.6(xx): The full adder example



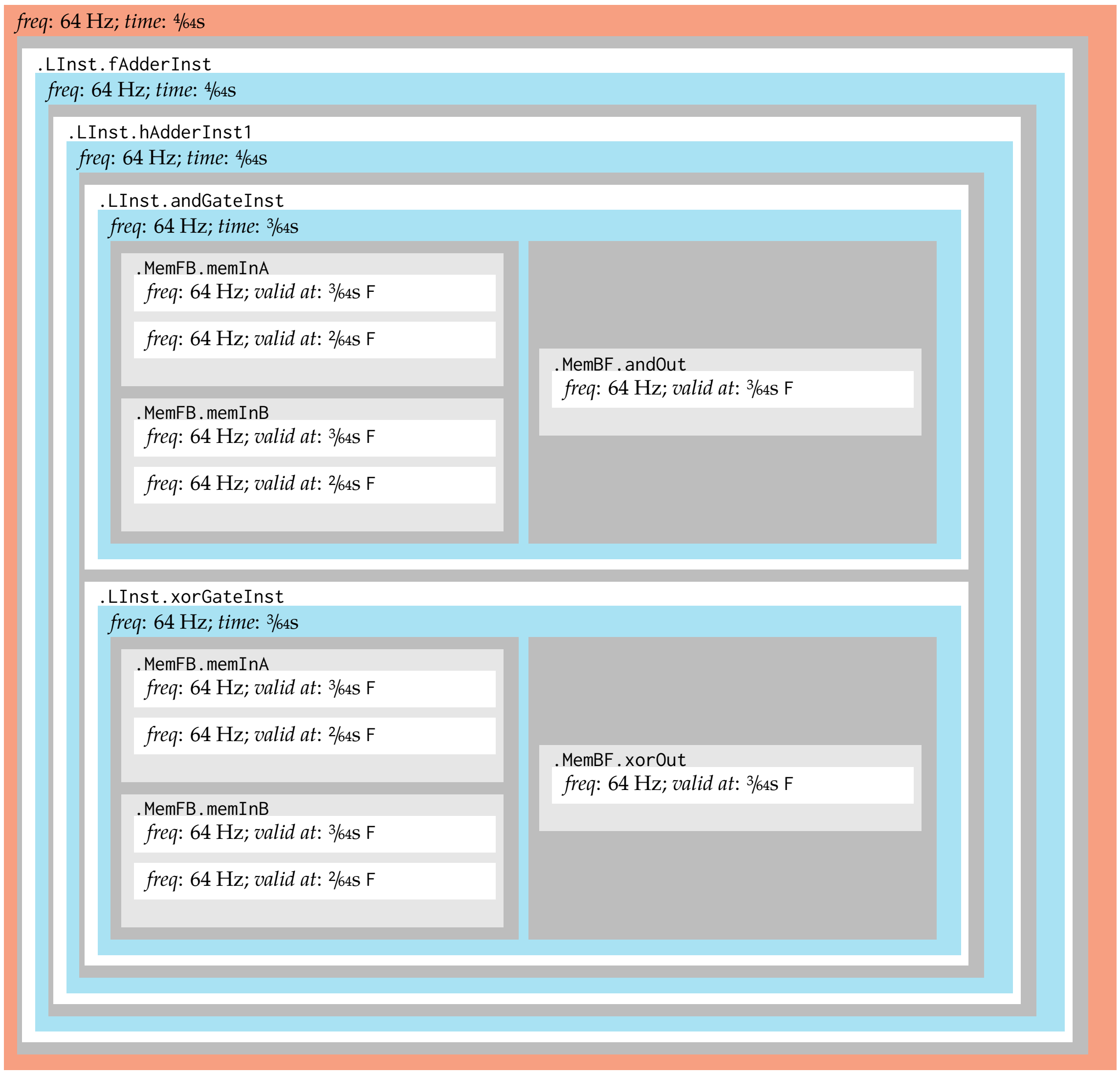
freq: 64 Hz; *time*: 3/64s
.LInst.fAdderInst
freq: 64 Hz; *time*: 3/64s
.LInst.orCarryInst
freq: 64 Hz; *time*: −1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: −1/64s
freq: 64 Hz; *valid at*: −2/64s
.MemBF.orOut
freq: 64 Hz; *valid at*: −1/64s

Listing F.6(xxi): The full adder example



```
freq: 64 Hz; time: 4/64s
  .LInst.fAdderInst
  freq: 64 Hz; time: 4/64s
    .LInst.hAdderInst1
    freq: 64 Hz; time: 4/64s
      .LInst.fanoutInst1
      freq: 64 Hz; time: 4/64s
        .MemFB.memIn
          freq: 64 Hz; valid at: 4/64s T
          freq: 64 Hz; valid at: 3/64s F
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemBF.fanoutA
          freq: 64 Hz; valid at: 4/64s T
        .MemBF.fanoutB
          freq: 64 Hz; valid at: 4/64s T
      .LInst.fanoutInst2
      freq: 64 Hz; time: 4/64s
        .MemFB.memIn
          freq: 64 Hz; valid at: 4/64s T
          freq: 64 Hz; valid at: 3/64s F
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemBF.fanoutA
          freq: 64 Hz; valid at: 4/64s T
        .MemBF.fanoutB
          freq: 64 Hz; valid at: 4/64s T
```

Listing F.6(xxii): The full adder example

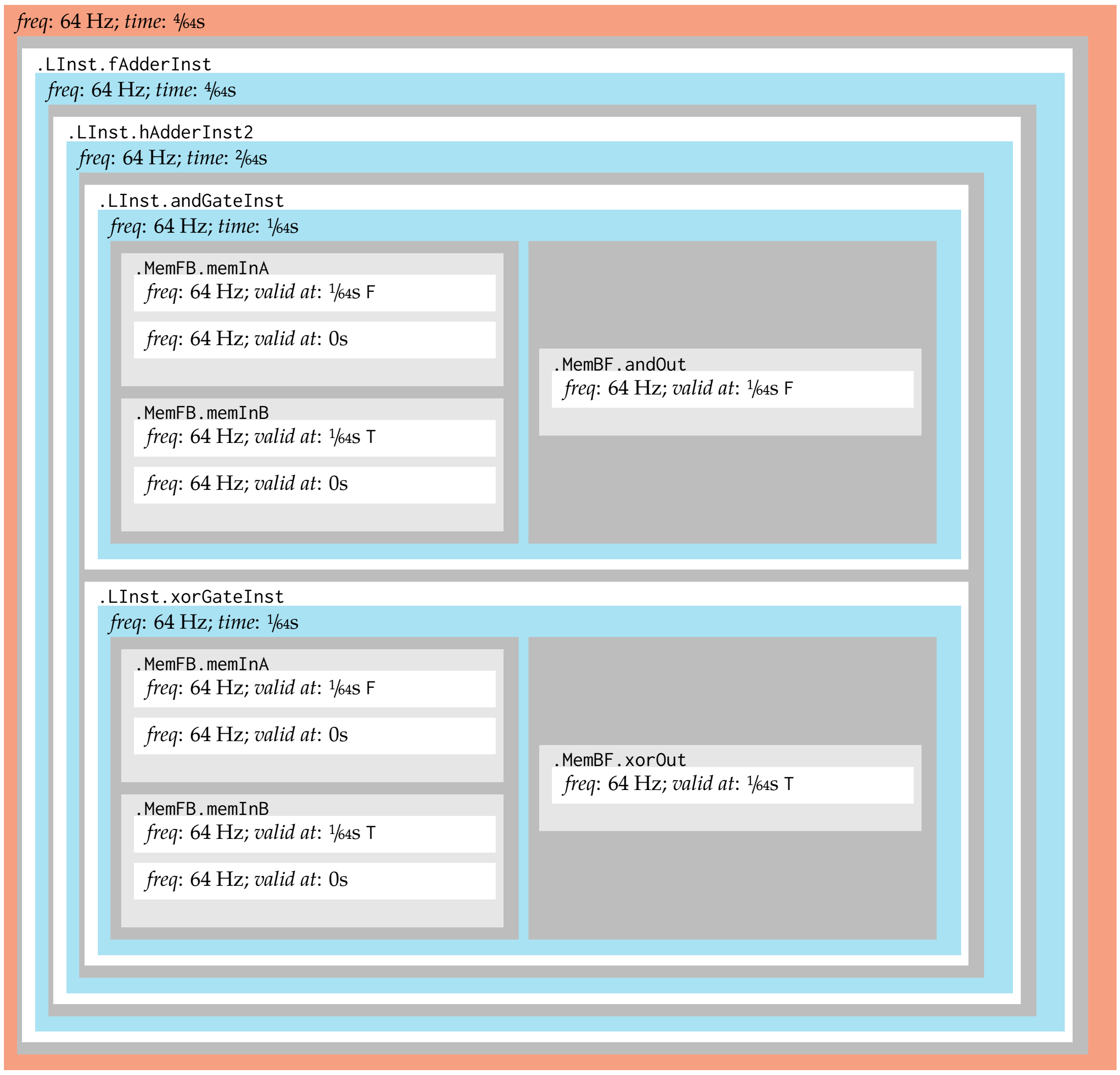


freq: 64 Hz; *time*: $^4/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^4/_{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $^4/_{64}$s
.LInst.andGateInst
freq: 64 Hz; *time*: $^3/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^3/_{64}$s F
freq: 64 Hz; *valid at*: $^2/_{64}$s F
.MemFB.memInB
freq: 64 Hz; *valid at*: $^3/_{64}$s F
freq: 64 Hz; *valid at*: $^2/_{64}$s F
.MemBF.andOut
freq: 64 Hz; *valid at*: $^3/_{64}$s F
.LInst.xorGateInst
freq: 64 Hz; *time*: $^3/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^3/_{64}$s F
freq: 64 Hz; *valid at*: $^2/_{64}$s F
.MemFB.memInB
freq: 64 Hz; *valid at*: $^3/_{64}$s F
freq: 64 Hz; *valid at*: $^2/_{64}$s F
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^3/_{64}$s F

Listing F.6(xxiii): The full adder example

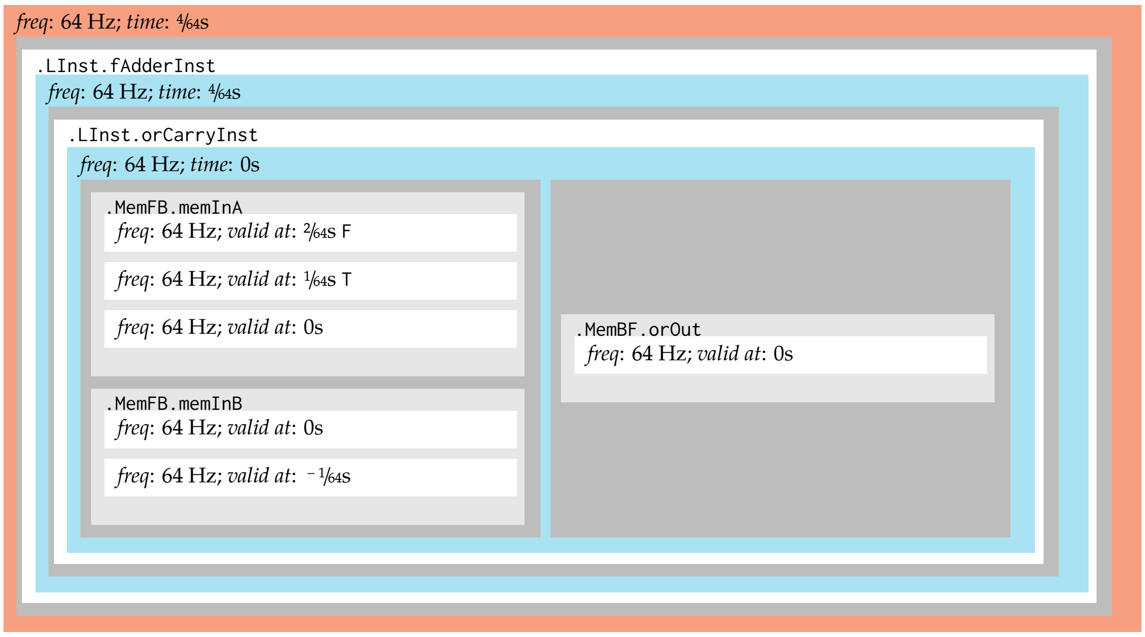
```
freq: 64 Hz; time: 4/64s
  .LInst.fAdderInst
  freq: 64 Hz; time: 4/64s
    .LInst.hAdderInst2
    freq: 64 Hz; time: 2/64s
      .LInst.fanoutInst1
      freq: 64 Hz; time: 2/64s
        .MemFB.memIn
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s F
          freq: 64 Hz; valid at: 0s
        .MemBF.fanoutA
          freq: 64 Hz; valid at: 2/64s F
        .MemBF.fanoutB
          freq: 64 Hz; valid at: 2/64s F
      .LInst.fanoutInst2
      freq: 64 Hz; time: 2/64s
        .MemFB.memIn
          freq: 64 Hz; valid at: 4/64s T
          freq: 64 Hz; valid at: 3/64s F
          freq: 64 Hz; valid at: 2/64s F
          freq: 64 Hz; valid at: 1/64s T
        .MemBF.fanoutA
          freq: 64 Hz; valid at: 2/64s F
        .MemBF.fanoutB
          freq: 64 Hz; valid at: 2/64s F
```

Listing F.6(xxiv): The full adder example



freq: 64 Hz; *time*: $4/64$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $4/64$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $2/64$s
.LInst.andGateInst
freq: 64 Hz; *time*: $1/64$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $1/64$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $1/64$s T
freq: 64 Hz; *valid at*: 0s
.MemBF.andOut
freq: 64 Hz; *valid at*: $1/64$s F
.LInst.xorGateInst
freq: 64 Hz; *time*: $1/64$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $1/64$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $1/64$s T
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: $1/64$s T

Listing F.6(xxv): The full adder example


freq: 64 Hz; *time*: $\frac{4}{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $\frac{4}{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: 0s
.MemFB.memInA
freq: 64 Hz; *valid at*: $\frac{2}{64}$s F
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 0s
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.MemBF.orOut
freq: 64 Hz; *valid at*: 0s

Listing F.6(xxvi): The full adder example

```
freq: 64 Hz; time: 5/64s
  .LInst.fAdderInst
  freq: 64 Hz; time: 5/64s
    .LInst.hAdderInst1
    freq: 64 Hz; time: 5/64s
      .LInst.fanoutInst1
      freq: 64 Hz; time: 5/64s
        .MemFB.memIn
          freq: 64 Hz; valid at: 5/64s F
          freq: 64 Hz; valid at: 4/64s T
          freq: 64 Hz; valid at: 3/64s F
          freq: 64 Hz; valid at: 2/64s F
        .MemBF.fanoutA
          freq: 64 Hz; valid at: 5/64s F
        .MemBF.fanoutB
          freq: 64 Hz; valid at: 5/64s F
      .LInst.fanoutInst2
      freq: 64 Hz; time: 5/64s
        .MemFB.memIn
          freq: 64 Hz; valid at: 5/64s F
          freq: 64 Hz; valid at: 4/64s T
          freq: 64 Hz; valid at: 3/64s F
          freq: 64 Hz; valid at: 2/64s F
        .MemBF.fanoutA
          freq: 64 Hz; valid at: 5/64s F
        .MemBF.fanoutB
          freq: 64 Hz; valid at: 5/64s F
```

Listing F.6(xxvii): The full adder example

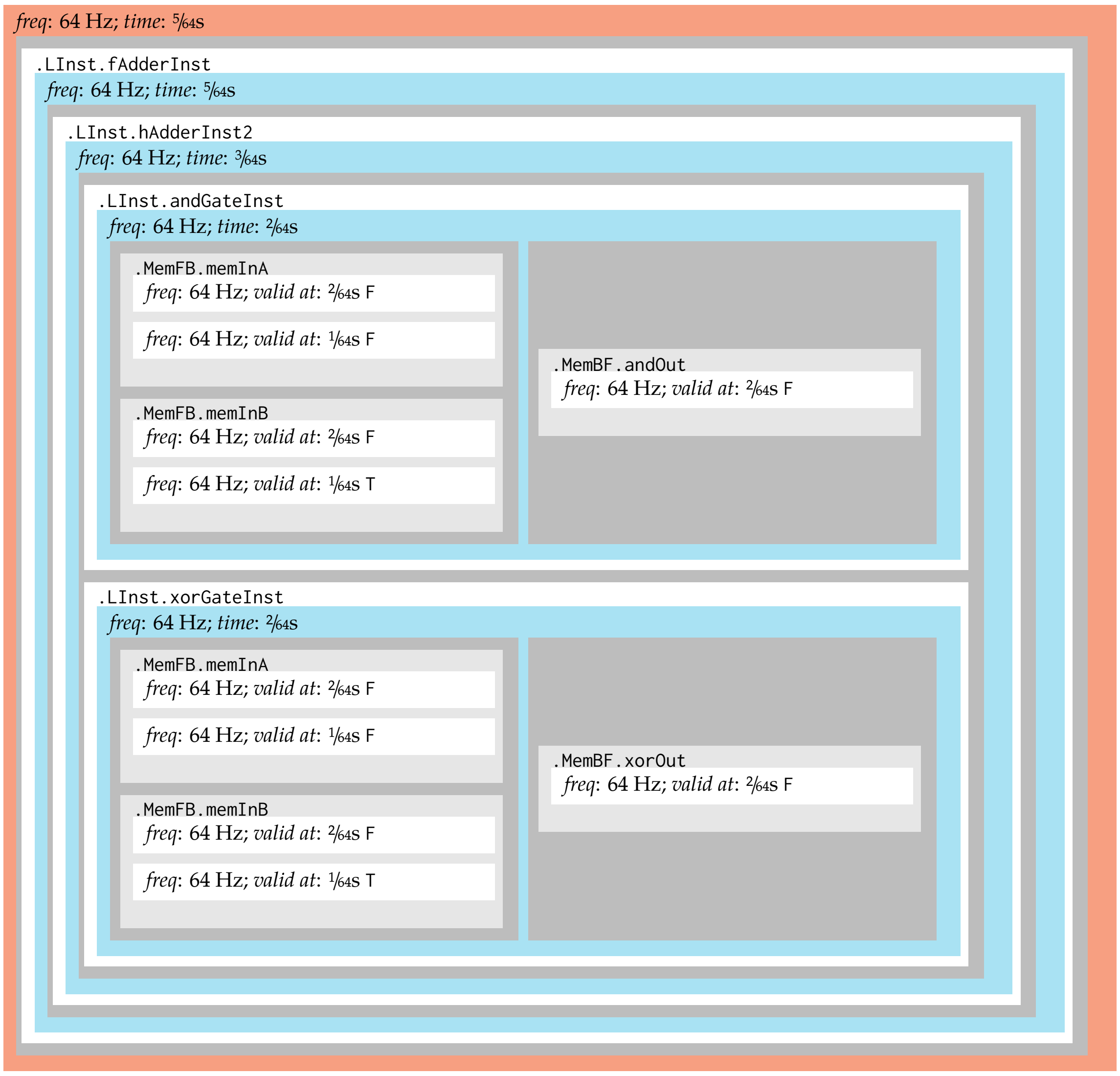
- *freq*: 64 Hz; *time*: $^{5}/_{64}$s
  - `.LInst.fAdderInst`
    - *freq*: 64 Hz; *time*: $^{5}/_{64}$s
      - `.LInst.hAdderInst1`
        - *freq*: 64 Hz; *time*: $^{5}/_{64}$s
          - `.LInst.andGateInst`
            - *freq*: 64 Hz; *time*: $^{4}/_{64}$s
              - `.MemFB.memInA`
                - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `F`
              - `.MemFB.memInB`
                - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `F`
              - `.MemBF.andOut`
                - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `T`
          - `.LInst.xorGateInst`
            - *freq*: 64 Hz; *time*: $^{4}/_{64}$s
              - `.MemFB.memInA`
                - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `F`
              - `.MemFB.memInB`
                - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s `F`
              - `.MemBF.xorOut`
                - *freq*: 64 Hz; *valid at*: $^{4}/_{64}$s `F`

Listing F.6(xxviii): The full adder example

freq: 64 Hz; *time*: $\frac{5}{64}$s

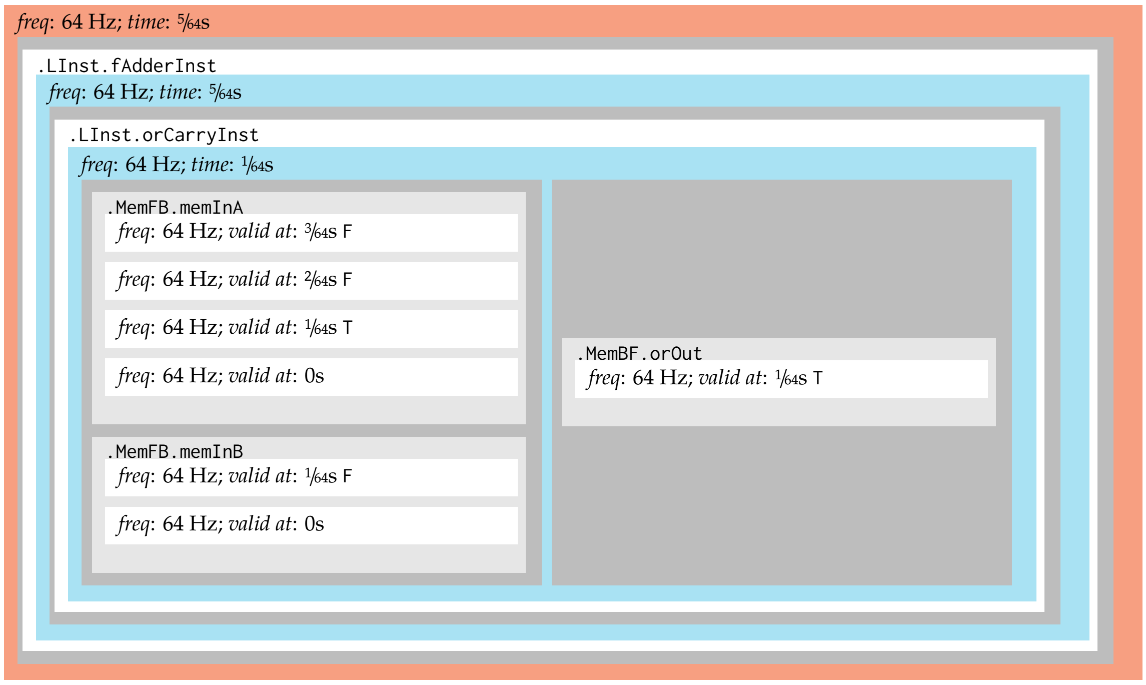
- `.LInst.fAdderInst`
  - *freq*: 64 Hz; *time*: $\frac{5}{64}$s
    - `.LInst.hAdderInst2`
      - *freq*: 64 Hz; *time*: $\frac{3}{64}$s
        - `.LInst.fanoutInst1`
          - *freq*: 64 Hz; *time*: $\frac{3}{64}$s
            - `.MemFB.memIn`
              - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s F
              - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s F
              - *freq*: 64 Hz; *valid at*: 0s
            - `.MemBF.fanoutA`
              - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s F
            - `.MemBF.fanoutB`
              - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s F
        - `.LInst.fanoutInst2`
          - *freq*: 64 Hz; *time*: $\frac{3}{64}$s
            - `.MemFB.memIn`
              - *freq*: 64 Hz; *valid at*: $\frac{5}{64}$s F
              - *freq*: 64 Hz; *valid at*: $\frac{4}{64}$s T
              - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s F
              - *freq*: 64 Hz; *valid at*: $\frac{2}{64}$s F
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
            - `.MemBF.fanoutA`
              - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s F
            - `.MemBF.fanoutB`
              - *freq*: 64 Hz; *valid at*: $\frac{3}{64}$s F

Listing F.6(xxix): The full adder example



freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.fAdderInst
freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $^{3}/_{64}$s
.LInst.andGateInst
freq: 64 Hz; *time*: $^{2}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemBF.andOut
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
.LInst.xorGateInst
freq: 64 Hz; *time*: $^{2}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F

Listing F.6(xxx): The full adder example



freq: 64 Hz; *time*: $^5/_{64}$s

```
.LInst.fAdderInst
```

freq: 64 Hz; *time*: $^5/_{64}$s

```
.LInst.orCarryInst
```

freq: 64 Hz; *time*: $^1/_{64}$s

```
.MemFB.memInA
```

freq: 64 Hz; *valid at*: $^3/_{64}$s `F`

freq: 64 Hz; *valid at*: $^2/_{64}$s `F`

freq: 64 Hz; *valid at*: $^1/_{64}$s `T`

freq: 64 Hz; *valid at*: 0s

```
.MemFB.memInB
```

freq: 64 Hz; *valid at*: $^1/_{64}$s `F`

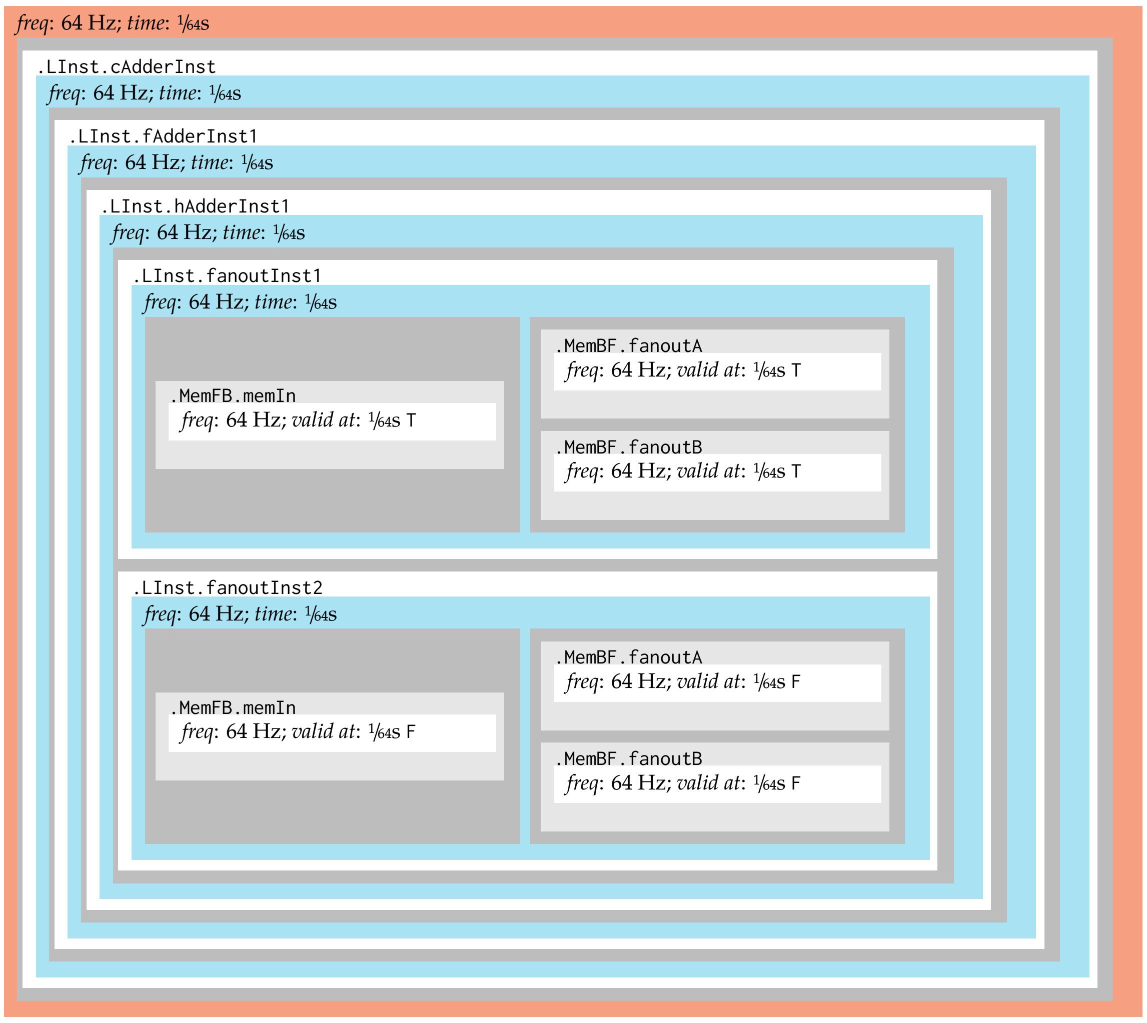
freq: 64 Hz; *valid at*: 0s

```
.MemBF.orOut
```

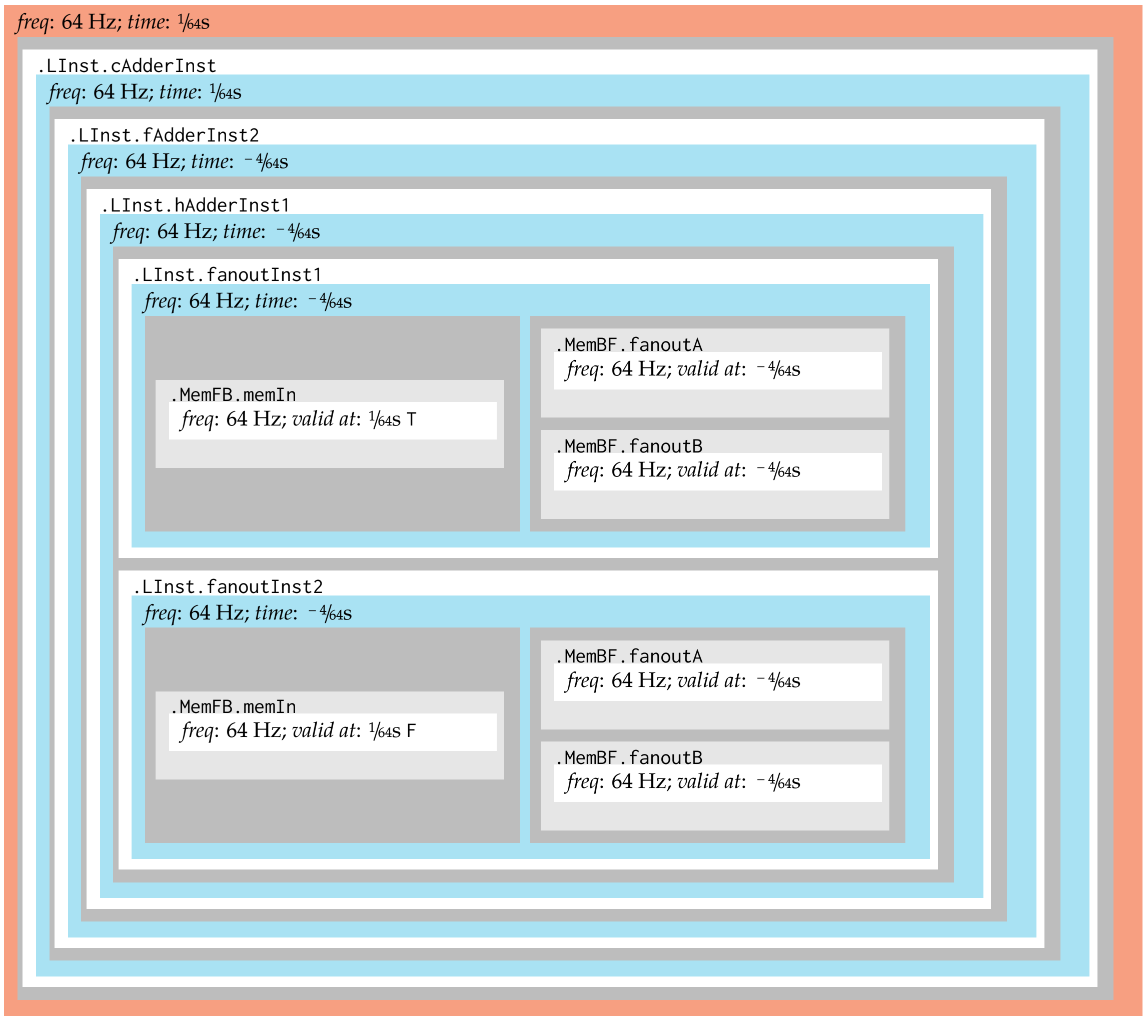
freq: 64 Hz; *valid at*: $^1/_{64}$s `T`

F.7 Cascade adder

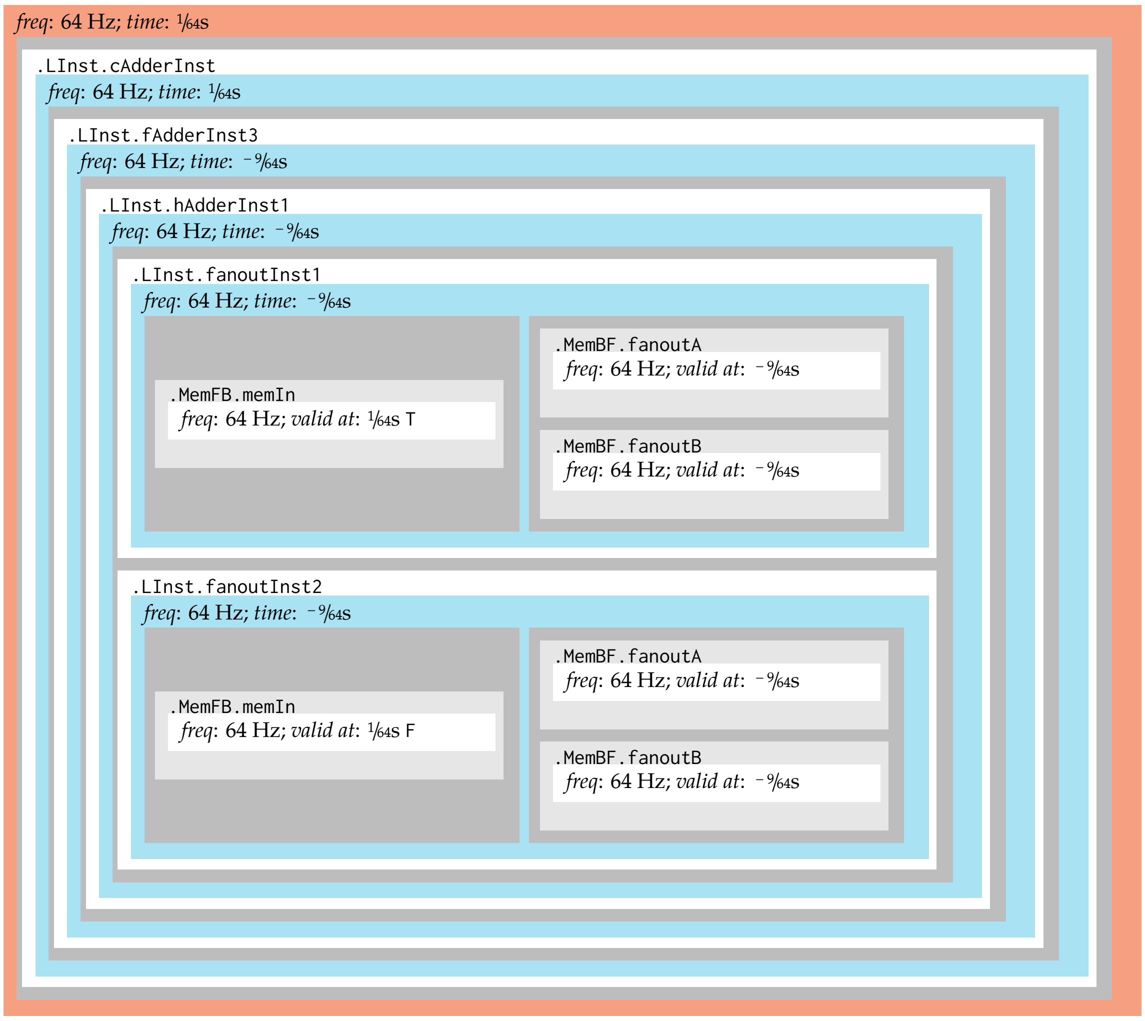
Listing F.7(i): The cascade adder


freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.fAdderInst1
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $^1/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.LInst.fanoutInst2
freq: 64 Hz; *time*: $^1/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $^1/_{64}$s F
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $^1/_{64}$s F

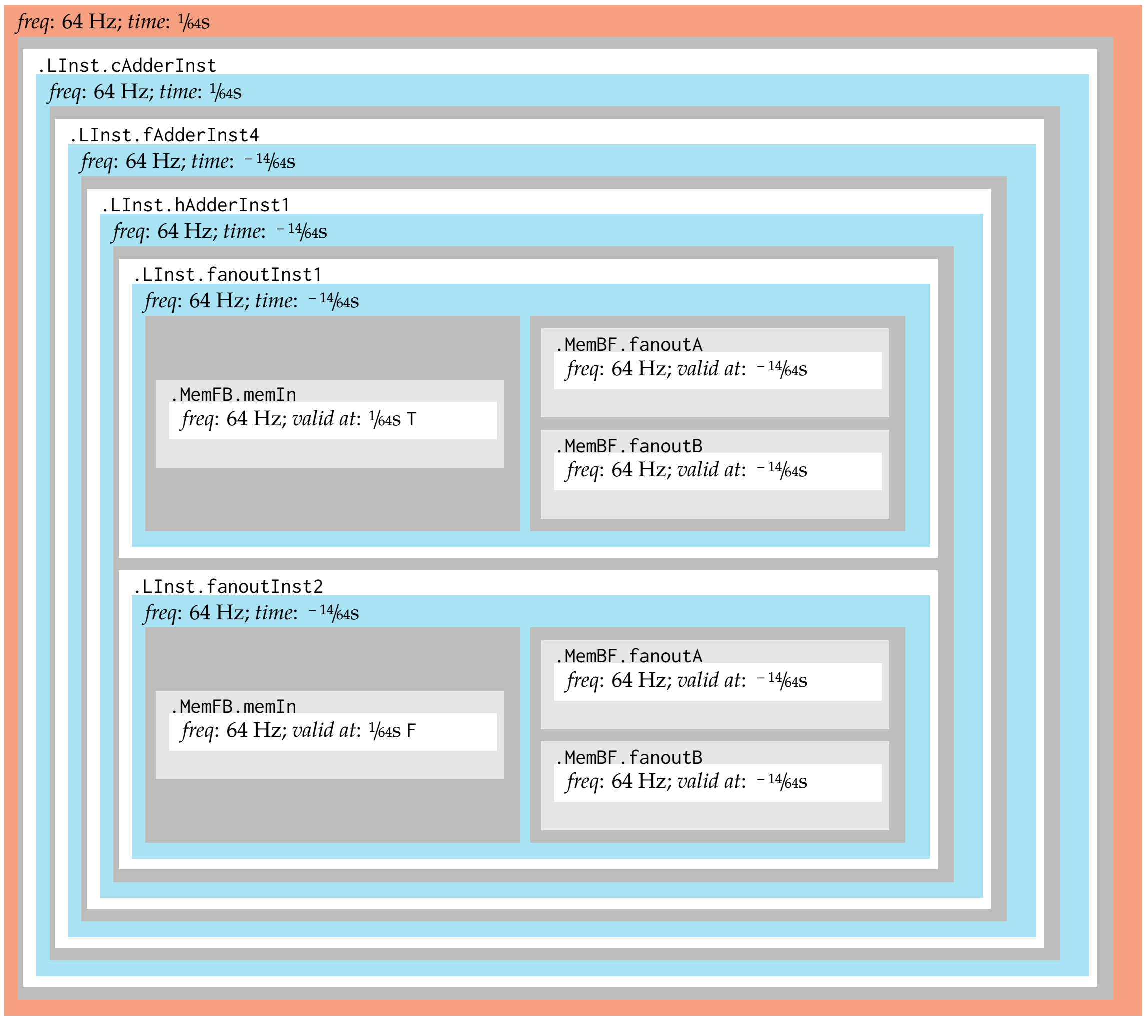
Listing F.7(ii): The cascade adder


freq: 64 Hz; *time*: 1/64s
.LInst.cAdderInst
freq: 64 Hz; *time*: 1/64s
.LInst.fAdderInst2
freq: 64 Hz; *time*: −4/64s
.LInst.hAdderInst1
freq: 64 Hz; *time*: −4/64s
.LInst.fanoutInst1
freq: 64 Hz; *time*: −4/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: −4/64s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: −4/64s
.LInst.fanoutInst2
freq: 64 Hz; *time*: −4/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: −4/64s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: −4/64s

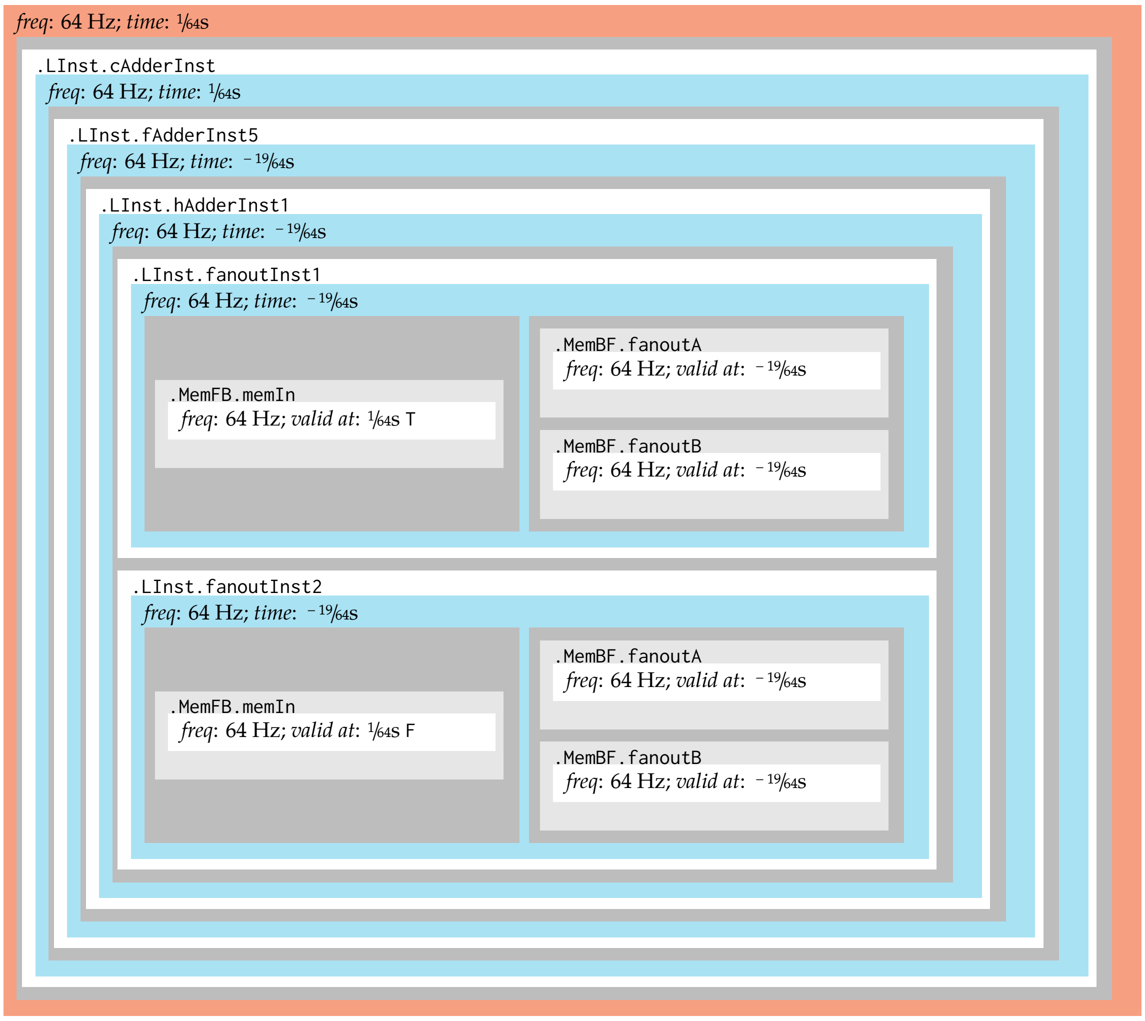
Listing F.7(iii): The cascade adder


freq: 64 Hz; *time*: $^{1}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.LInst.fAdderInst3
freq: 64 Hz; *time*: $-^{9}/_{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $-^{9}/_{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $-^{9}/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-^{9}/_{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-^{9}/_{64}$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $-^{9}/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-^{9}/_{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-^{9}/_{64}$s

Listing F.7(iv): The cascade adder


freq: 64 Hz; *time*: $1/64$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $1/64$s
.LInst.fAdderInst4
freq: 64 Hz; *time*: $-14/64$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $-14/64$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $-14/64$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $1/64$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-14/64$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-14/64$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $-14/64$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $1/64$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-14/64$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-14/64$s

Listing F.7(v): The cascade adder


freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^1/_{64}$s
.LInst.fAdderInst5
freq: 64 Hz; *time*: $-^{19}/_{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $-^{19}/_{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $-^{19}/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-^{19}/_{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-^{19}/_{64}$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $-^{19}/_{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $^1/_{64}$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-^{19}/_{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-^{19}/_{64}$s

Listing F.7(vi): The cascade adder

freq: 64 Hz; *time*: $^{1}/_{64}$s

`.LInst.cAdderInst`

freq: 64 Hz; *time*: $^{1}/_{64}$s

`.LInst.fAdderInst6`

freq: 64 Hz; *time*: $^{-24}/_{64}$s

`.LInst.hAdderInst1`

freq: 64 Hz; *time*: $^{-24}/_{64}$s

`.LInst.fanoutInst1`

freq: 64 Hz; *time*: $^{-24}/_{64}$s

`.MemFB.memIn`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `T`

`.MemBF.fanoutA`

freq: 64 Hz; *valid at*: $^{-24}/_{64}$s

`.MemBF.fanoutB`

freq: 64 Hz; *valid at*: $^{-24}/_{64}$s

`.LInst.fanoutInst2`

freq: 64 Hz; *time*: $^{-24}/_{64}$s

`.MemFB.memIn`

freq: 64 Hz; *valid at*: $^{1}/_{64}$s `F`

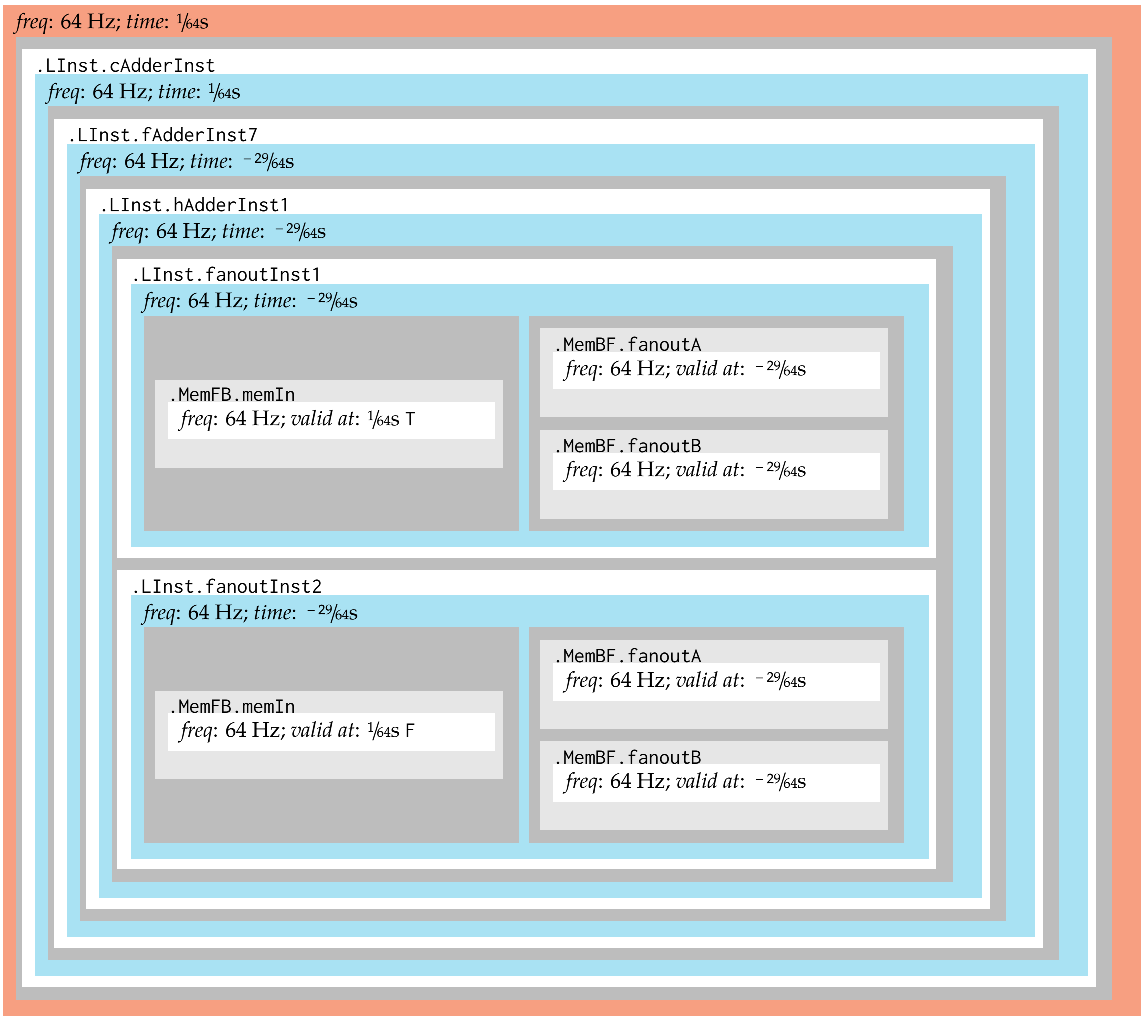
`.MemBF.fanoutA`

freq: 64 Hz; *valid at*: $^{-24}/_{64}$s

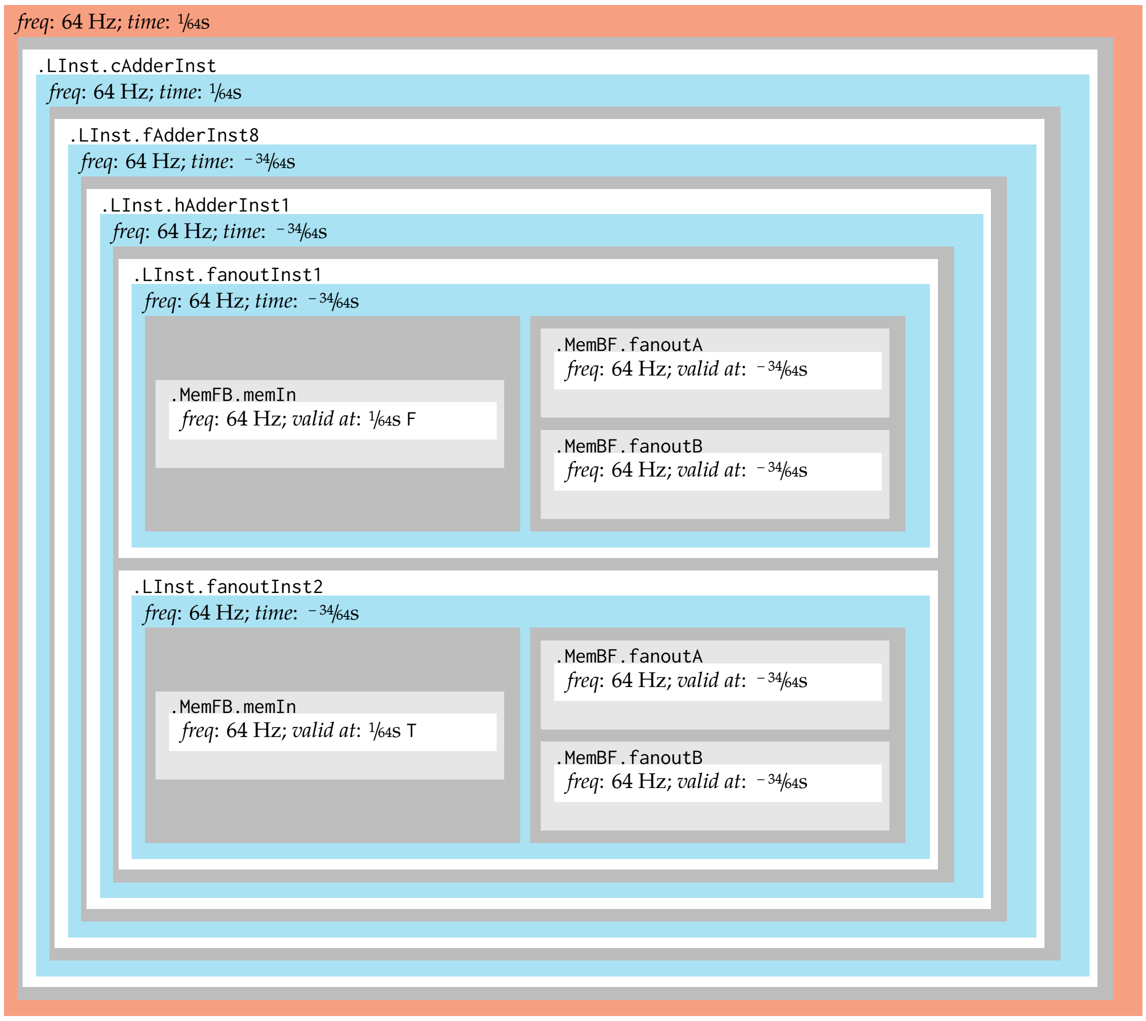
`.MemBF.fanoutB`

freq: 64 Hz; *valid at*: $^{-24}/_{64}$s

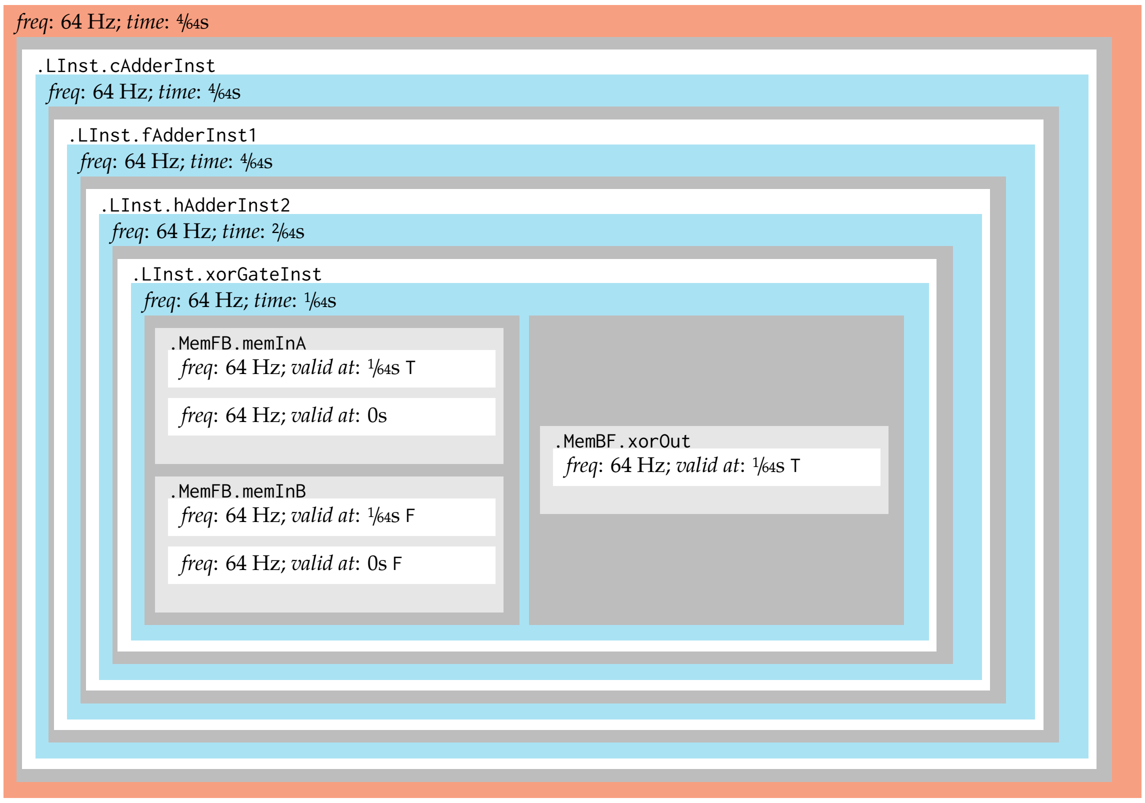
Listing F.7(vii): The cascade adder


freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.fAdderInst7
freq: 64 Hz; *time*: $-\frac{29}{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $-\frac{29}{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $-\frac{29}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{29}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{29}{64}$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $-\frac{29}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{29}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{29}{64}$s

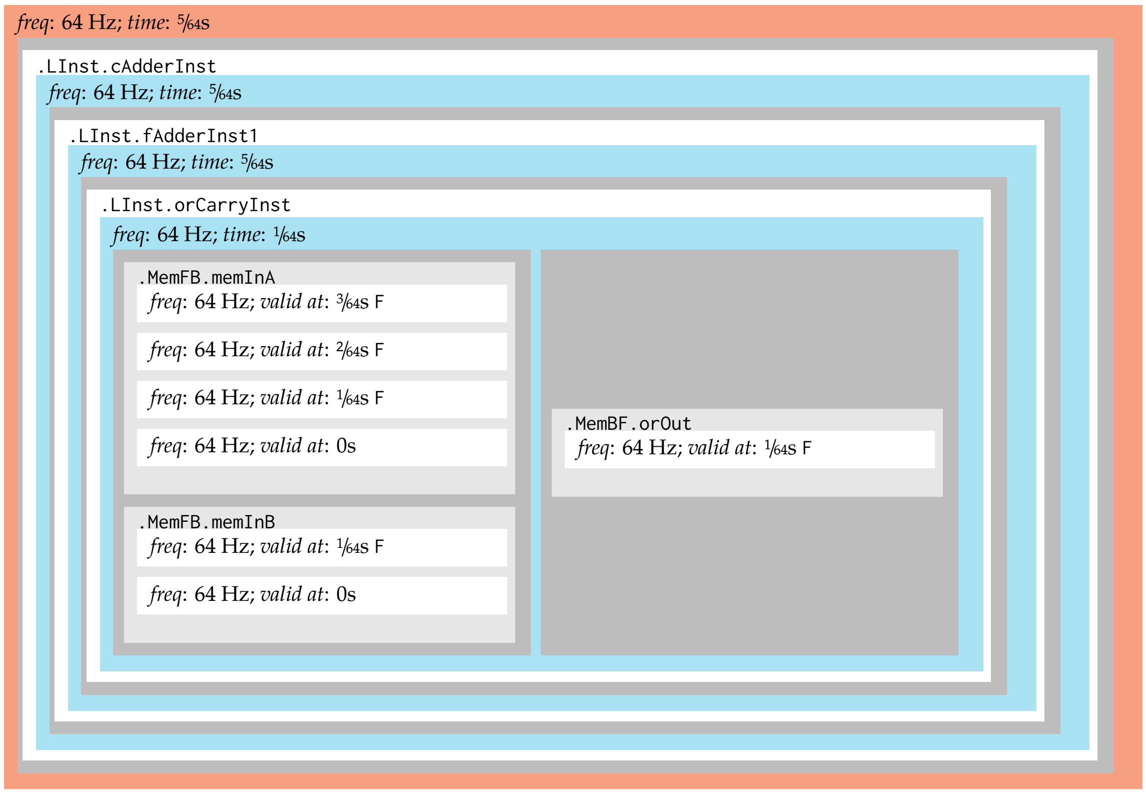
Listing F.7(viii): The cascade adder


freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.fAdderInst8
freq: 64 Hz; *time*: $-\frac{34}{64}$s
.LInst.hAdderInst1
freq: 64 Hz; *time*: $-\frac{34}{64}$s
.LInst.fanoutInst1
freq: 64 Hz; *time*: $-\frac{34}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{34}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{34}{64}$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $-\frac{34}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{34}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{34}{64}$s

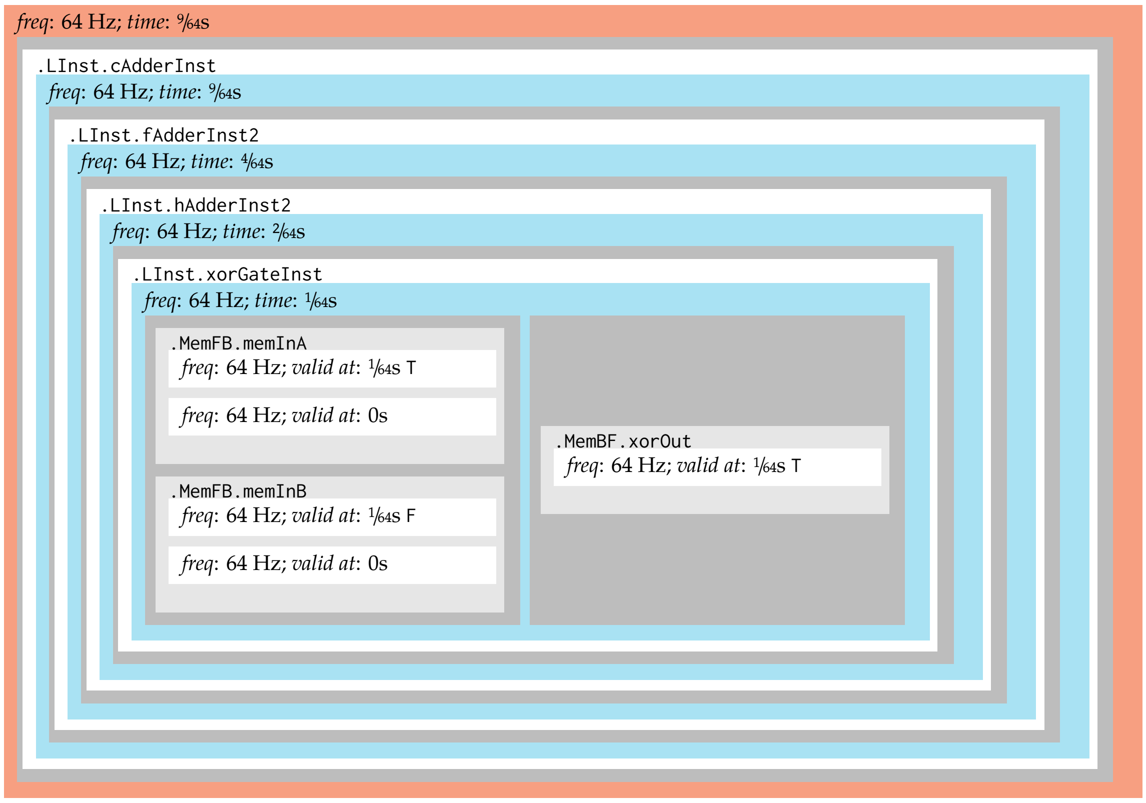
Listing F.7(ix): The cascade adder


freq: 64 Hz; *time*: $^4\!/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^4\!/_{64}$s
.LInst.fAdderInst1
freq: 64 Hz; *time*: $^4\!/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $^2\!/_{64}$s
.LInst.xorGateInst
freq: 64 Hz; *time*: $^1\!/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^1\!/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^1\!/_{64}$s F
freq: 64 Hz; *valid at*: 0s F
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^1\!/_{64}$s T

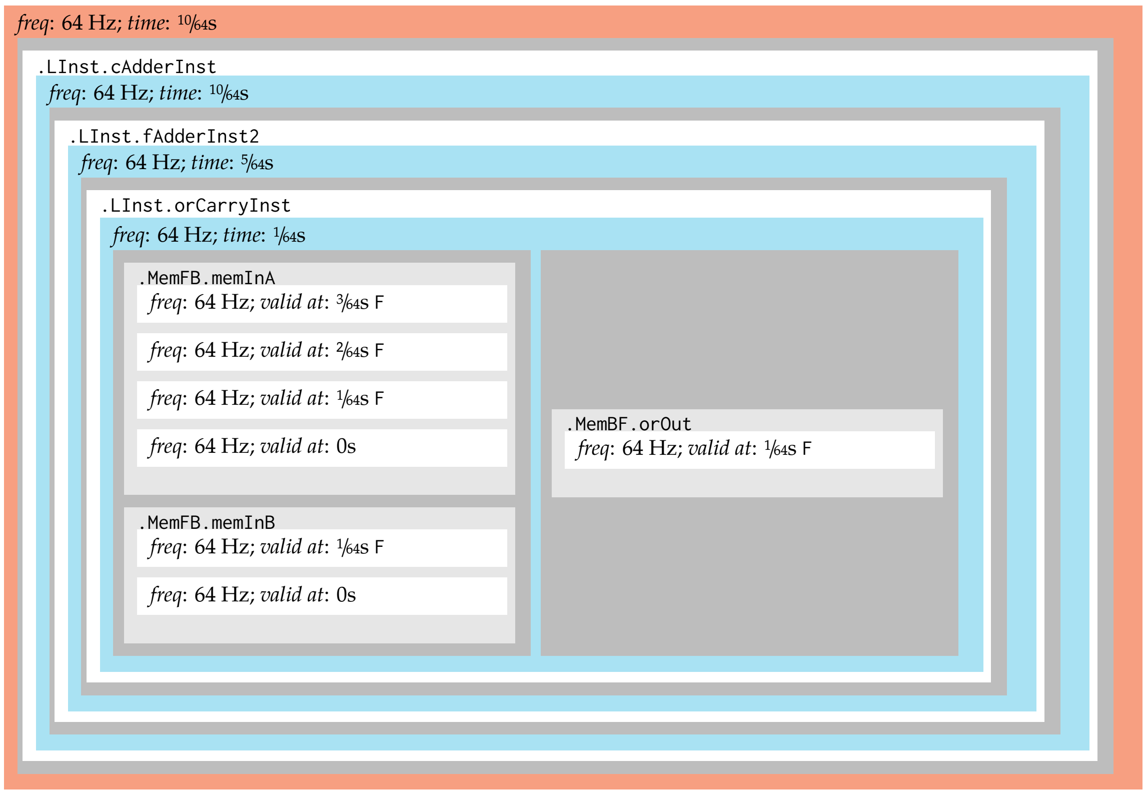
Listing F.7(x): The cascade adder


freq: 64 Hz; *time*: $^5\!/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^5\!/_{64}$s
.LInst.fAdderInst1
freq: 64 Hz; *time*: $^5\!/_{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $^1\!/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^3\!/_{64}$s F
freq: 64 Hz; *valid at*: $^2\!/_{64}$s F
freq: 64 Hz; *valid at*: $^1\!/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^1\!/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: $^1\!/_{64}$s F

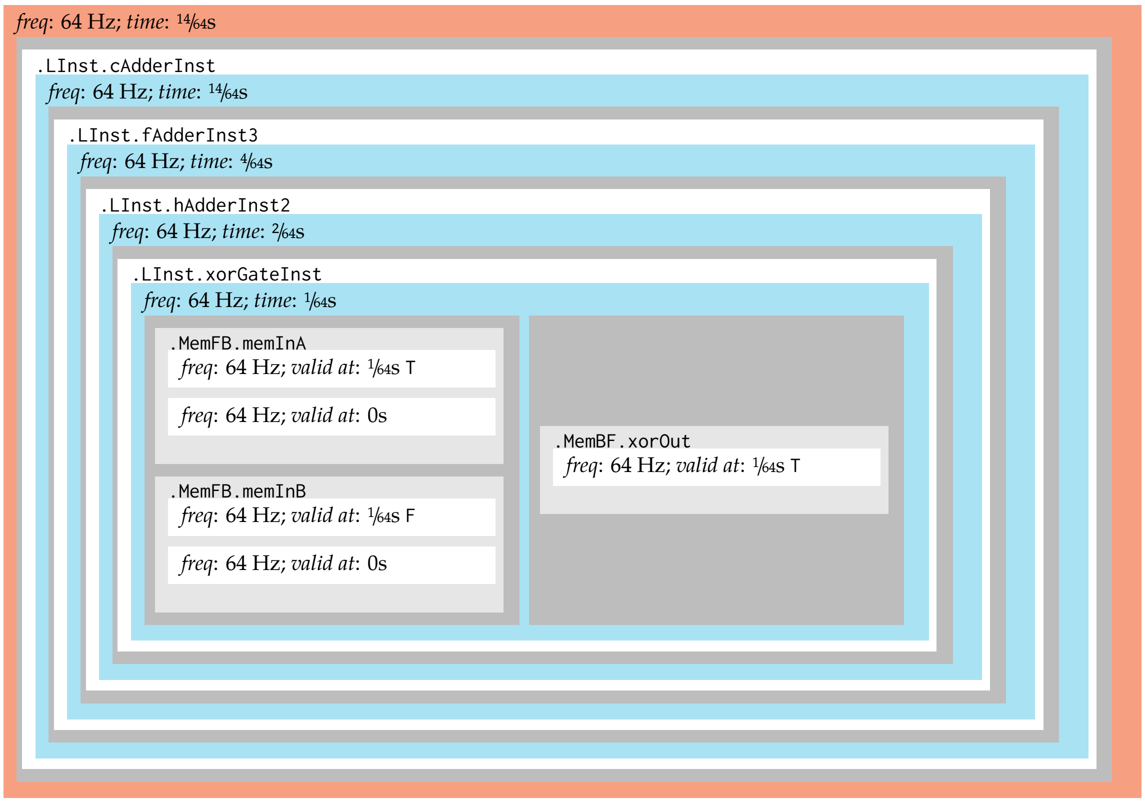
Listing F.7(xi): The cascade adder


freq: 64 Hz; *time*: 9/64s
.LInst.cAdderInst
freq: 64 Hz; *time*: 9/64s
.LInst.fAdderInst2
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s T

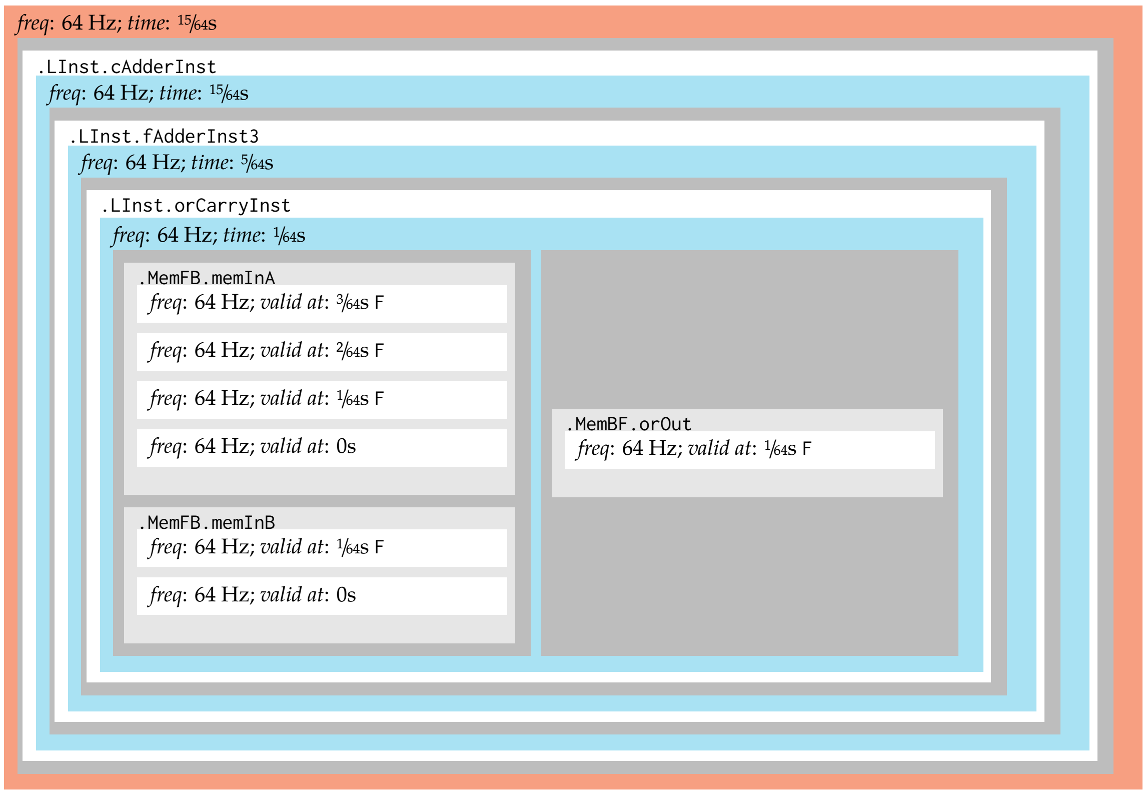
Listing F.7(xii): The cascade adder


freq: 64 Hz; *time*: $^{10}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{10}/_{64}$s
.LInst.fAdderInst2
freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{3}/_{64}$s F
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F

Listing F.7(xiii): The cascade adder

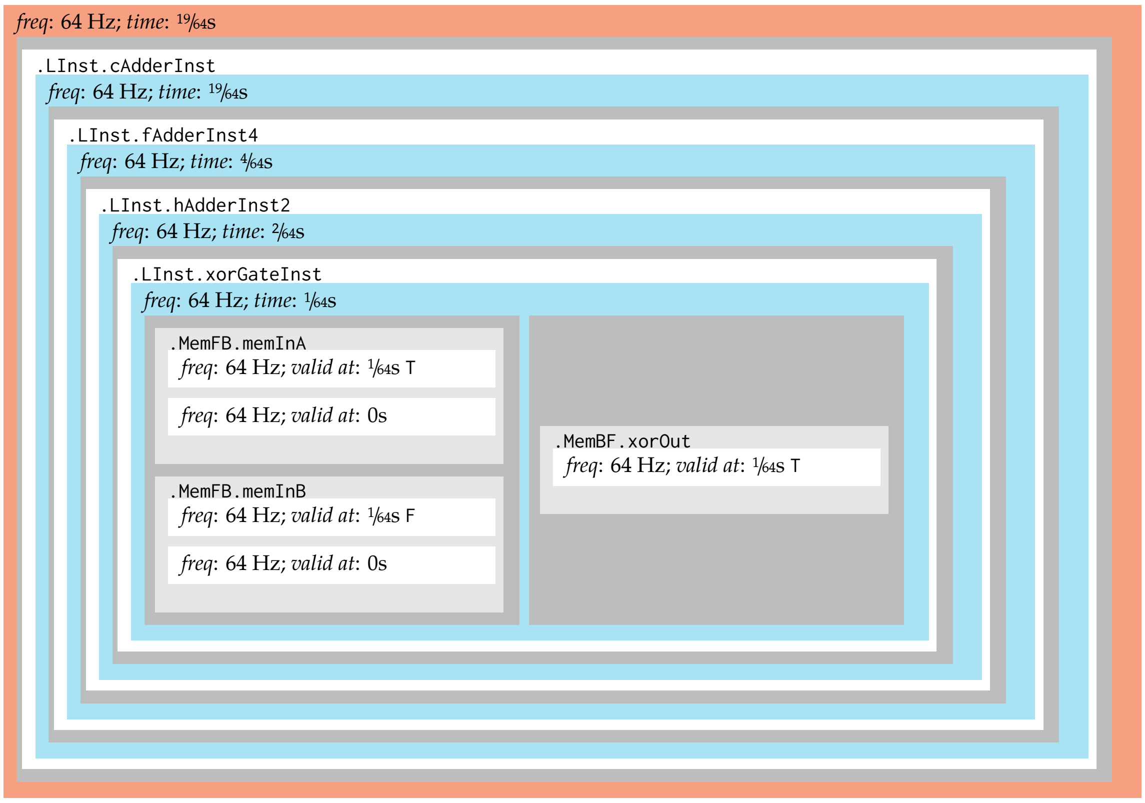

freq: 64 Hz; *time*: $^{14}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{14}/_{64}$s
.LInst.fAdderInst3
freq: 64 Hz; *time*: $^{4}/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.xorGateInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

Listing F.7(xiv): The cascade adder



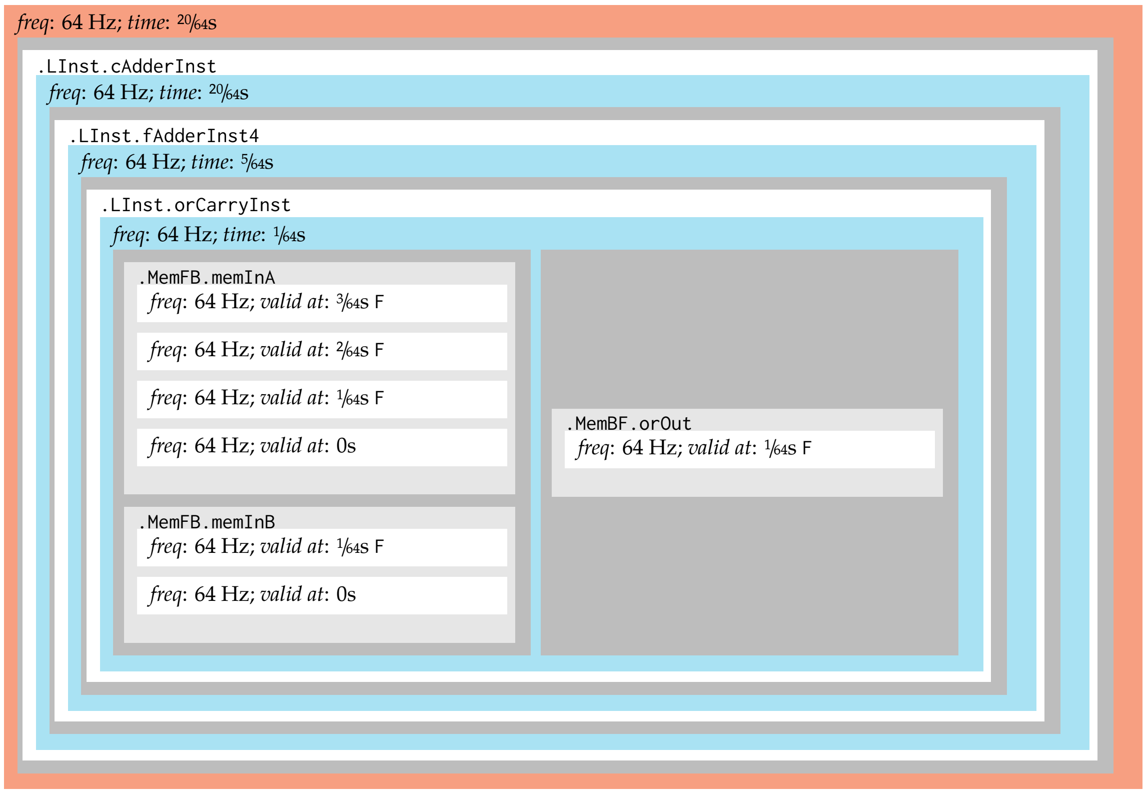
freq: 64 Hz; *time*: $\frac{15}{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $\frac{15}{64}$s
.LInst.fAdderInst3
freq: 64 Hz; *time*: $\frac{5}{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $\frac{3}{64}$s F
freq: 64 Hz; *valid at*: $\frac{2}{64}$s F
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F

Listing F.7(xv): The cascade adder



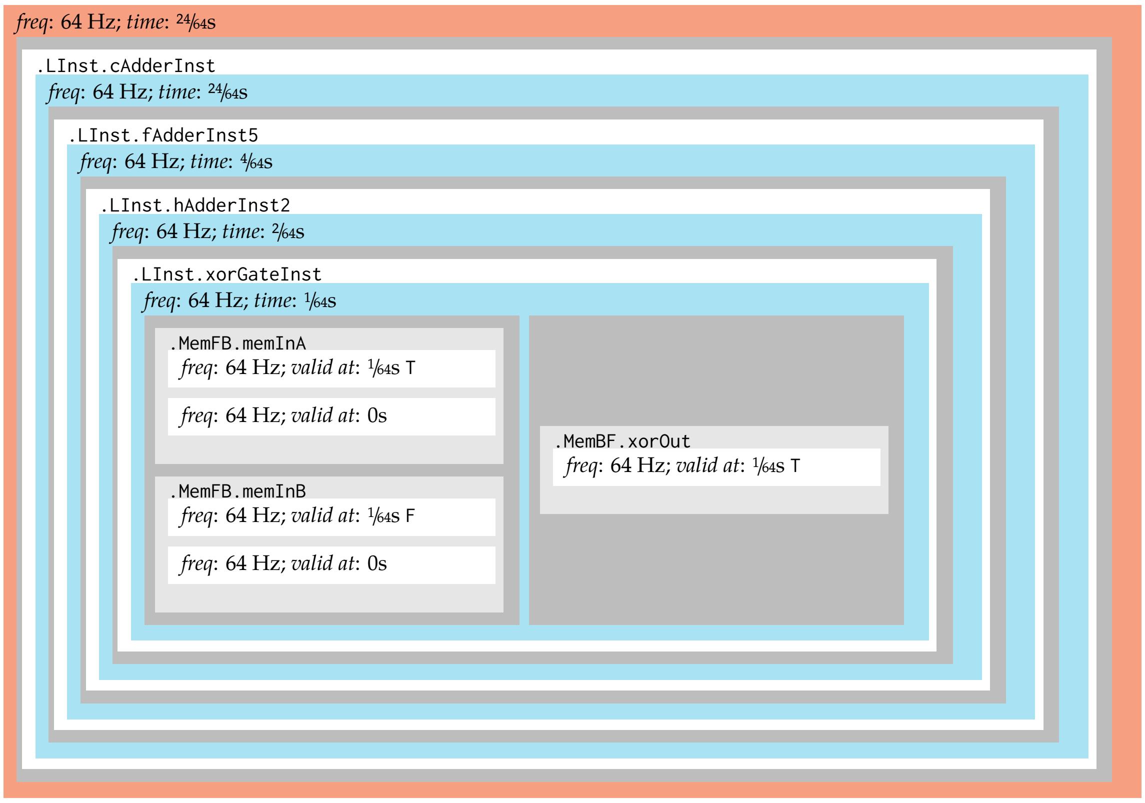
freq: 64 Hz; *time*: 19/64s
.LInst.cAdderInst
freq: 64 Hz; *time*: 19/64s
.LInst.fAdderInst4
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s T

Listing F.7(xvi): The cascade adder

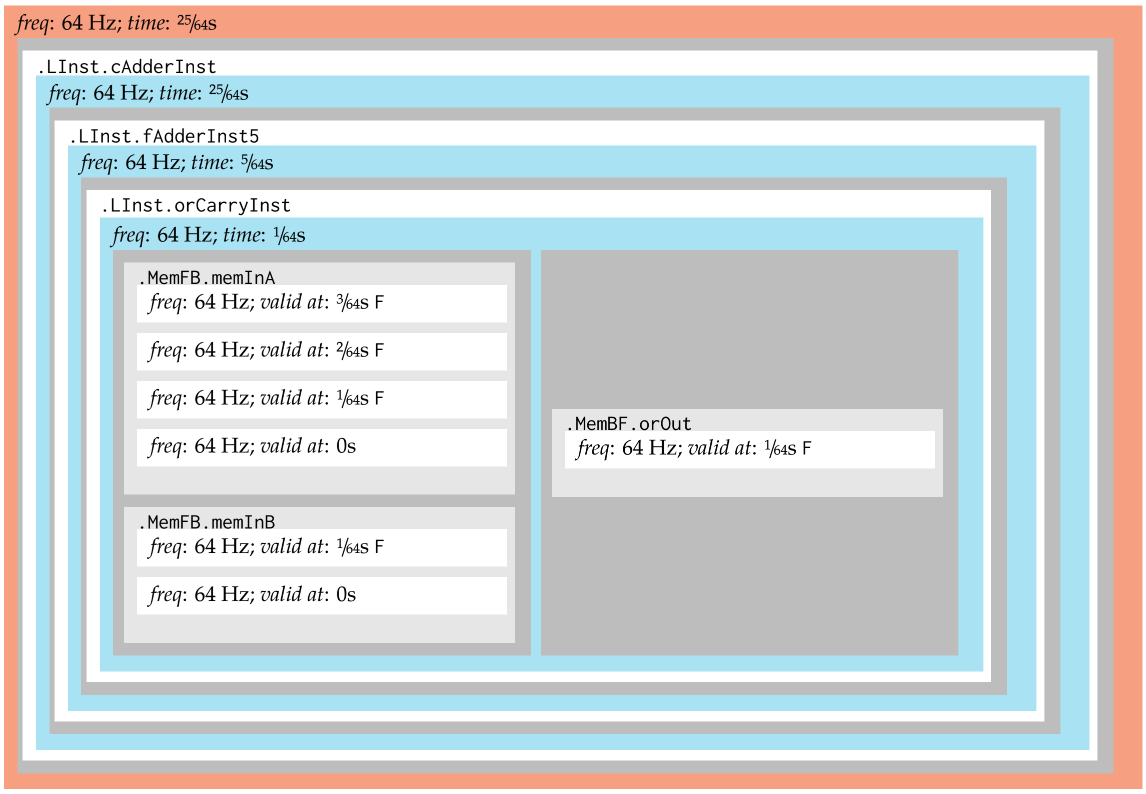


- *freq*: 64 Hz; *time*: $^{20}/_{64}$s
  - `.LInst.cAdderInst`
  - *freq*: 64 Hz; *time*: $^{20}/_{64}$s
    - `.LInst.fAdderInst4`
    - *freq*: 64 Hz; *time*: $^{5}/_{64}$s
      - `.LInst.orCarryInst`
      - *freq*: 64 Hz; *time*: $^{1}/_{64}$s
        - `.MemFB.memInA`
          - *freq*: 64 Hz; *valid at*: $^{3}/_{64}$s F
          - *freq*: 64 Hz; *valid at*: $^{2}/_{64}$s F
          - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s F
          - *freq*: 64 Hz; *valid at*: 0s
        - `.MemFB.memInB`
          - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s F
          - *freq*: 64 Hz; *valid at*: 0s
        - `.MemBF.orOut`
          - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s F

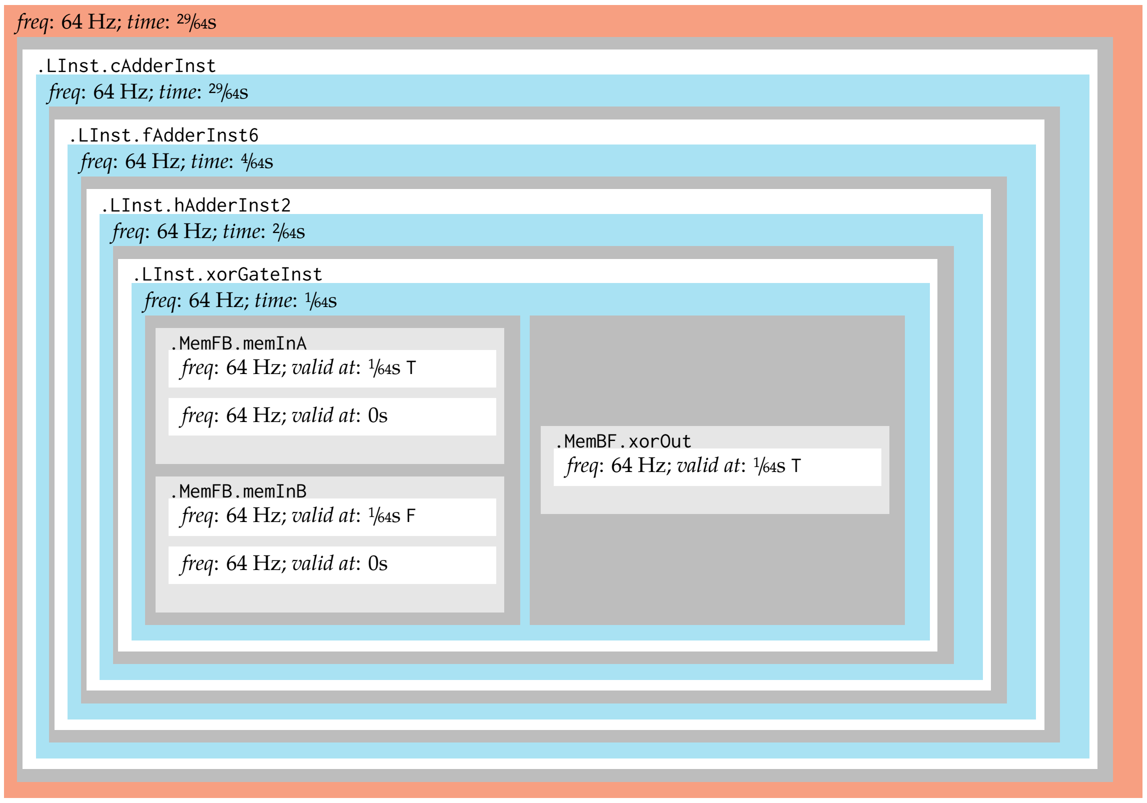
Listing F.7(xvii): The cascade adder


freq: 64 Hz; *time*: 24⁄64s
.LInst.cAdderInst
freq: 64 Hz; *time*: 24⁄64s
.LInst.fAdderInst5
freq: 64 Hz; *time*: 4⁄64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2⁄64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1⁄64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1⁄64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1⁄64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1⁄64s T

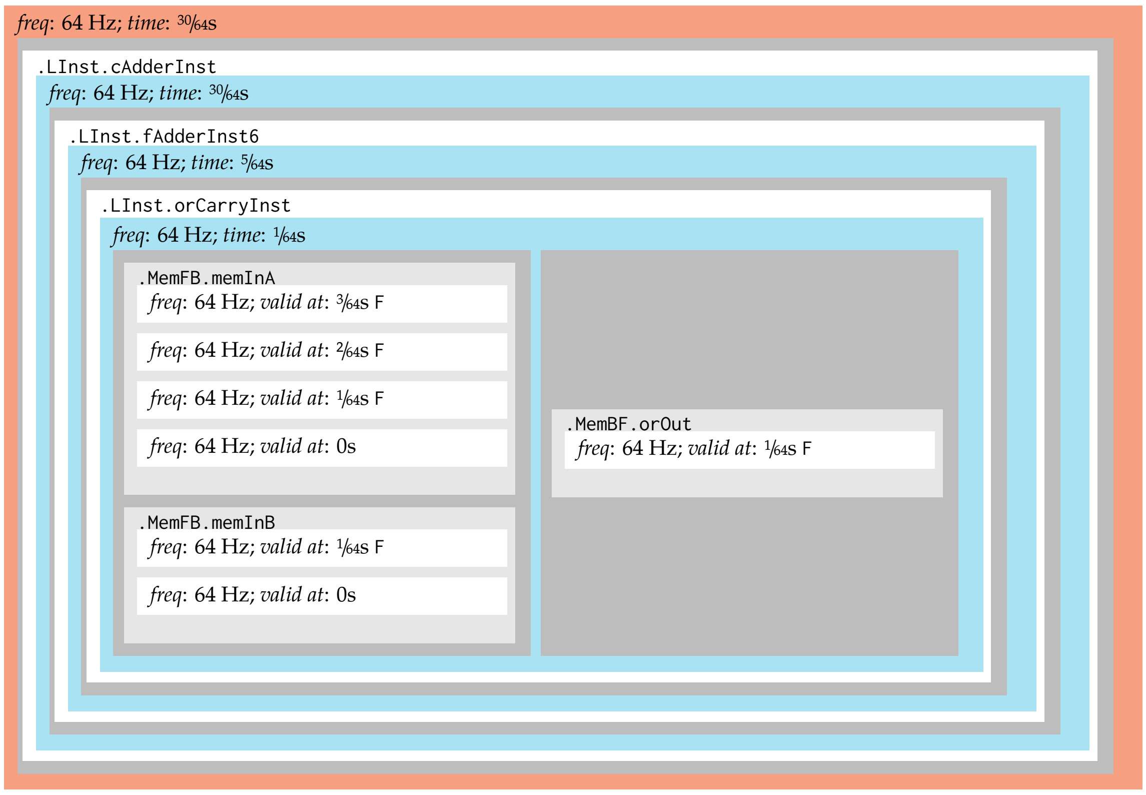
Listing F.7(xviii): The cascade adder


freq: 64 Hz; *time*: $^{25}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{25}/_{64}$s
.LInst.fAdderInst5
freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{3}/_{64}$s F
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F

Listing F.7(xix): The cascade adder

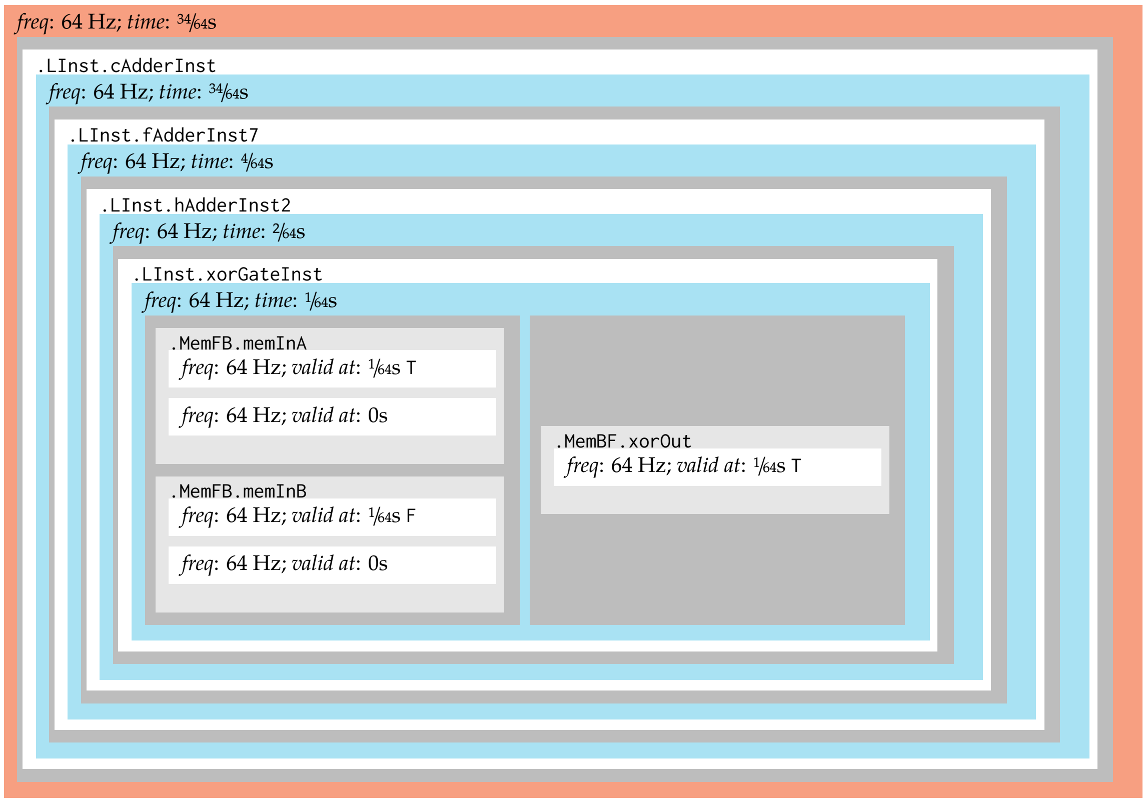

freq: 64 Hz; *time*: $^{29}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{29}/_{64}$s
.LInst.fAdderInst6
freq: 64 Hz; *time*: $^{4}/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.xorGateInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

Listing F.7(xx): The cascade adder

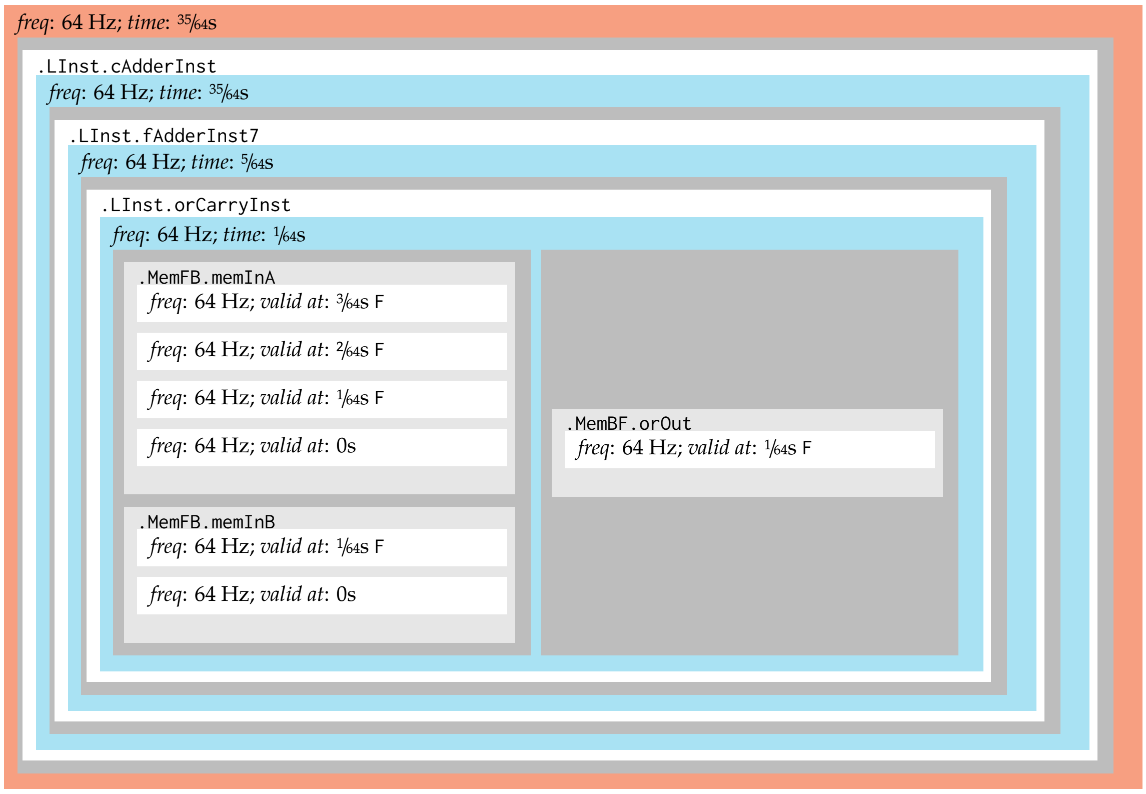


freq: 64 Hz; *time*: $^{30}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{30}/_{64}$s
.LInst.fAdderInst6
freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{3}/_{64}$s F
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F

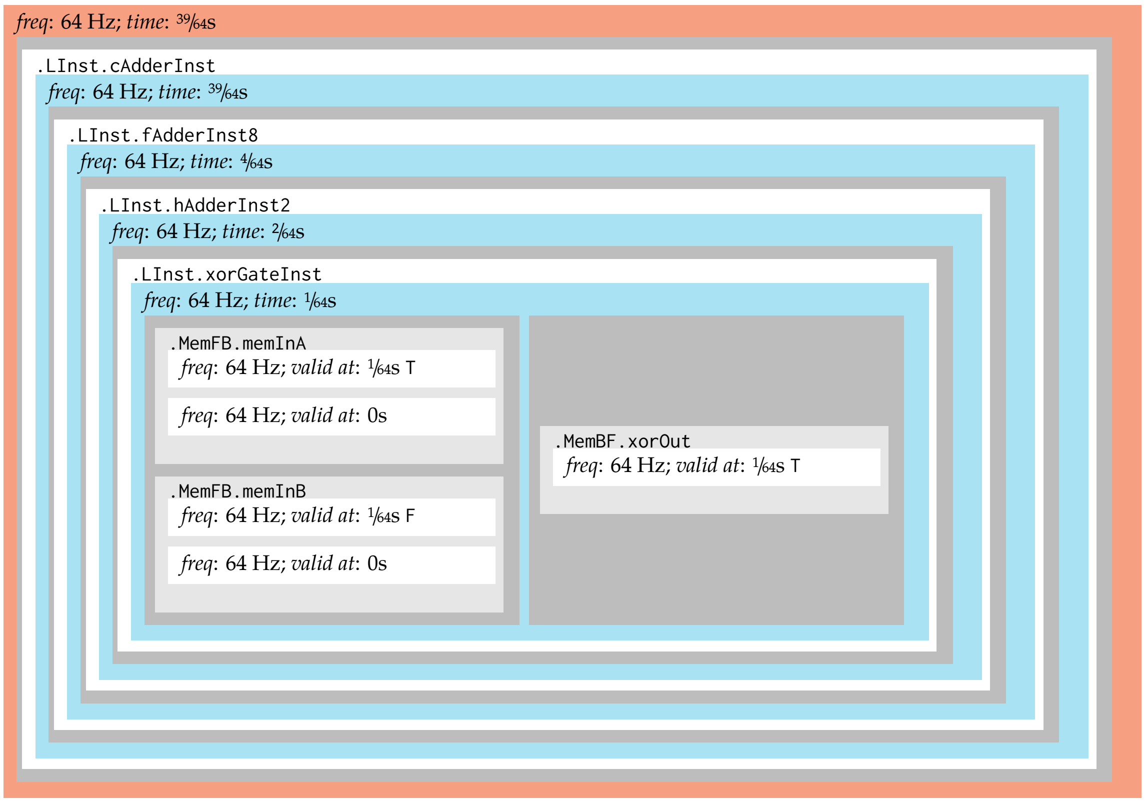
Listing F.7(xxi): The cascade adder


freq: 64 Hz; *time*: 34/64s
.LInst.cAdderInst
freq: 64 Hz; *time*: 34/64s
.LInst.fAdderInst7
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s T

Listing F.7(xxii): The cascade adder

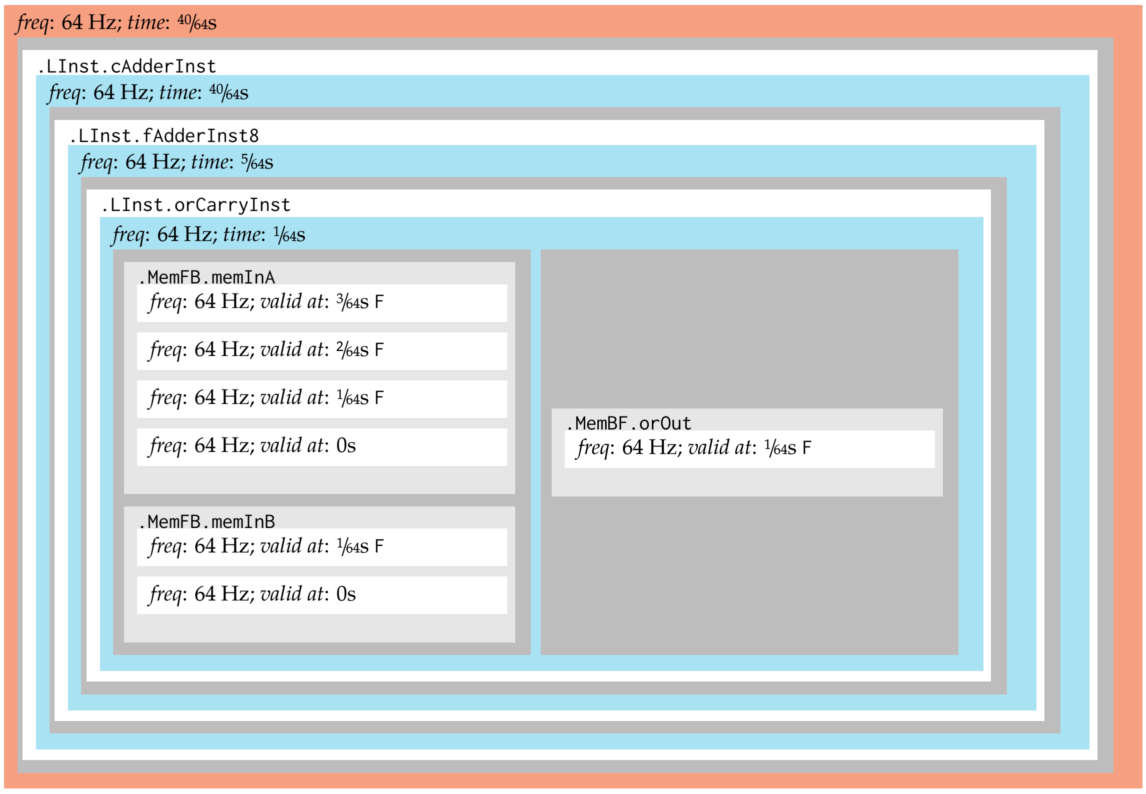

freq: 64 Hz; *time*: $^{35}/_{64}$s
.LInst.cAdderInst
freq: 64 Hz; *time*: $^{35}/_{64}$s
.LInst.fAdderInst7
freq: 64 Hz; *time*: $^{5}/_{64}$s
.LInst.orCarryInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{3}/_{64}$s F
freq: 64 Hz; *valid at*: $^{2}/_{64}$s F
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F

Listing F.7(xxiii): The cascade adder



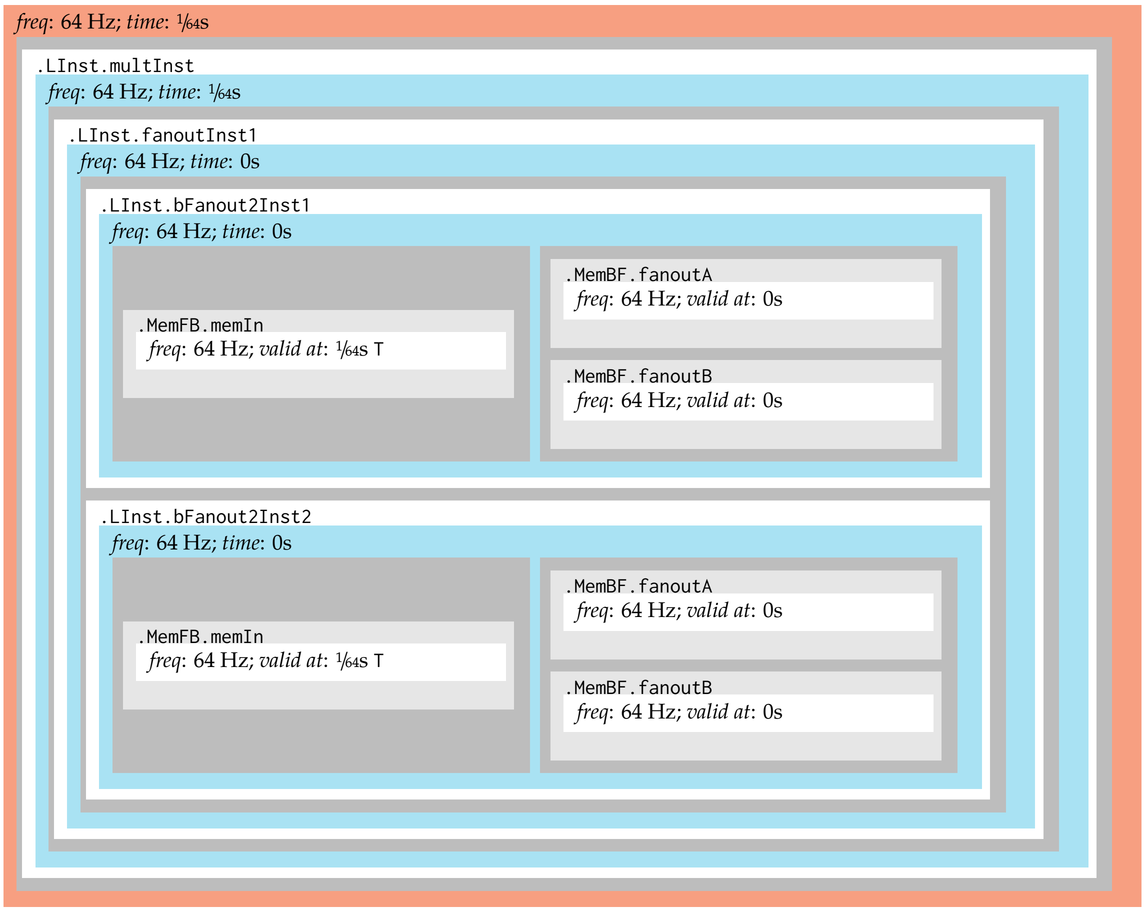
```
freq: 64 Hz; time: 39/64s
  .LInst.cAdderInst
  freq: 64 Hz; time: 39/64s
    .LInst.fAdderInst8
    freq: 64 Hz; time: 4/64s
      .LInst.hAdderInst2
      freq: 64 Hz; time: 2/64s
        .LInst.xorGateInst
        freq: 64 Hz; time: 1/64s
          .MemFB.memInA
            freq: 64 Hz; valid at: 1/64s T
            freq: 64 Hz; valid at: 0s
          .MemFB.memInB
            freq: 64 Hz; valid at: 1/64s F
            freq: 64 Hz; valid at: 0s
          .MemBF.xorOut
            freq: 64 Hz; valid at: 1/64s T
```

Listing F.7(xxiv): The cascade adder


freq: 64 Hz; *time*: 40/64s
.LInst.cAdderInst
freq: 64 Hz; *time*: 40/64s
.LInst.fAdderInst8
freq: 64 Hz; *time*: 5/64s
.LInst.orCarryInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 3/64s F
freq: 64 Hz; *valid at*: 2/64s F
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.orOut
freq: 64 Hz; *valid at*: 1/64s F

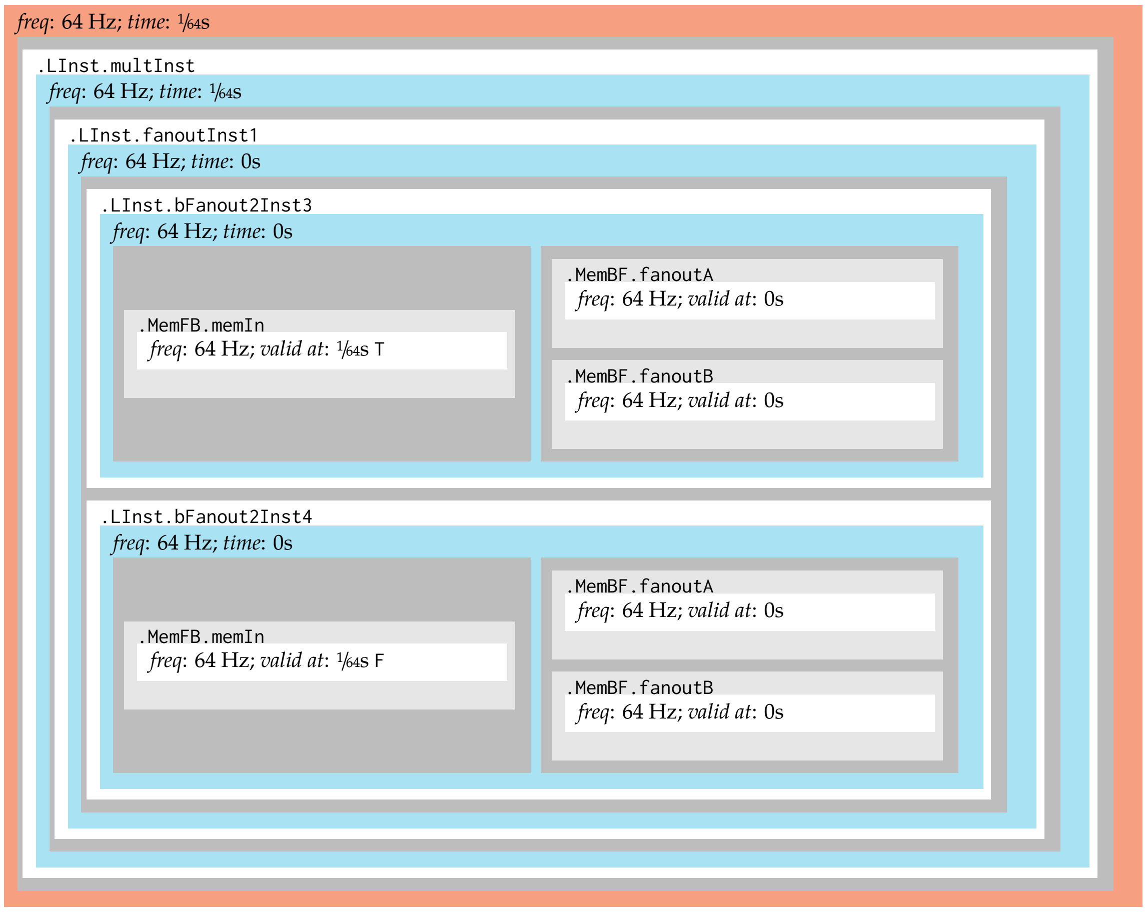
F.8 Multiplier

Listing F.8(i): The 4-bit multiplier

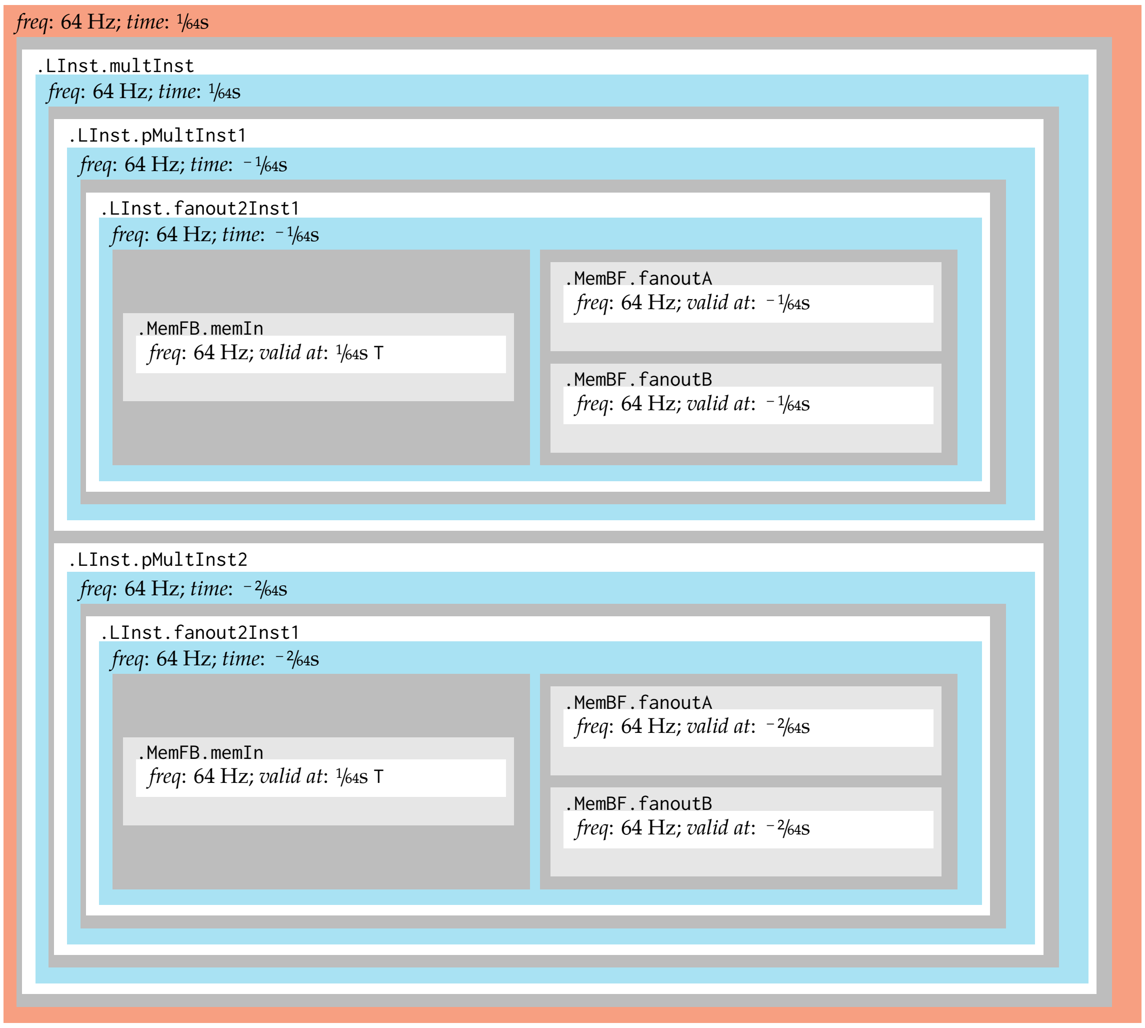


freq: 64 Hz; *time*: 1/64s
.LInst.multInst
freq: 64 Hz; *time*: 1/64s
.LInst.fanoutInst1
freq: 64 Hz; *time*: 0s
.LInst.bFanout2Inst1
freq: 64 Hz; *time*: 0s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 0s
.LInst.bFanout2Inst2
freq: 64 Hz; *time*: 0s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 0s

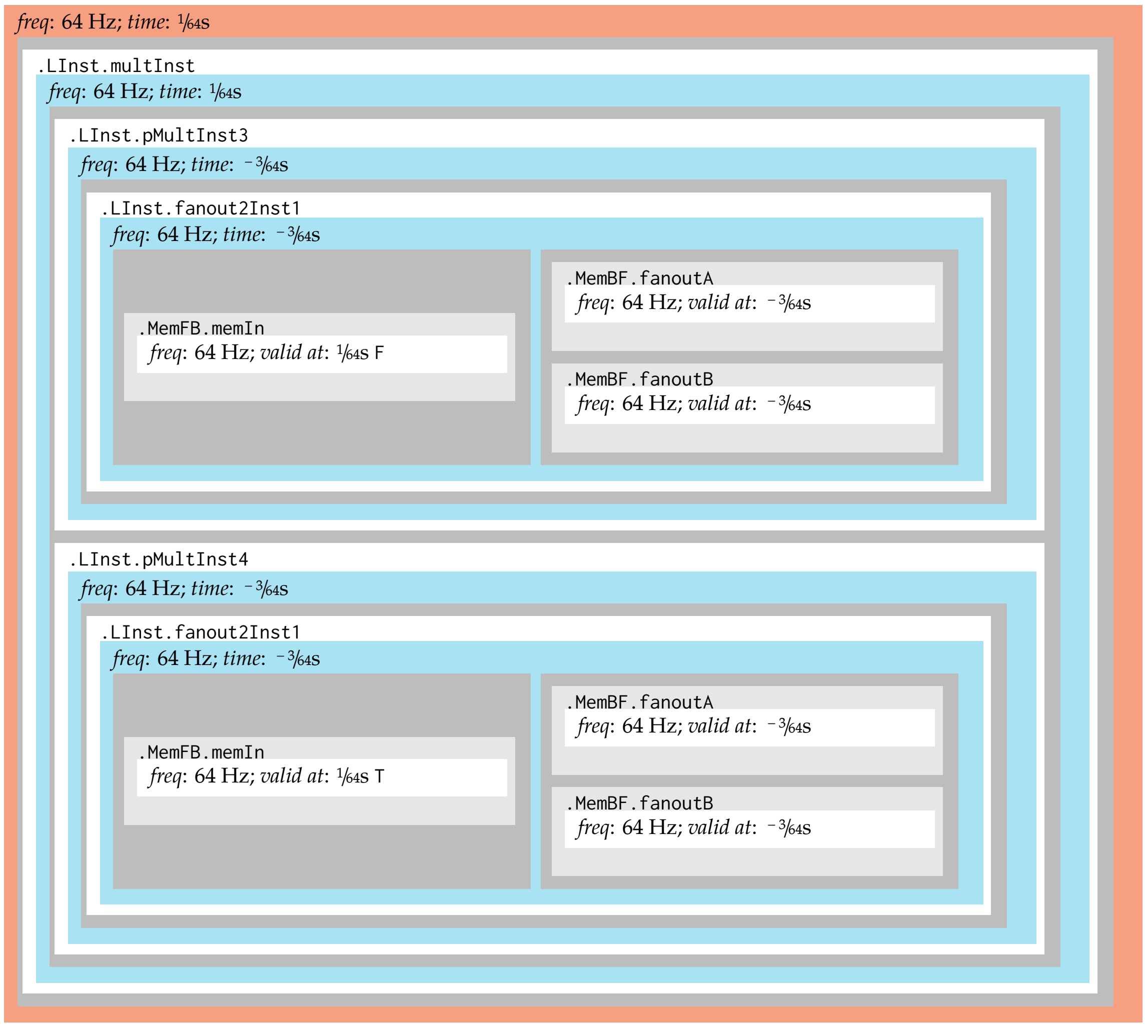
Listing F.8(ii): The 4-bit multiplier


freq: 64 Hz; *time*: 1/64s
.LInst.multInst
freq: 64 Hz; *time*: 1/64s
.LInst.fanoutInst1
freq: 64 Hz; *time*: 0s
.LInst.bFanout2Inst3
freq: 64 Hz; *time*: 0s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 0s
.LInst.bFanout2Inst4
freq: 64 Hz; *time*: 0s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 0s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 0s

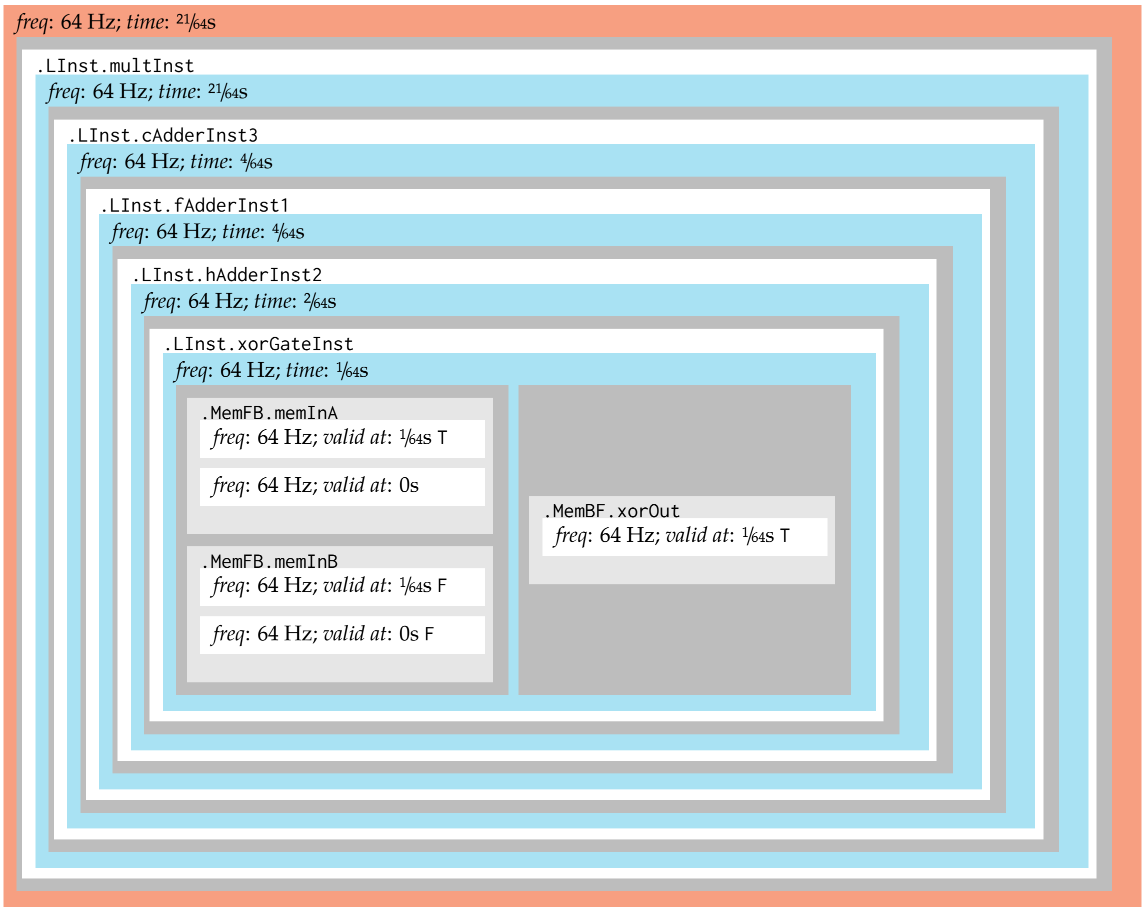
Listing F.8(iii): The 4-bit multiplier


freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.multInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.pMultInst1
freq: 64 Hz; *time*: $-\frac{1}{64}$s
.LInst.fanout2Inst1
freq: 64 Hz; *time*: $-\frac{1}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{1}{64}$s
.LInst.pMultInst2
freq: 64 Hz; *time*: $-\frac{2}{64}$s
.LInst.fanout2Inst1
freq: 64 Hz; *time*: $-\frac{2}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{2}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{2}{64}$s

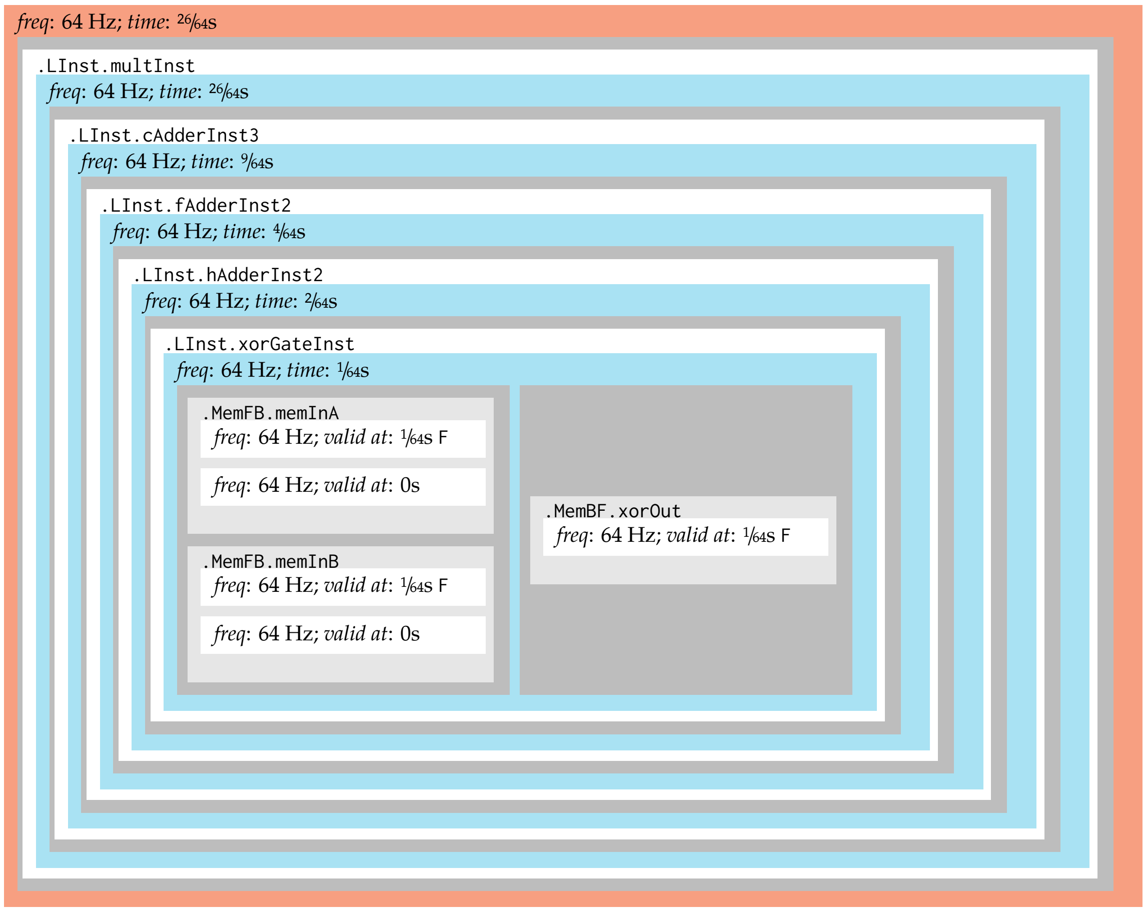
Listing F.8(iv): The 4-bit multiplier


freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.multInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.pMultInst3
freq: 64 Hz; *time*: $-\frac{3}{64}$s
.LInst.fanout2Inst1
freq: 64 Hz; *time*: $-\frac{3}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{3}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{3}{64}$s
.LInst.pMultInst4
freq: 64 Hz; *time*: $-\frac{3}{64}$s
.LInst.fanout2Inst1
freq: 64 Hz; *time*: $-\frac{3}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $-\frac{3}{64}$s
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $-\frac{3}{64}$s

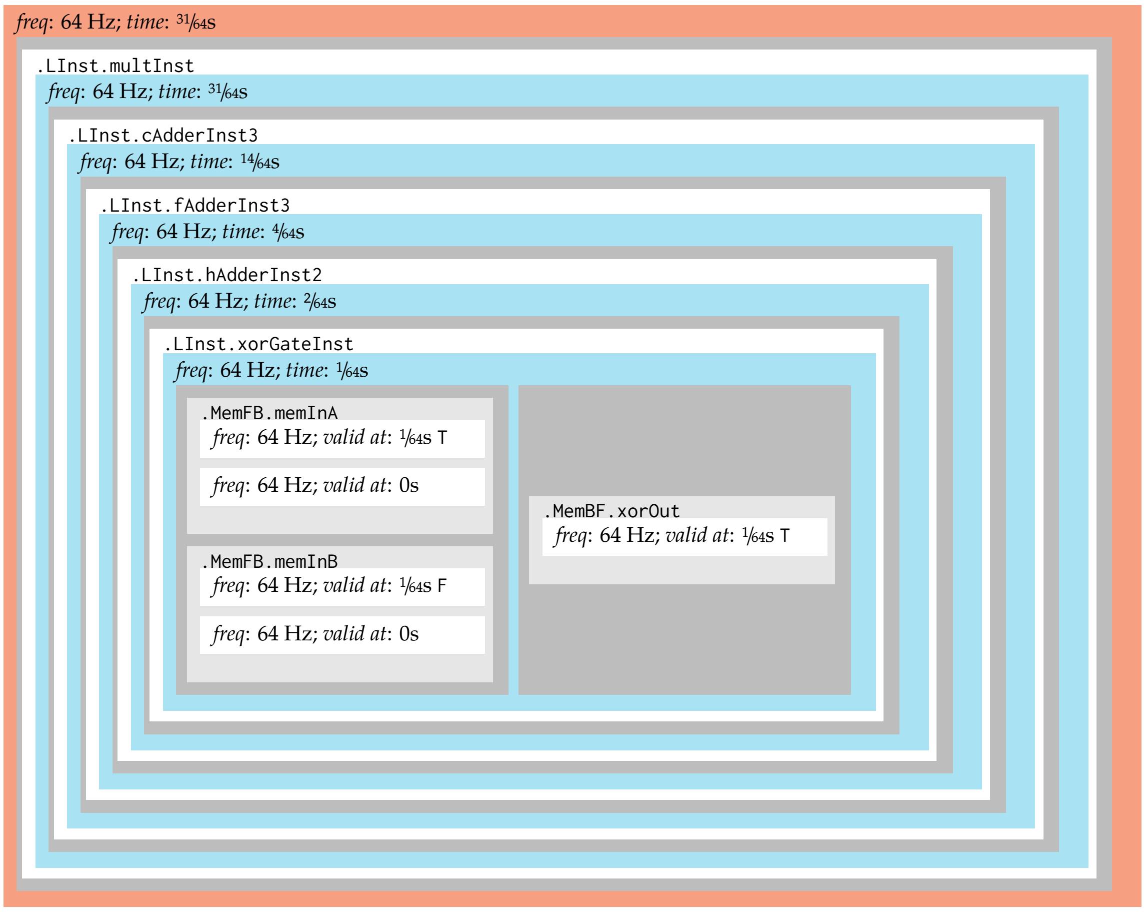
Listing F.8(v): The 4-bit multiplier


freq: 64 Hz; *time*: $^{21}/_{64}$s
.LInst.multInst
freq: 64 Hz; *time*: $^{21}/_{64}$s
.LInst.cAdderInst3
freq: 64 Hz; *time*: $^{4}/_{64}$s
.LInst.fAdderInst1
freq: 64 Hz; *time*: $^{4}/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.xorGateInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s F
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

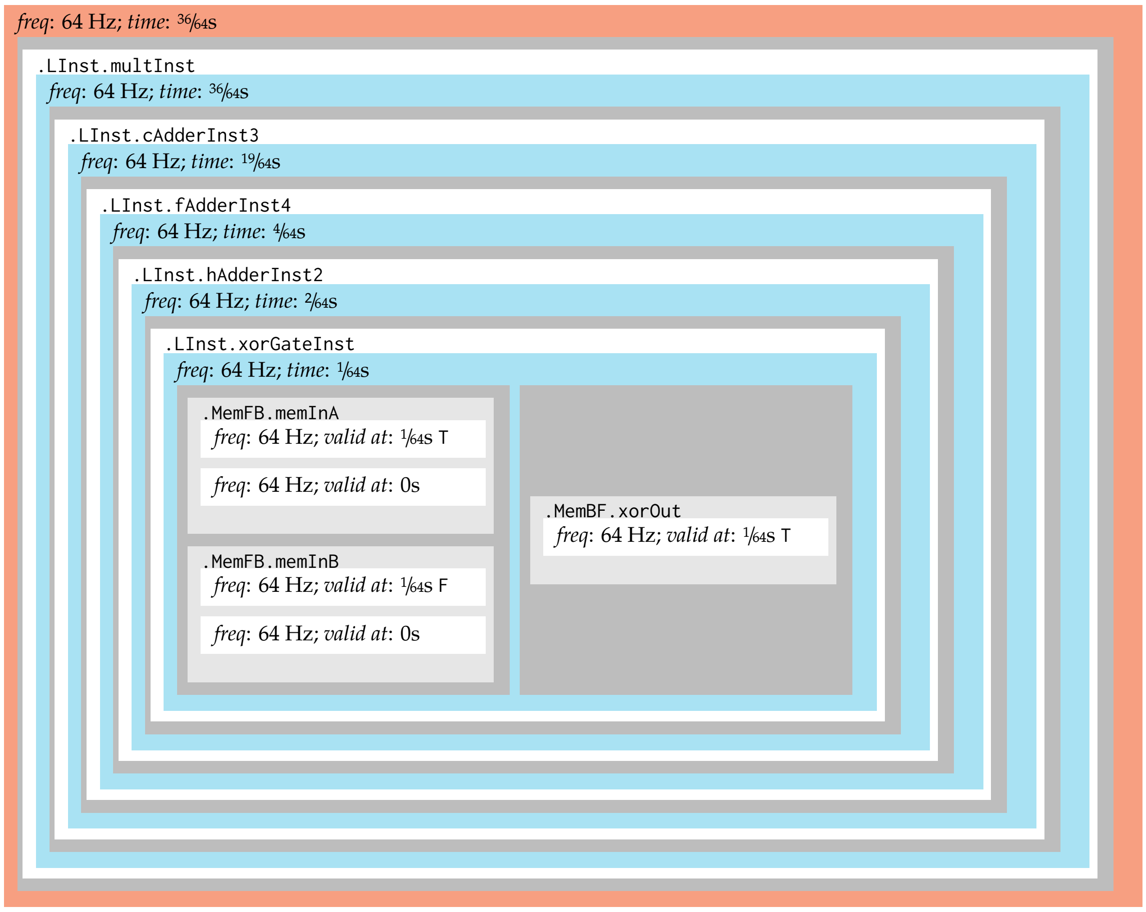
Listing F.8(vi): The 4-bit multiplier


freq: 64 Hz; *time*: 26/64s
.LInst.multInst
freq: 64 Hz; *time*: 26/64s
.LInst.cAdderInst3
freq: 64 Hz; *time*: 9/64s
.LInst.fAdderInst2
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s F

Listing F.8(vii): The 4-bit multiplier

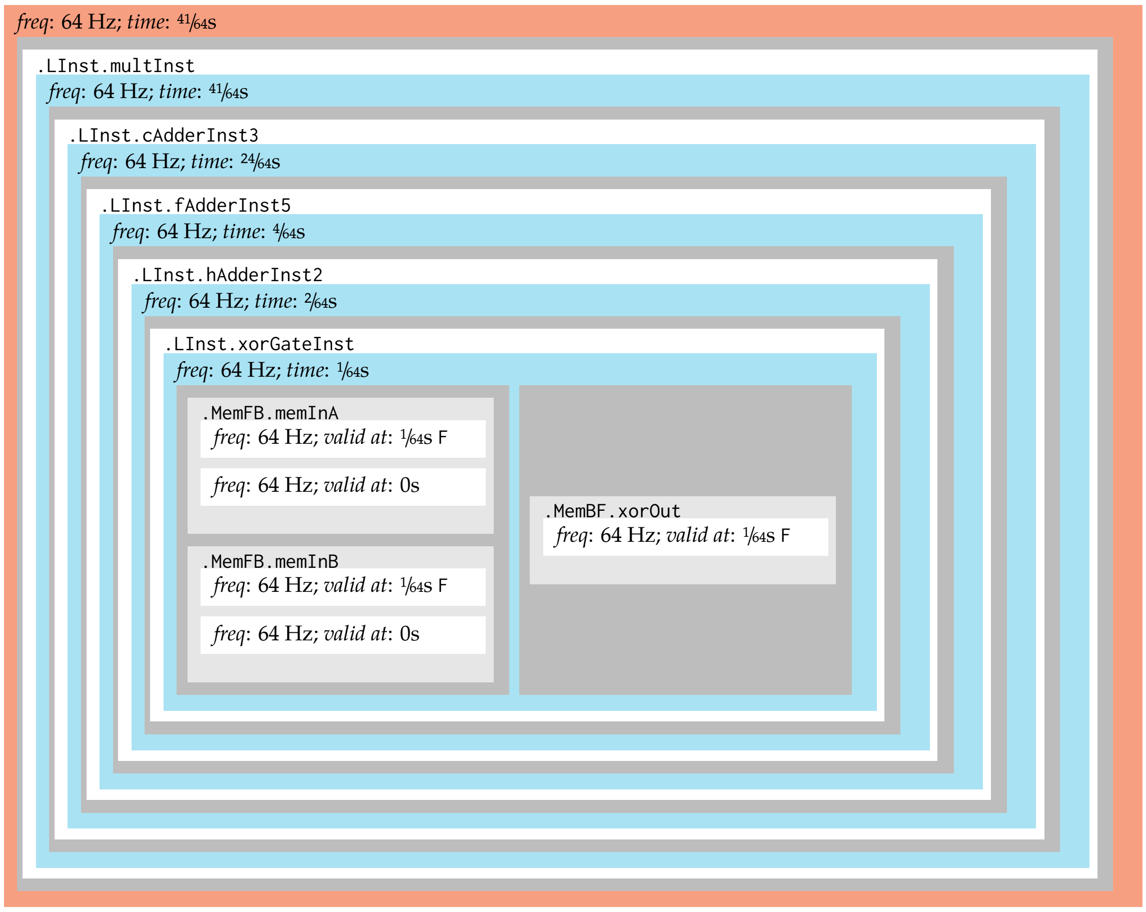

freq: 64 Hz; *time*: 31/64s
.LInst.multInst
freq: 64 Hz; *time*: 31/64s
.LInst.cAdderInst3
freq: 64 Hz; *time*: 14/64s
.LInst.fAdderInst3
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s T

Listing F.8(viii): The 4-bit multiplier

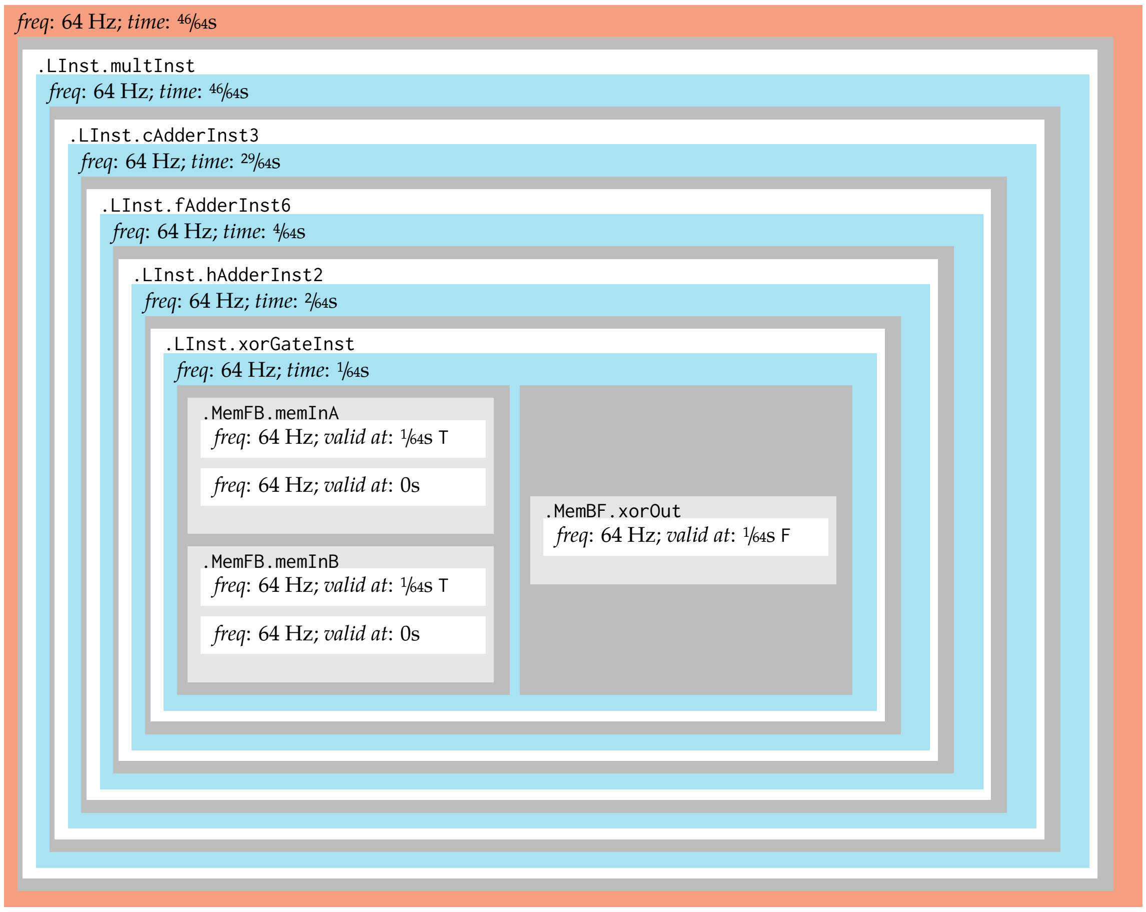


freq: 64 Hz; *time*: 36⁄64s
.LInst.multInst
freq: 64 Hz; *time*: 36⁄64s
.LInst.cAdderInst3
freq: 64 Hz; *time*: 19⁄64s
.LInst.fAdderInst4
freq: 64 Hz; *time*: 4⁄64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2⁄64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1⁄64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1⁄64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1⁄64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1⁄64s T

Listing F.8(ix): The 4-bit multiplier

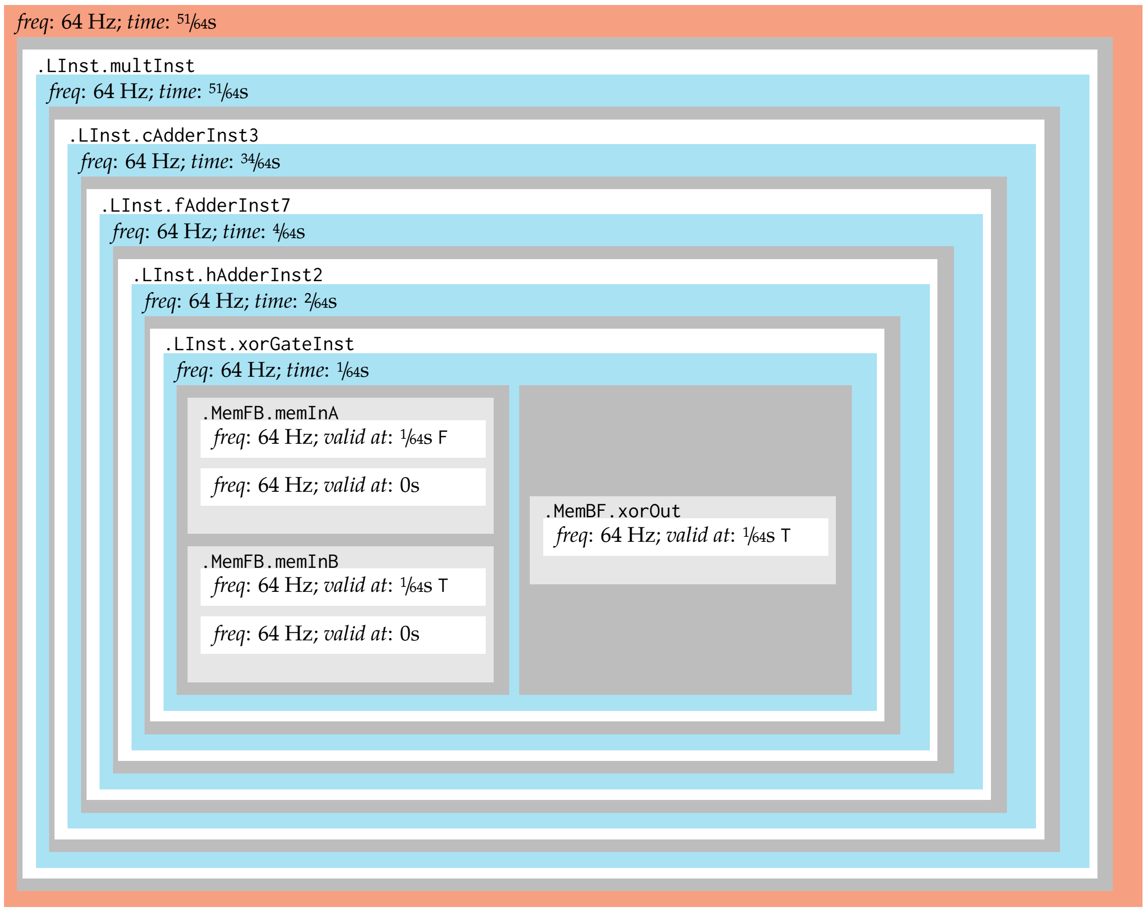

freq: 64 Hz; *time*: 41⁄64s
.LInst.multInst
freq: 64 Hz; *time*: 41⁄64s
.LInst.cAdderInst3
freq: 64 Hz; *time*: 24⁄64s
.LInst.fAdderInst5
freq: 64 Hz; *time*: 4⁄64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2⁄64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1⁄64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1⁄64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1⁄64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1⁄64s F

Listing F.8(x): The 4-bit multiplier

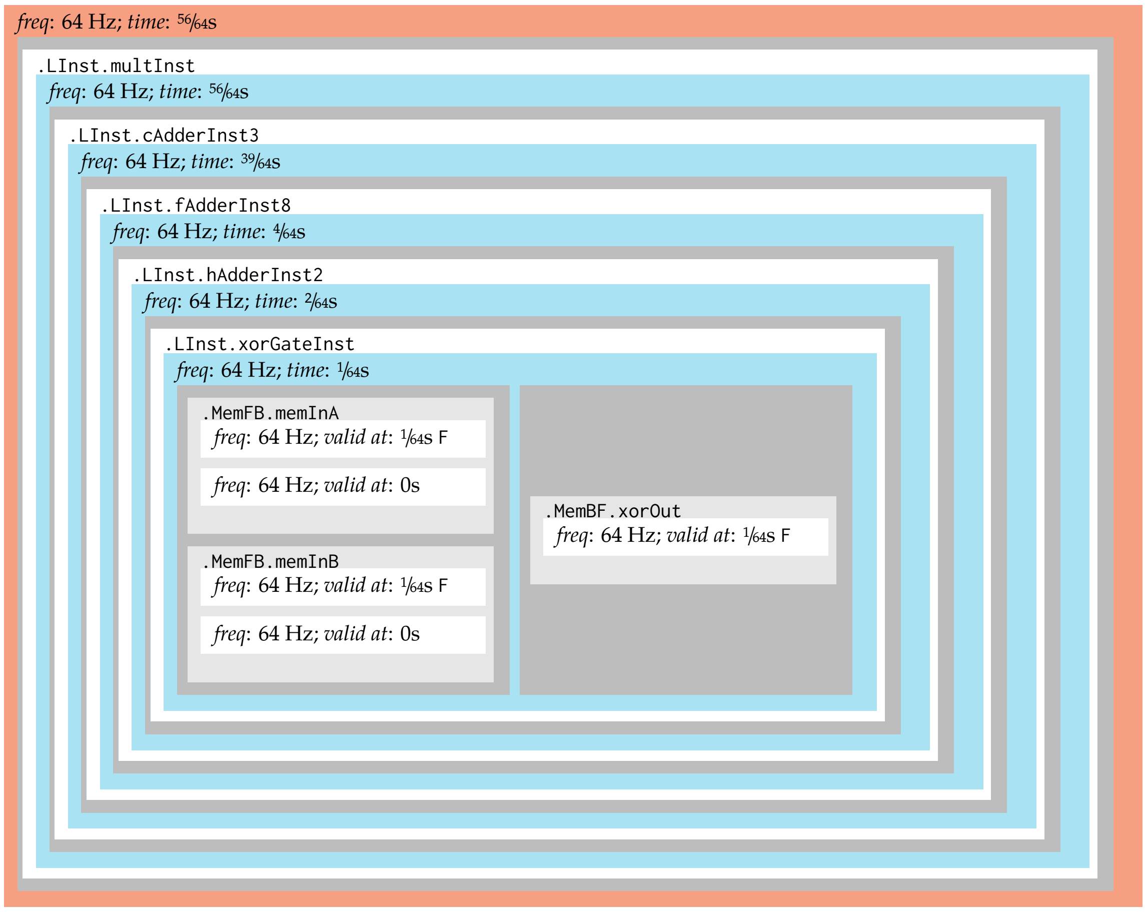


freq: 64 Hz; *time*: 46/64s
.LInst.multInst
freq: 64 Hz; *time*: 46/64s
.LInst.cAdderInst3
freq: 64 Hz; *time*: 29/64s
.LInst.fAdderInst6
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s T
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s F

Listing F.8(xi): The 4-bit multiplier

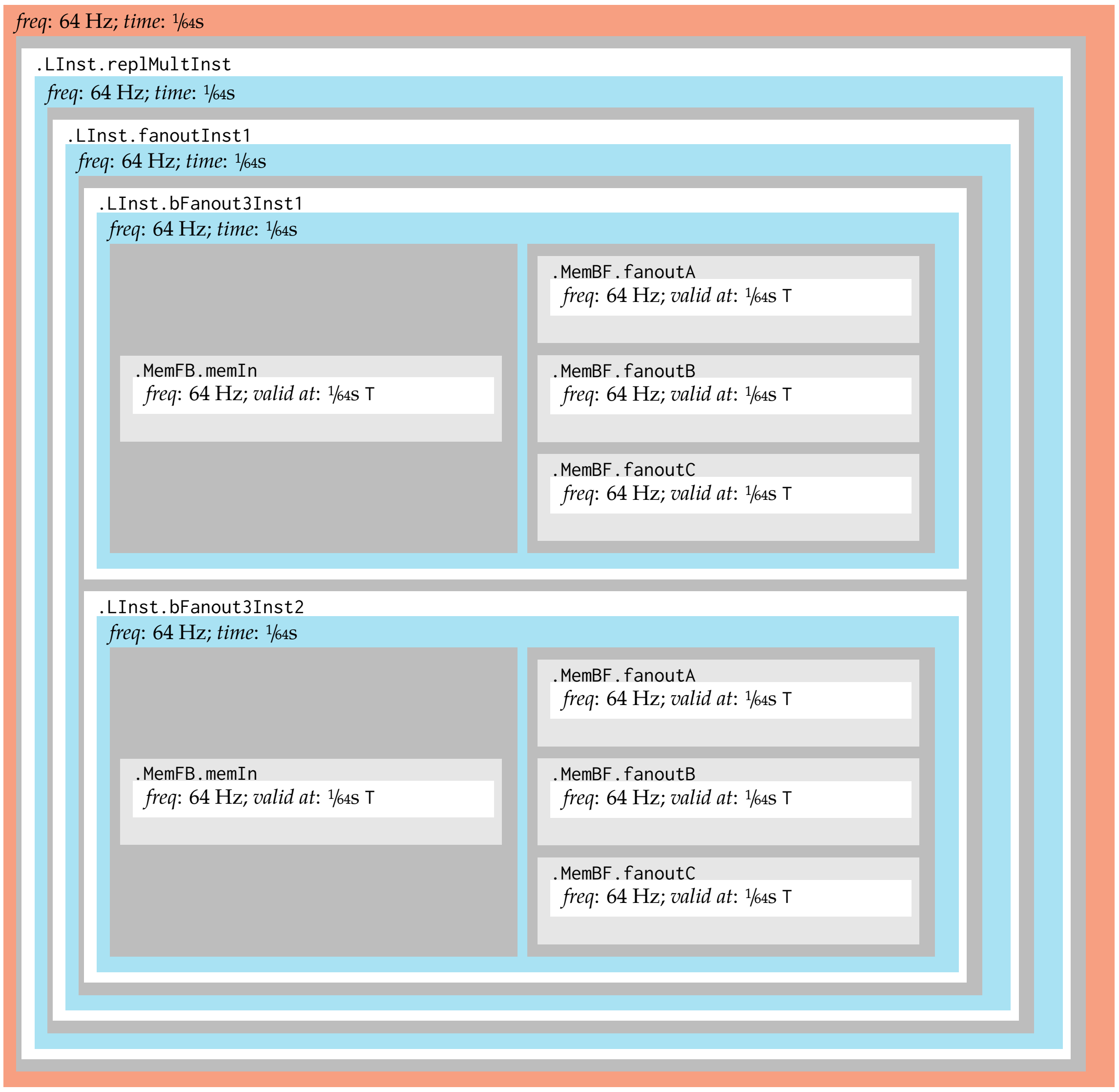

freq: 64 Hz; *time*: $^{51}/_{64}$s
.LInst.multInst
freq: 64 Hz; *time*: $^{51}/_{64}$s
.LInst.cAdderInst3
freq: 64 Hz; *time*: $^{34}/_{64}$s
.LInst.fAdderInst7
freq: 64 Hz; *time*: $^{4}/_{64}$s
.LInst.hAdderInst2
freq: 64 Hz; *time*: $^{2}/_{64}$s
.LInst.xorGateInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

Listing F.8(xii): The 4-bit multiplier


freq: 64 Hz; *time*: 56/64s
.LInst.multInst
freq: 64 Hz; *time*: 56/64s
.LInst.cAdderInst3
freq: 64 Hz; *time*: 39/64s
.LInst.fAdderInst8
freq: 64 Hz; *time*: 4/64s
.LInst.hAdderInst2
freq: 64 Hz; *time*: 2/64s
.LInst.xorGateInst
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.xorOut
freq: 64 Hz; *valid at*: 1/64s F

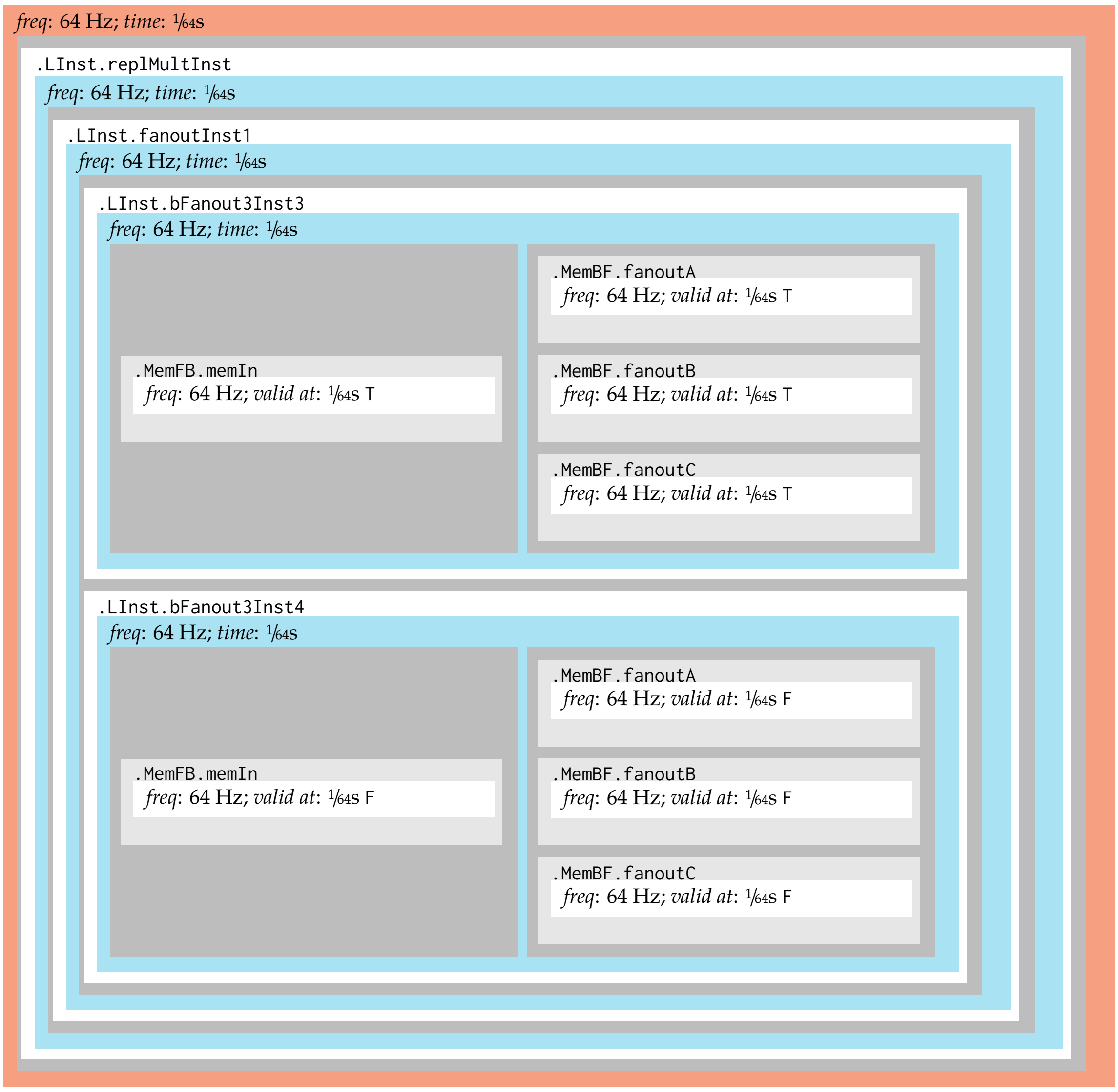
F.9 Replicated multiplier

Listing F.9(i): The replicated 4-bit multiplier



freq: 64 Hz; *time*: 1⁄64s
.LInst.replMultInst
freq: 64 Hz; *time*: 1⁄64s
.LInst.fanoutInst1
freq: 64 Hz; *time*: 1⁄64s
.LInst.bFanout3Inst1
freq: 64 Hz; *time*: 1⁄64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1⁄64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1⁄64s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1⁄64s T
.MemBF.fanoutC
freq: 64 Hz; *valid at*: 1⁄64s T
.LInst.bFanout3Inst2
freq: 64 Hz; *time*: 1⁄64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1⁄64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1⁄64s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1⁄64s T
.MemBF.fanoutC
freq: 64 Hz; *valid at*: 1⁄64s T

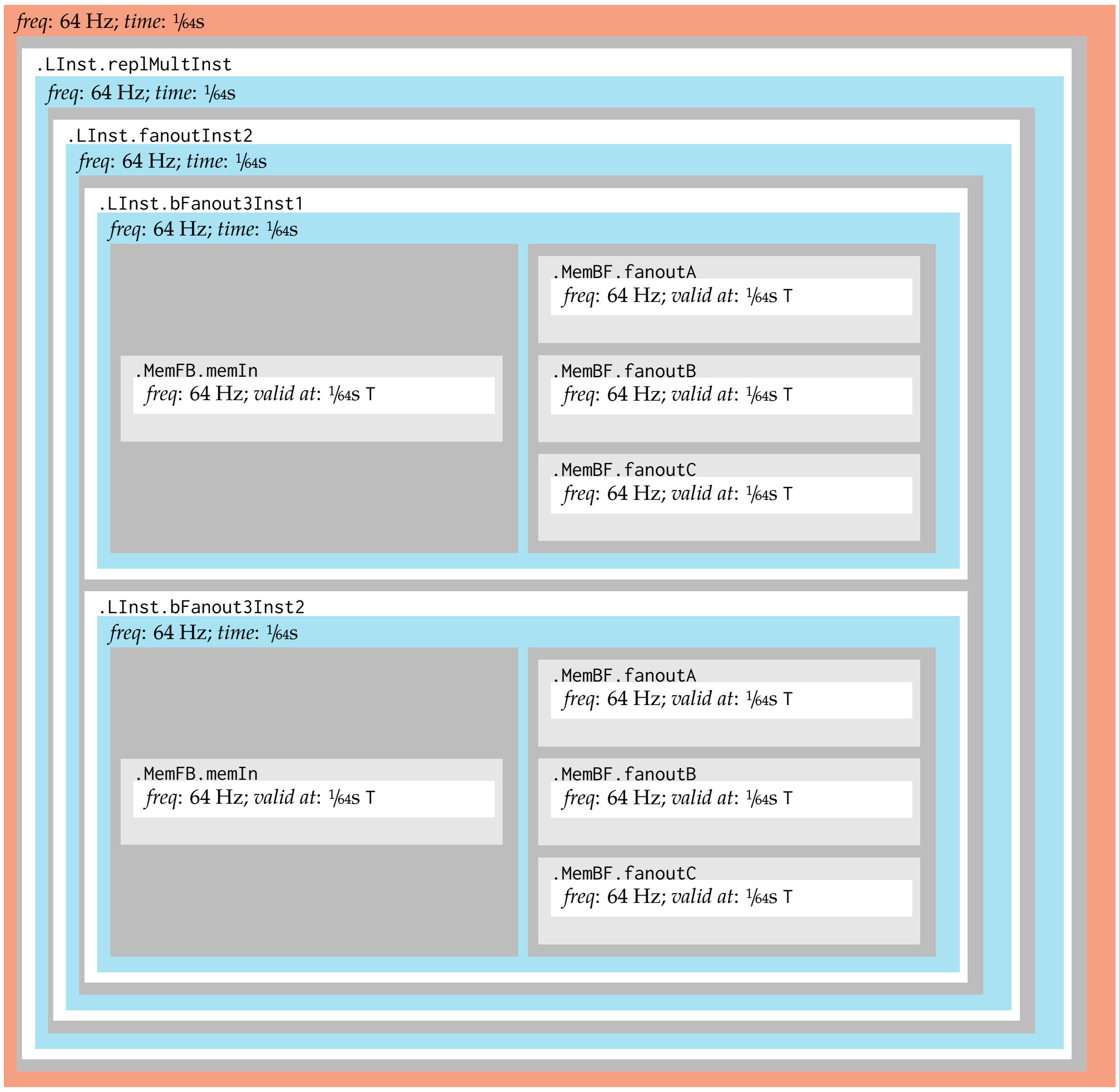
Listing F.9(ii): The replicated 4-bit multiplier



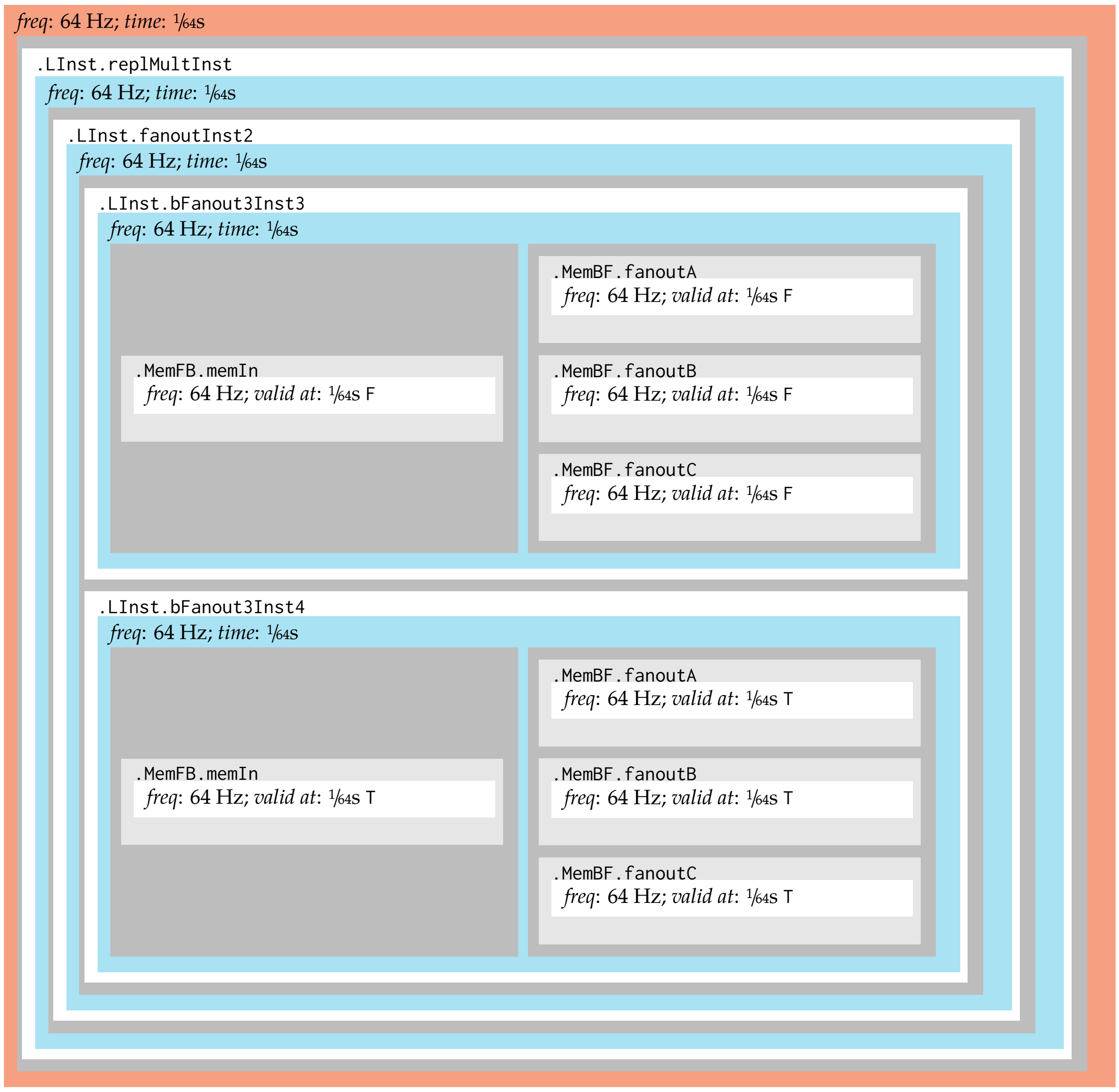
freq: 64 Hz; *time*: $\frac{1}{64}$s

- .LInst.replMultInst
  - *freq*: 64 Hz; *time*: $\frac{1}{64}$s
    - .LInst.fanoutInst1
      - *freq*: 64 Hz; *time*: $\frac{1}{64}$s
        - .LInst.bFanout3Inst3
          - *freq*: 64 Hz; *time*: $\frac{1}{64}$s
            - .MemFB.memIn
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
            - .MemBF.fanoutA
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
            - .MemBF.fanoutB
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
            - .MemBF.fanoutC
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s T
        - .LInst.bFanout3Inst4
          - *freq*: 64 Hz; *time*: $\frac{1}{64}$s
            - .MemFB.memIn
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s F
            - .MemBF.fanoutA
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s F
            - .MemBF.fanoutB
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s F
            - .MemBF.fanoutC
              - *freq*: 64 Hz; *valid at*: $\frac{1}{64}$s F

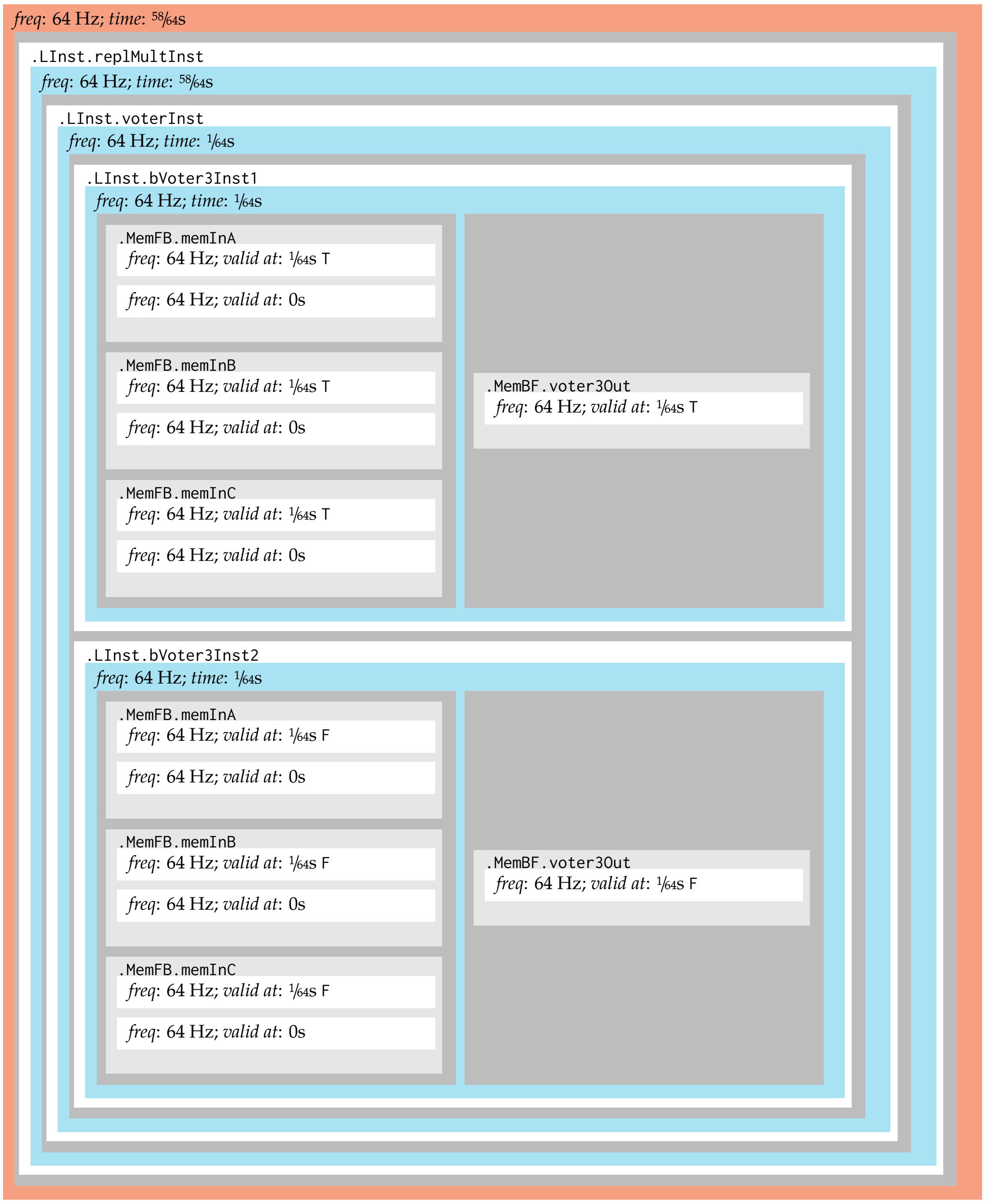
Listing F.9(iii): The replicated 4-bit multiplier


freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.replMultInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.fanoutInst2
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.bFanout3Inst1
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutC
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.LInst.bFanout3Inst2
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.memIn
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.MemBF.fanoutC
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T

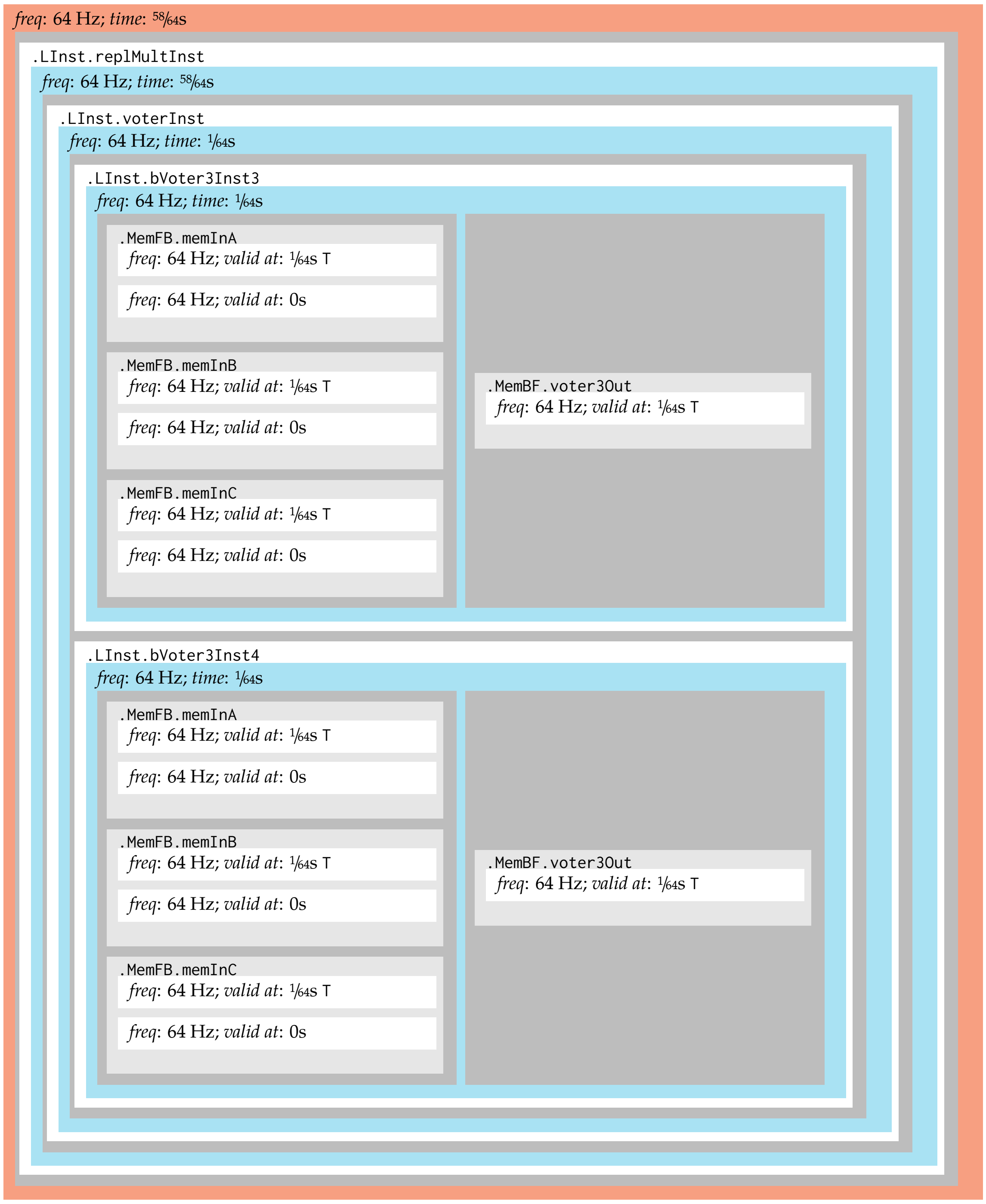
Listing F.9(iv): The replicated 4-bit multiplier


freq: 64 Hz; *time*: 1/64s
.LInst.replMultInst
freq: 64 Hz; *time*: 1/64s
.LInst.fanoutInst2
freq: 64 Hz; *time*: 1/64s
.LInst.bFanout3Inst3
freq: 64 Hz; *time*: 1/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s F
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1/64s F
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1/64s F
.MemBF.fanoutC
freq: 64 Hz; *valid at*: 1/64s F
.LInst.bFanout3Inst4
freq: 64 Hz; *time*: 1/64s
.MemFB.memIn
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutA
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutB
freq: 64 Hz; *valid at*: 1/64s T
.MemBF.fanoutC
freq: 64 Hz; *valid at*: 1/64s T

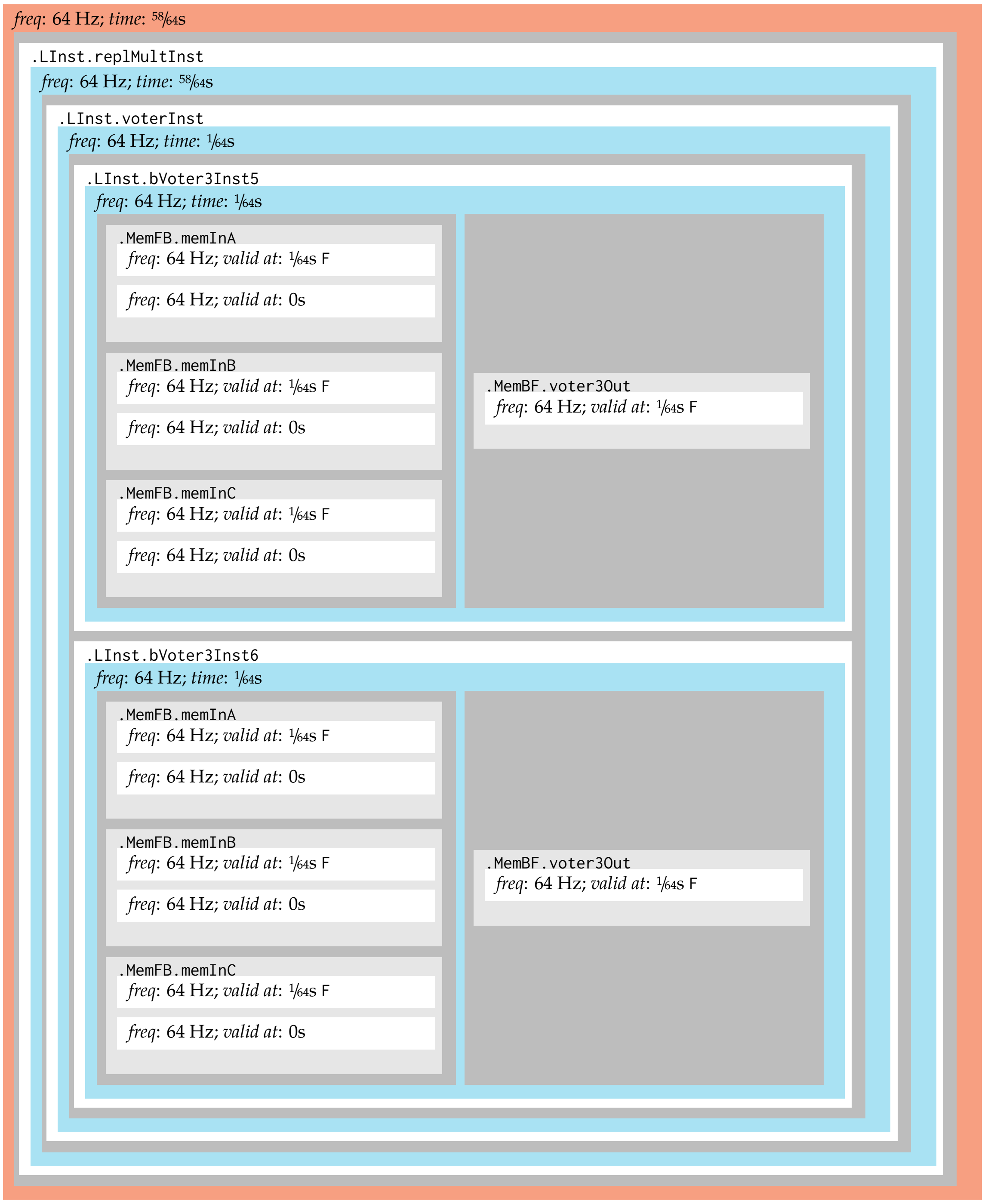
Listing F.9(v): The replicated 4-bit multiplier


freq: 64 Hz; *time*: $\frac{58}{64}$s
.LInst.replMultInst
freq: 64 Hz; *time*: $\frac{58}{64}$s
.LInst.voterInst
freq: 64 Hz; *time*: $\frac{1}{64}$s
.LInst.bVoter3Inst1
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInC
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemBF.voter3Out
freq: 64 Hz; *valid at*: $\frac{1}{64}$s T
.LInst.bVoter3Inst2
freq: 64 Hz; *time*: $\frac{1}{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInC
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F
freq: 64 Hz; *valid at*: 0s
.MemBF.voter3Out
freq: 64 Hz; *valid at*: $\frac{1}{64}$s F

Listing F.9(vi): The replicated 4-bit multiplier


freq: 64 Hz; *time*: $^{58}/_{64}$s
.LInst.replMultInst
freq: 64 Hz; *time*: $^{58}/_{64}$s
.LInst.voterInst
freq: 64 Hz; *time*: $^{1}/_{64}$s
.LInst.bVoter3Inst3
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInC
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemBF.voter3Out
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
.LInst.bVoter3Inst4
freq: 64 Hz; *time*: $^{1}/_{64}$s
.MemFB.memInA
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemFB.memInC
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T
freq: 64 Hz; *valid at*: 0s
.MemBF.voter3Out
freq: 64 Hz; *valid at*: $^{1}/_{64}$s T

Listing F.9(vii): The replicated 4-bit multiplier



freq: 64 Hz; *time*: 58/64s
.LInst.replMultInst
freq: 64 Hz; *time*: 58/64s
.LInst.voterInst
freq: 64 Hz; *time*: 1/64s
.LInst.bVoter3Inst5
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInC
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.voter3Out
freq: 64 Hz; *valid at*: 1/64s F
.LInst.bVoter3Inst6
freq: 64 Hz; *time*: 1/64s
.MemFB.memInA
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInB
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemFB.memInC
freq: 64 Hz; *valid at*: 1/64s F
freq: 64 Hz; *valid at*: 0s
.MemBF.voter3Out
freq: 64 Hz; *valid at*: 1/64s F

Listing F.9(viii): The replicated 4-bit multiplier

- *freq*: 64 Hz; *time*: $^{58}/_{64}$s
  - `.LInst.replMultInst`
    - *freq*: 64 Hz; *time*: $^{58}/_{64}$s
      - `.LInst.voterInst`
        - *freq*: 64 Hz; *time*: $^{1}/_{64}$s
          - `.LInst.bVoter3Inst7`
            - *freq*: 64 Hz; *time*: $^{1}/_{64}$s
              - `.MemFB.memInA`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: 0s
              - `.MemFB.memInB`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: 0s
              - `.MemFB.memInC`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `T`
                - *freq*: 64 Hz; *valid at*: 0s
              - `.MemBF.voter3Out`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `T`
          - `.LInst.bVoter3Inst8`
            - *freq*: 64 Hz; *time*: $^{1}/_{64}$s
              - `.MemFB.memInA`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `F`
                - *freq*: 64 Hz; *valid at*: 0s
              - `.MemFB.memInB`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `F`
                - *freq*: 64 Hz; *valid at*: 0s
              - `.MemFB.memInC`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `F`
                - *freq*: 64 Hz; *valid at*: 0s
              - `.MemBF.voter3Out`
                - *freq*: 64 Hz; *valid at*: $^{1}/_{64}$s `F`