The Hume Report, Version 0.2

Greg Michaelson[1] Kevin Hammond[2]

[1]Department of Computing and Electrical Engineering
Heriot-Watt University
greg@cee.hw.ac.uk, +44 131 451 3422

[2]School of Computer Science, University of St Andrews
kh@dcs.st-and.ac.uk, +44 1334 463241

Contents

David Hume, Scottish Sceptical Philosopher: 1711-1776

Chapter 1

Introduction

This document describes the Hume programming language. *Hume* (Higher-order Unified Meta-Environment) is a strongly typed, functionally-based language with an integrated tool set for developing, proving and assessing concurrent, resource-limited systems, such as embedded or safety-critical systems. It aims to extend the frontiers of language design for such systems, introducing new levels of abstraction and provability.

Hume is named for the Scottish Enlightenment sceptical philosopher David Hume (1711-1776), who counseled that:

> *To begin with clear and self-evident principles, to advance by timorous and sure steps, to review frequently our conclusions, and examine accurately all their consequences; though by these means we shall make both a slow and a short progress in our systems; are the only methods, by which we can ever hope to reach truth, and attain a proper stability and certainty in our determinations.*
> D. Hume, An Enquiry Concerning Human Understanding, 1748

These sentiments epitomise the philosophy of programming language design that has been followed in this document.

This report is structured as follows: the remainder of this chapter provides motivation and general background; Chapter 2 is an overview of the Hume language design, including detailed informal descriptions of the process and coordination sub-languages; future chapters will cover implementation and cost modelling. Appendix A describes the concrete syntax; Appendix B is the formal static semantics, including the type system and Appendix C gives the formal dynamic semantics. Finally Appendix D defines the Hume standard prelude.

1.1 Motivation and objectives

Since the focus of the Hume design is on high reliability applications (such as safety critical or embedded systems), it is paramount that Hume programs have predictable and, preferably, provable properties. However, the strong properties of program equivalence, termination and time and space use are undecidable for Turing computable languages. Conversely, languages in which such properties are decidable (i.e. finite state machines) lack expressiveness. The goal of the Hume language design is to support a high level of expressive power, whilst providing strong guarantees of dynamic behavioural properties such as execution time and space usage.

Program proof and manipulation are greatly eased by abstractness as well as by succinctness. In particular, it is relatively hard to construct formal theories for imperative language constructs, where time ordering greatly complicates reasoning about programs. However, programs are ultimately intended to realise solutions to concrete problems on physical computers. Increased abstractness in languages, in particular away from modifiable state, tends to greater distance from the von Neumann paradigm, with corresponding complications and efficiency losses in implementations. The Hume design combines the desirable properties of abstraction and succinctness that are provided by a good functional programming language with a coordination language that explicitly captures time and space behaviour. Runtime efficiency is maintained through careful language design with a view to straightforward implementation on conventional computer architectures or embedded systems.

Where formal theories can be constructed, their static application to non-trivial programs is characterised by poor scalability through exponential growth in the space of properties to be explored. Alternatively, accuracy is lost through simplifying assumptions and heuristics. Dynamic evaluation of programs through instrumentation and profiling suffers from similar limitations. Typically, the volume of test data and the time required for exhaustive empirical exploration of program behaviour is prohibitive, both growing rapidly with the fineness of granularity at which exploration is conducted. Contrariwise, accuracy is lost at coarser granularity or with non-exhaustive testing.

Hume reflects these considerations in:

- the separation of the *expression* and *coordination* aspects of the language;
- the provision of an integrated *tool set*, spanning both static and dynamic program analysis and manipulation.

1.1.1 Important Design Characteristics

In general, high reliability systems must meet both strong correctness criteria and strict performance criteria. The latter are most easily attained by working at a low level, whereas the former are most easily attained by working at a high level. A primary objective of the Hume design is to allow both types of criteria to be met while working at a high level of abstraction.

The language has been designed to allow relatively simple formal cost models to be developed, capable of costing both space and time usage. This requires some restrictions on the expression language in cases which are cost or space critical. The first version of the language is deliberately rather sparse, allowing experimentation with essential features but omitting some desirable syntax or other language features, such as overloaded polymorphic types. Future versions of the language should address these omissions. The language definition does support a wide range of (particularly numeric) basic types. This is because issues of type coercion and type safety are fundamental to ensuring both correctness and security.

Both system level and process level exceptions are supported, including the ability to set timeouts for expression computations. Exceptions may be raised from within the expression language but can only be handled by the process language. This reduces the cost of handling exceptions and maintains a pure expression language, as well as simplifying the expression cost calculus.

A radical design decision for high reliability systems is the use of automatic memory management techniques. Automatic memory management has the advantage of reducing errors due to poor manual management of memory. The disadvantage lies in terms of excessive time or space usage. Hume implementations will use static analysis tools to limit space usage, and will incorporate recent developments in bounded-time memory management techniques.

1.2 Language Structure

In common with other coordination language approaches such as Linda [**?**], Hume takes a layered approach. The outermost layer is a static *declaration language* that provides definitions of types, streams etc. to be used in the dynamic parts of the language. The innermost layer is a conventional *expression language* which is used to define values and (potentially higher-order) functions. Finally, the middle layer is a coordination language that links functions into possibly concurrent processes.

1.2.1 The Hume Expression Language

The Hume *expression language* is a purely functional, primitive recursive language with a strict semantics. It is intended to be used for the description of single, one-shot, non-reentrant processes. The expression language has statically provable properties of:

1. *determinism*;
2. *termination*; and
3. *bounded time and space behaviour*

through the provision of appropriate type systems and semantics (Appendices B and C).

Note that the expression language has no concept of external, imperative state. Such state considerations are encapsulated entirely within the coordination language.

1.2.2 The Hume Coordination Language

The Hume *coordination language* is a finite state language for the description of multiple, interacting, re-entrant processes built from the purely functional expression layer. The coordination language is designed to have statically provable properties that include both *process equivalence* and *safety* properties such as the absence of deadlock, livelock or resource starvation. The coordination language also inherits properties from the expression language that is embedded within it.

The basic unit of coordination is the *box*, an abstract notion of a process that specifies the links between its input and output channels in terms of functional expressions, and which provides exception handling facilities including timeouts and system exceptions The coordination language is responsible for interaction with external, imperative state through streams and ports that are ultimately connected to external devices.

1.3 Tools to Support the Hume Language

We envisage the construction of a number of tools to support Hume programmers. By *tools* we understand formal definitions and calculi, as well as software language processors such as compilers, interpreters, type checkers etc. The tools we intend to produce are:

- the Hume language definition: syntax, types and semantics;
- the Hume abstract machine(HAM)/abstract machine code(HAMC): syntax, types and semantics;
- a compiler from Hume source → HAMC supporting separate compilation;
- a compiler from HAMC → native assembler code.

The Hume language semantics tools comprise:

operational semantics — reference interpreter;
axiomatic semantics — correctness prover;
termination semantics — termination prover;
specification notation — refinement calculus;
rule checker — literate specification;
cost calculus;
transformation system.

The abstract machine tools comprise:

- the interpreter including a profiler and instrumentor;
- a transformation system.

The HAMC to native assembly code tools include: the run-time system; a profiler; and aninstrumentor.

1.4 The Hume Research Programme

Our intention is for the Hume design to proceed in a series of planned stages.

Our immediate priorities are:

- the core language definition - syntax, types & type system, and operational semantics;
- a reference interpreter;
- a set of reference Hume programs.

We will then develop the HAM/HAMC formal definition a HAMC interpreter, and a Hume → HAMC compiler, using the reference programs to ensure behavioural consistency with the reference interpreter.

We will next consider the cost/termination calculi systems; the profiler/instrumentor; and the program transformer; and use them to analyse the reference program set.

We envisage native code compilation, proof, specification, and refinement as longer term objectives.

We see proof of formal properties of language processing tools as central to this programme, notably,

1. consistency with the Hume definition;
2. preservation of the meaning and behaviour of Hume programs.

1.5 Related Work

1.5.1 Embedded Systems

Real-time embedded systems are typically programmed using low-level languages and techniques. Some high level languages have, however, been designed or adapted for such use.

Ada is widely used for embedded systems, and many tools have been constructed to assist the understanding of space and time behaviour [3]. Compared with ANSI standard Ada, Hume provides much higher level of abstraction with a far more rigorously defined semantics, which is specifically designed to support cost semantics.

There has been recent interest in using variants of Java as the basis for embedded systems, though to our knowledge there is as yet no specifically safety-critical design. Two interesting variants are Embedded Java [57] and RTJava [?], for soft real-time applications. Like Hume, both languages support dynamic memory allocation with automatic garbage collection and provide strong exception handling mechanism. The primary differences from Hume are the omission of arbitrary recursion, an absence of formal design principles, the use of a single-layered approach in which coordination is merged with computation, and of course the use of an object-oriented expression language rather than one that is purely functional. We believe that the design choices made here are more suitable for applications where safety or correctness are important. For example, the use of purely functional rather than dynamically-linked object-oriented design allows straightforward static reasoning about the meaning of programs, at the cost of convenience in modifying a running system.

1.5.2 Real-Time Safety-Critical Systems

Typically, a formal approach to designing safety-critical systems progresses rigorously from requirements specification to systems prototyping. Languages and notations for specification/prototyping provide good formalisms and proof support, but are often weak on essential support for programming abstractions, such as data structures and recursion. Implementation therefore usually proceeds less formally, or more tediously, using conventional languages and techniques. Hume is intended to simplify this process by allowing more direct implementation of the abstractions provided by formal specification languages. Alternatively, in a less formal development process, it can be used to give a higher-level, more intuitive implementation of a real-time problem.

Specification Languages. Safety-critical systems have strong time-based correctness requirements, which can be expressed formally as properties of *safety*, *liveness* and *timeliness* [7]. Formal requirements specifications are expressed using notations such as temporal logics (e.g. XCTL [17] or MTL [28]), non-temproal logics (e.g. RTL [24]), or timed process algebras (e.g. LOTOS-T [40], Timed CCS [62] or Timed CSP [51]). Such notations are deliberately non-deterministic in order to allow alternative implementations, and may similarly leave some or all timing issues unspecified. It is essential to crystallise these factors amongst others when producing a working implementation.

Non-Determinism. Although non-determinism may be required in specification languages such as LOTOS [21], it is usually undesirable in implementation languages such as Hume, where predictable and repeatable behaviour is required [7]. Hume thus incorporates deterministic processes, but with the option of fair choice to allow the definition of alternative acceptable outcomes. Because of the emphasis on hard real-time, it is not possible to use the event synchronising approach based on *delayed timestamps* which has been adopted by e.g. the concurrent functional language BRISK [18]. The advantage of the BRISK approach is in ensuring strong determinism without requiring explicit specifications of time constraints as in Hume.

Synchronicity. *Synchronous* languages such as Signal [5], Lustre [10], Esterel [8, 6] or the visual formalism Statecharts [16] obey the *synchrony hypothesis*: they assume that all events occur instantaneously, with no passage of time between the occurrence of consecutive events [4]. In contrast, *asynchronous* languages, such as the extended finite state machine languages Estelle [22] and SDL [23], make no such assumption. Hume uses an asynchronous approach, for reasons of both expressiveness and realism. Like Estelle and SDL, it also employs an asynchronous model of communication and supports asynchronous execution of concurrent processes.

Persistency. In order to ensure essential progress even in the absence of some inputs, Hume is deliberately *non-persistent* [7]: the passage of time can force a timeout on an input channel, which can thus influence the choice made by a process. It is also possible for a timeout on an internal computation to have the same effect, although in this case no input will have been consumed. Determinacy is maintained through a strong formal cost model integrated with a formal dynamic semantics which collectively fully prescribe the outcome of a process instance given the inputs that have been provided.

Dynamic Process Networks. The initial Hume design uses a static process network, as with Petri net approaches,but unlike recent innovations such as π-calculus [41]. This simplifies the formal language semantics, and very importantly, allows the total cost to be specified for the active process network, but does prevent the direct definition of e.g. mobile processes. We do anticipate that some forms of dynamic process could be supported without destroying our overall cost semantics, but have not yet explored this issue.

Summary Comparison. As a vehicle for implementing safety-critical or hard real-time problems, Hume thus has advantages over widely-used existing language designs. Compared with Estelle or SDL, for example, it is formally defined, deterministic, and provably bounded in both space and time. These factors lead to a better match with formal requirements specifications and enhance confidence in the correctness of Hume programs. Hume has the advantage over Lustre and Esterel of providing asynchronicity, which is required for distributed systems. Finally, it has the advantage over LOTOS or other process algbras of being designed as an implementation rather than specification language: *inter alia* it supports normal program and data structuring constructs, allowing a rich programming environment.

1.5.3 Bounded Time/Space Models

Other than our own work, we are aware of three main studies of formally bounded time and space behaviour in a functional setting [9, 20, 61].

Embedded ML. In their recent proposal for Embedded ML, Hughes and Pareto [20] have combined the earlier *sized type system* [19] with the notion of *region types* [58] to give bounded space and termination for a first-order strict functional language [20]. Their language is more restricted than Hume in a number of ways: most notably in not supporting higher-order functions, and in requiring programmer-specified memory usage.

Inductive Cases. Burstall[9] proposed the use of an extended *ind case* notation in a functional context, to define inductive cases from inductively defined data types. Here, notation is introduced to constrain recursion to always act on a component of the "argument" to the *ind case* i.e. a component of the data type pattern on which a match is made. While *ind case* enables static confirmation of termination, Burstall's examples suggest that considerable ingenuity is required to recast terminating functions based on a laxer syntax.

Elementary Strong Functional Programming. Turner's *elementary strong functional programming* [61] has similarly explored issues of guaranteed termination in a purely functional programming language. Turner's approach separates finite data structures such as tuples from potentially infinite structures such as streams. This allows the definition of functions that are guaranteed to be primitive recursive. In contrast with the Hume expression layer, it is necessary to identify functions that may be more generally recursive. We will draw on Turner's experiences in developing our termination analysis.

Other Related Work. Recent research by Kamareddine and Monin has formlised automatic proofs of termination of recursive functions, by augmenting proof trees with measures that establish an appropriate decreasing property [25]. They have also investigated widening the scope of automatic termination proof from inductive to non-inductive cases [26].

Also relevant to the problem of bounding time costs is recent work on *cost calculi* [52, 54] and *cost modelling* [50, 32, 55], which has so far been primarily applied to parallel computing.

1.6 Papers

Research papers on Hume can be found at the Hume page `http://www-fp.dcs.st-and.ac.uk/hume`.

Chapter 2

Hume Overview

This chapter introduces the Hume language. Section 2.1 describes the set of fundamental types that are supported by the language, together with the operations that are provided on those types. These types are intended to form a fairly minimal set, allowing the construction of realistic programs without requiring complex implementation in the initial stages. In the longer term, we expect to support a richer set of types in later versions of Hume.

One important concern for any such language is the matter of type coercion and conversion, especially between scalar values. Hume therefore provides a wide range of scalar types, and defines precisely the conversions between values of those types. Hume scalar types include booleans, characters (including unicode), variable sized word values, fixed-precision integers (including natural numbers), floating-point values, fixed-exponent real numbers, and exact real numbers.

A second, related concern is the need to specify the sizes of such values. Hume meets this concern by requiring the size of all scalar values to be specified precisely.

In addition to scalar types, Hume supports three kinds of structured type: vectors, lists and tuples. Vector and tuple types are fixed size, whereas lists may be arbitrary sized. All the elements of a single vector or list must have the same type.

More detail on wiring, including fuller examples, can be found in the separate wiring document [15].

2.1 Types

2.1.1 Base Types

All Hume type domains are unpointed [?]. That is, there is no explicit notion of an *undefined* value ($\perp$) in each type domain. The Hume *base types* are shown in Table 2.1. The type `bit` is a synonym for `word 1`, and the type `byte` is a synonym for `word 8`.

The basic operations provided for each type are shown in Table2.2. The integer division and remainder operators (mod) have the property that `a == (a 'div' b)*b + (a 'mod' b)`. The result of `x 'div' y` has the same sign as `x * y` and is truncated towards zero. The value of `x ** 0` is 1 for any `x`, including zero. For `word`, `& | ^ ~` are bitwise and, inclusive or, exclusive or and negation respectively. The `<<` and `>>` operations pad to right/left with `0`s respectively, as required.

bool	boolean i.e. true , false
char	8 bit - ISO Latin-1 denoted by: ' <printable> '
unicode	16 bit Unicode
word <precision> int <precision>	bits of specified size, in the range $0 \ldots\ 2^n - 1$ 2's complement integer of specified bit size, in the range $-2^{n-1} \ldots\ 2^{n-1} - 1$
nat <precision>	natural number i.e. $\geq$ 0 of specified bit size, in the range $0 \ldots\ 2^n - 1$
float <precision>	floating point number of specified bit size (IEEE representation)
fixed <precision> [@ (2|10|16) [** <exponent>]]	fixed exponent real of specified bit size, and optional base / exponent
exact	exact real number

Table 2.1: Hume base types

bool	not, and, or
int nat	+ - * div mod unary - — not provided for nat ** — power < <= == >= > !=
float fixed exact	+ - * / unary - sin cos tan asin acos atan log exp sqrt ** — power < <= == >= !=
word	>> << — left/right shift rotl rotr — rotate left/right & | ^ ~ — bitwise operations < <= == >= !=
char unicode	< <= == >= > !=

Table 2.2: Basic operations on base types

vector	fixed length sequence of uniform type with a lower bound of 1 type: vector <size> of <type> where <size> ≥ 0 denoted by: < <expr1>, ... , <exprN> > where N ≥ 0
tuple	fixed length sequence of mixed type type: (<type1>, ... , <typeN>) where N ≥ 0 denoted by: (<expr1>, ... , <exprN>) where N $= 0$ or N > 1
list	variable length sequence of uniform type type: [<type>] denoted by: [<expr1>, ... , <exprN>] where N ≥ 0
discriminated union	declared by: data <id> = <id1> <type11> ... <type1N> | ... where N ≥ 0 type: <id> <type1> ... <typeN> where N ≥ 0 denoted by: <id> <expr1> ... <exprN> where N ≥ 0
stream	type: stream <type>
port	type: port <type>
time	type: time
bandwidth	type: bandwidth

Table 2.3: Structured types

2.1.2 Structured types

The Hume structured types are shown in Table 2.3, with the corresponding operations shown in Table 2.4. Streams and ports may only be associated with boxes — see below. The type string is a synonym for [char]. Strings may be constructed by denotation: " <printable1> ... <printableN> ".

2.1.3 Type Conversions

There are two kinds of type conversion. Casting (or viewing) involves treating a value as if it belonged to another equivalent type. One type may be cast to another using <expr> :: <type> if there is no loss of information when converting from a value of the type of <expr> to <type>, and if the conversion can be done with no runtime cost.

The second form of type conversion is coercion. In this case, there may be loss of information and there may also be a runtime cost. The corresponding form is <expr> as <type>.

The conformancy between base types is as shown in Table 2.5. The most significant bit in a word is to the left. Base values are right aligned and left padded.

Coerced structured types:

- must have the same number of elements at all levels
- are aligned top down, recursively, element by element left to right

2.1.4 Exceptions

Exceptions are:

- declared by: exception <id> <type>
  within declarations (Section 2.2);

vector	construction by denotation selection by pattern matching @ <expr> — select <expr>th element length + — vector concatenation < <= == $\geq$ > e
Tuples	construction by denotation @ <expr> — select <expr>th element selection by pattern matching < <= == $\geq$ > e
Lists	: — list constructor construction by denotation length hd tl @ <expr> — select <expr>th element selection by pattern matching + — list concatenation == e

Table 2.4: Basic Operations on Structured types

Table 2.5: Conformancy between Hume base types.

	bool	int	nat	float	fixed	exact	char	unicode	word
bool		Y	Y		Y	Y			Y
int	Y(1)		Y(2)	Y	Y	Y	Y(3)	Y(4)	Y(5)
nat	Y(1)	Y(6)		Y	Y	Y	Y(3)	Y(4)	Y(5)
float		Y(7)	Y(8)		Y(7)	Y			Y(5)
fixed	Y(1)	Y(7)	Y(8)	Y(7)		Y			Y(5)
exact	Y(1)	Y(9)	Y(9)	Y(9)	Y(9)				Y(9)
char		Y	Y					Y	Y(5)
unicode		Y	Y						Y(5)
word	Y	Y(5)	Y(5)	Y(5)	Y(5)	Y(9)	Y(5)	Y(5)	

Notes

1. int = 0 or int = 1
2. int $\geq$ 0
3. 0 <= int <= 255
4. 0 <= int <= 2**16-1
5. 0 <= int <= 2**word precision-1
6. nat precision <= int precision-1
7. trunc, round, ceiling
8. float > 0 and as int
9. subject to further discussion...

- raised by: `raise <id> <expr>`
  within expressions (Section 2.3.9);
- handled by boxes (Section 2.4.3).

System exceptions may be handled either within a box or by a general system handler. If a box defines a handler for a system exception, and the exception is raised as a consequence of executing that box, then the specified handler is called. If a box fails to define a handler for a system exception and that system exception occurs during the process of executing the box, then the general system handler is called. There must be precisely one general handler for each system exception. The system exceptions are:

`Div0`	division by 0
`Overflow/Underflow`	numeric overflow/underflow
`OutOfBounds`	out of bounds vector index
`NullStructure`	null structure access
`StructureTooDeep`	structure depth exceeded
`NoMoreMemory`	memory allocation failed
`Hume`	abject scepticism
`NoSuchStream`	failed stream designation
`EmptyStream`	empty stream access
`NoSuchPort`	failed port designation
`NoMoreForks`	no more forks

2.2 The Declaration Language

The declaration language introduces types and values that scope over either or both the coordination and expression languages. The coordination language is embedded in terms of box and wiring declarations while the expression language is embedded in terms of function declarations.

While it is possible to define recursive and mutually recursive functions, it is not possible to do the same for simple values.

Declarations to be discussed here.

2.3 The Expression Language

The Hume expression language follows the design of widely used functional languages such as Standard ML [42] and Haskell [?]. Like Standard ML, the language has a strict evaluation order. This allows tight cost functions to be derived for Hume expressions and allows a relatively simple semantics of exceptions to be specified (see Appendix C). Like Haskell, the Hume expression language is purely functional. Unlike either Haskell or ML, the language is restricted to *primitive recursive* forms of functions. This is to simplify the automatic derivation of precise cost functions. The syntax of the Hume expression language is broadly based on that of Haskell, and is fully described in Appendix A. The formal dynamic semantics of the Hume expression language is given in Appendix C.

2.3.1 Constants

Constants are simple constant values covering the basic Hume types.

```
<expr> ::= ...
```

```
          <constant>

<constant> ::=
          <intconst>
        | <floatconst>
        | <boolconst>
        | <charconst>
        | <stringconst>
        | <wordconst>
        | <timeconst>
```

2.3.2 Variables

Variables are defined either in function declarations, constant declarations, or in pattern matches. In the first two cases, their value is as specified in the corresponding declaration. The value of a constant can be obtained without runtime computation, that of a variable declared in a function declaration may require computation. In the final case, the value of the variable is obtained by deconstructing the matched expression as a consequence of the pattern matching operation. No further computation is required.

```
<expr> ::= ...
        | <varid>                      variable/constant
```

2.3.3 Constructors

Constructors are used build new data structures. They are defined by `union` declarations to be components of some discrimated union type.

```
<expr> ::= ...
        | <conid> <expr1> ... <exprn>   constructor appl., n $\ge$ 0
```

2.3.4 Tuples, Lists and Vectors

Tuples lists and vectors are created in a similar way to user-defined constructors, but for convenience, special syntax is provided. It is not possible to create a tuple of one element. An "empty" tuple can be created using the syntax `()`.

```
<expr> ::= ...
        | "[" <exprs> "]"              list
        | "()"                         empty tuple
        | "(" <expr> "," <exprs> ")"   tuple
        | "<" <exprs> ">"              vector

<exprs> ::=
        <expr0> "," ... "," <exprn>    n $\ge$ 0
```

2.3.5 Function Applications

Hume function applications have a strict semantics. All arguments to a function are evaluated from right-to-left before the function is called. All functions in Hume must be fully applied: Currying and partial applications are not supported. Higher-order functions *are* supported, but may have cost implications as discussed in Section **??**.

```
<expr> ::= ...
        | <expr1> <op> <expr2>          binary operator
        | <varid> <expr1> ... <exprn>   function appl., n $\ge$ 1
```

2.3.6 Case Expressions

Case expressions must be *complete* in the sense that all possible values of the type of the expression which is discrimated on must be matched by one or more of the specified patterns. In determining whether an expression matches a pattern, the patterns are matched in order top-to-bottom, left-to-right. In matching a pattern, a variable or wildcard matches any value (in the former case also creating a new binding for the variable to the matched sub-expression), any other pattern matches if the pattern constructor matches the expression's constructor and all sub-expressions match all corresponding sub-patterns. Patterns are left-linear (there are no repeated variables within a single pattern).

```
<expr> ::= ...
        | "case" <expr> "of" <matches>  case expression

<matches> ::=
        <match1> "|"..."|" <matchn>     n $\ge$ 1

<match> ::=
       <patt> "->" <expr>
```

Conditionals can be seen as a special case of case-expressions where the expression being discrimated on has type `bool` and there are precisely two alternatives depending on whether the value of the expression is `true` or `false`.

```
        | "if" <expr1> "then" <expr2>   conditional
                      "else" <expr3>
```

2.3.7 Local Declarations

Local declarations are used to introduce one or more bindings of variables to expressions with a limited scope. The name introduced by a binding is visible within other bindings in the same set of declarations as well as within the target expression. All value bindings are evaluated before the body of the expression is evaluated.

```
<expr> ::= ...
        | "let" <fundecls> "in" <expr>  local definition
```

2.3.8 Type Expressions and Type Coercions

Types can be given to an expression using the "::" operator. In this case, the compiler will verify statically that the expression has the specified type, or can be "viewed" as the specified type. These operations are purely static and have no dynamic effect.

More powerful dynamic *type coercions* can be specified using "as"-expressions. A table of types that are compatible for coercion purposes is given in Section **??**. In this case, some computation may be required to coerce a value from one type to another. Unlike the use of the "::" operator, a type coercion may not be reversible: information may be lost during the coercion process, for example. Coercions must therefore be treated with care.

```
<expr> ::= ...
        | <expr> "::" <exprtype>        type cast/view
        | <expr> "as" <exprtype>        type coercion
```

2.3.9 Exceptions

Exceptions can be raised in any expression. The exception is propagated immediately it is raised to the enclosing *box*, which must provide a handler to handle the exception.

```
<expr> ::= ...
        | "raise" <exnid> <expr>        raise an exception
```

2.3.10 Timeouts

Timeouts specify that evaluation of the associated expression must complete within the specified constant time. Failure to do some causes the exception `timeout` to be raised.

```
<expr> ::= ...
        | <expr> "within" <cexpr>
```

2.3.11 Constant Expressions

In some places, expressions have a statically fixed value. This is indicated by `<cexpr>`. Such expressions may include variables, constructors, constants, and predefined operators on such values, but may not include user-defined function calls, raise expressions, timeouts or case/if/let expressions where any of the above rules are violated, or which use any non-constant variable identifiers other than as the sole result of the expression. The compiler will evaluate such expressions at compile-time and generate code to ensure that the appropriate value or variable is loaded in constant time at runtime.

```
<cexpr> ::= <expr>
```

2.4 The Coordination Language

This section describes the Hume coordination language. The formal dynamic semantics of this language is given in Appendix C.

2.4.1 Boxes

The Hume unit of coordination is the *box*. A box has a unique name, specified in its *prelude*. A box has *inputs* and *outputs* termed *ins* and *outs*. Ins and outs are fixed width sequences of *inout* type. An *inout* type is any Hume type excluding a function, stream, port or exception. A box's *ins* and *outs* are specified in its prelude. Each *in* and *out* has a unique name, and is typed. The exceptions a box handles are specified in the box's prelude.

```
<boxdecl> ::= <prelude> <body>

<prelude> ::=
        "box" <boxid>
        "in"  <inoutlist>
```

```
        "out" <inoutlist>
        [ "handles" <exnidlist> ]

<inoutlist> ::=
        <inout1> "," ... "," <inoutn>           n $\ge$ 1

<inout> ::=
        <varid> "::" <exprtype> [ "timeout" <cexpr> ]
```

Each unique stream or port may be associated with only one box.

2.4.2 Box Bodies

The body of a box consists of a set of matches against input values, an optional timeout covering all the matches, plus the exception handlers that apply during each iteration of the body.

```
<body> ::=
        ( "match" | "fair" )
        <matches>
        [ "timeout" <cexpr> ]
        [ "handle" <handlers> ]
```

Each `<match>` in a box must have:

1. a pattern component `<patt>` which is type consistent with the `in` declaration; and
2. an expression component `<expr>` which is type consistent with the `out` declaration.

Top-level patterns may include `*`s. The purpose of a `*` is to indicate that the corresponding input should not be matched or consumed.

Matching may be either sequential (unfair) or "fair". Rules introduced by the `match` keyword are matched in order from top to bottom. The first rule (if any) that fully matches the inputs is selected. Thus a single rule may be matched repeatedly if the same inputs are encountered. In some cases, this can result in certain rules never being used. Fair matching, in contrast, guarantees that all rules are given an equal probability of being matched.

2.4.3 Exception Handlers

There must be a `<handler>` for each exception specified in the box's `handles` clause. All non-system exceptions that can be raised by any expression within the body of the box, or which occur through input timeouts, must be handled by an explicit handler. No handler can perform any computation.

```
<handlers> ::=
        <handler1> "|" ... "|" <handlern>       n $\ge$ 1

<handler> ::=
        <hpatt> "->" <cexpr>

<hpatt> ::=
        <exnid> <patt1> ... <pattn>             n $\ge$ 1
```

Every `<handler>` in a box must have:

1. a `<handlepatt>` corresponding to an entry in the `handles` declaration; and
2. a `<handleout>` which is type consistent with the `out`

2.4.4 Wiring

Boxes are wired together by specifying for each *in* or *out*, the corresponding source or destination box's *in* or *out*, or a stream, or a port, to which it is connected. Wires may either be specified for a complete set of box sources and destinations or individually for each input/output pair.

```
<wiringdecl> ::= ...
        | "wire" <boxid> <sources> <dests>
        | "wire" <link> "to" <link>

<sources>/<dests> ::=
          "(" <link1> "," ... "," <linkn> ")"     n $\ge$ 0

<link> ::=
          <linkspec> <linkprops>

<linkspec> ::=
          <connection>
        | <strid>
        | <portid>

<connection> ::= <boxid> "." <varid>
```

Connection to another box is specified by that box's name extended with the *in* or *out* name. Boxes may be wired to themselves.

Link Properties

Two kinds of link property can be specified for a connection. Connections may have an initial value, which is used to seed the wire. This is introduced by an `initially` property. They may also be traced dynamically during execution (specified by a `trace` property).

```
<linkprops> ::=
          <linkprop1> ... <linkpropn>

<linkprop> ::=
          "initially" <expr>
        | "trace"
```

It is also possible to specify initial values in a declaration that does not form part of a wiring declaration. This is especially useful for defining sets of initialisers using repetition or macros, for example, and allows separation of linkage from initialisation.

```
<wiringdecl> ::= ...
        | "initial" <wireid> <inits>

<inits> ::=
          "(" <init1> "," ... "," <initn> ")"           n >= 1

<init> ::=  <wireid> "=" <expr>
```

For example,

```
initial Ring  {Train1Pos} ( value = Just "Train1" );
```

2.4.5 Box Templates and Instantiation

A template can be defined to give the structure of a box, which is then instantiated to produced a number of boxes.

```
<\begin{verbatim}
<wiredecl> ::=
          "template" <templateid> <prelude> <body>
```

To simplify the construction of complex systems, both boxes and templates may be replicated to give new boxes. The box/template may be replicated either once or a number of times (indicated by `* <intconst>`). Thus, `instantiate t as b * 4` will introduce boxes `b1`, `b2`, `b3` and `b4`.

```
<wiringdecl> ::= ...
        | "replicate"   <boxid> "as" <boxid> [ "*" <intconst> ]
        | "instantiate" <templateid> "as" <boxid> [ "*" <intconst> ]
```

So, for example, a new template `t` can be introduced with corresponding instances and wirings as below.

```
template t in ( input, par :: short ) out ( output, par' :: short )
match
  ( i, v ) -> let res = i + v in ( res, if even res then 1 else 0 )
;

instantiate t as b1;
instantiate t as b2;


wire b1 ( StdIn, b1.par' initially 0 )    ( b2.input, b1.par );
wire b2 ( b1.input, b2.par' initially 0 ) ( StdOut, b2.par );
```

2.4.6 Wiring Macros

Wiring macros can be introduced by associating a wiring definition with a name and set of parameter names. The parameter names declared on the LHS may be used on the RHS of the wiring macro and substitute the corresponding concrete name. When used on the RHS, such names must be enclosed in braces.

```
<wiringdecl> ::= ...
         | "wire" <wmacid> "(" <id1> ... <idn> ")" "=" <wireid> <sources> <dests>
```

Such macros are used in a similar way to normal wiring declarations. In this case, however, there is only one set of arguments, representing the concrete parameters of the wiring macro, unlike a normal wiring declaration where both source and destination wires are identified.

```
<wiringdecl> ::= ...
         | "wire" <wmacid> <sources>
```

For example,

```
wire Track (
```

2.4.7 Repeated Wiring

Wiring declarations can be repeated under the control of a variable (optionally omitting certain values).

```
<wiringdecl> ::= ...
         | "for" <id> "=" <expr> "to" <expr> [ "except" <excepts> ]
            <wiringdecl>
```

The repetition variable may be used within the wiring declation (enclosed within braces), where it takes on . For example,

```
for i =  0 to 4 except (2, 1)
   instantiate Track as Ring{i};
```

will generate `Ring0`, `Ring3`, `Ring3` as instances of the Track template. It is possible to nest for-loops if required, and it is possible to use both loop variables, static constants and expression macros in the expressions. Note that such loops are part of the static coordination language designed to create a static process network rather than part of the dynamic expression language.

2.4.8 Stream and Port Declarations

Streams and ports are specified in the declation language.

```
<decl> ::= ...
           | "stream" <iodes>
           | "port"  <iodes>

<iodes> ::= ( <strid> | <portid> ) ( "from" | "to" ) <string>
```

The string is a system-specific designator identifying the operating system entity (file, device etc.) that the port or stream is attached to. Semantically, a stream differs from a port in that the latter may be read from or written to repeatedly, whereas the former is read from or written to once only.

Each stream or port must be wired to precisely one *in* or *out*. An *in* may be mentioned in only one `<dest>`. An *out* may, however, be mentioned in more than one `<source>`.

The type of a `<connection>/<stream>/<port>` must match that of the *in* or *out* with which it is associated. Only outputs of boxes or ports/streams that are attached using input (`from`) designators can be wired to `<source>`s. Conversely, only inputs to boxes or ports/streams that are attached using output (`to`) designators can be wired to `<dest>`s.

2.4.9 Expression Macros

Expression macros are used to construct simple compile-time macros that are resolved during construction of the static process network. It is also possible to use compile-time constants in order to construct static wiring.

```
<wiringdecl> ::= ...
          | "macro" <mid> <ids> "=" <expr>
```

Macros and constants are expanded statically at compile-time.

```
constant bmax = 8;

macro next n = n + 1;

instantiate t as b*bmax;

for i = 2 to bmax-1
   wire b{i} ( b{prev i}.output, b{i}.par' ) ( b{next i}.input, b{i}.par );

wire b1 ( StdIn, b1.par' ) ( b2.input, b1.par );
wire b{bmax} ( b{prev bmax}.output, b{bmax}.par' ) ( StdOut, b{bmax}.par );
```

Appendix A

Syntax

This appendix gives a BNF definition of the Hume syntax. The meta-syntax is conventional. Terminals are enclosed in double quotes `" ... "`. Non-terminals are enclosed in angle brackets `< ... >`. Vertical bars `|` are used to indicate alternatives. Constructs enclosed in brackets `[ ... ]` are optional. Parentheses `( ... )` are used to indicate grouping. Ellipses (`...`) indicate obvious repetitions. An asterisk (`*`) indicates zero or more repetitions of the previous element, and a plus (`+`) indicates one or more repetitions. The keyword "union" is accepted as a synonym for "data".

Programs and modules

```
<program> ::=
        "program" <decls>

<module> ::=
        "module" <modid> "where" <decls>
```

Declaration Language

```
<decls> ::=
        <decl1> ";" ... ";" <decln>     n >= 1

<decl> :: =
          "import" <idlist>
        | "export" <idlist>
        | "exception" <exnid> <exprtype>
        | "data" <typeid> <varids> "=" <constrs>
        | "type" <typeid> <varids> "=" <type>
        | "constant" <varid> "=" <cexpr>
        | "stream" <iodes>
        | "port" <iodes>
        | "expression" <expr>
        | <boxdecl>
        | <wiringdecl>
        | <fundecl>

<constrs> ::=
            <conid1> <type11> ... <type1n>      m > 0, n >= 0
        "|" ...
        "|" <conidm> <typem1> ... <typemn>

<iodes> ::=
        ( <streamid> | <portid> ) ( "from" | "to" ) <string>
        [ "timeout" <cexpr> ]

<fundecl> ::=
          <varid> "::" <type>
        | <varid> <args> "=" <expr>
        | <patt1> <op> <patt2> "=" <expr>

<args> ::=
        <patt1> ... <pattn>                    n >= 0

<fundecls> ::=
        <fundecl1> ";" ... ";" <fundecln>      n >= 1
```

Types

```
<type> ::=
          <exprtype>
        | "stream" <exprtype>           stream type
        | "port" <exprtype>             port type
        | "time"                        time type
        | "bandwidth"                   bandwidth type

<exprtype> ::=
          <basetype>                    base type
        | "vector" <range> "of" <type>  vector
        | "()"                          empty tuple
        | "(" <type> "," <types> ")"    tuple
        | "[" <types> "]"               list
        | <typeid> <type1> ... <typen>  discr. union, n >= 0
        | <type> "->" <type>            function type
        | "view" <type>                 view as type
        | "(" <exprtype> ")"            grouping

<types> ::=
        <type1> "," ... "," <typen>     n >= 0

<basetype> ::=
          "int" <precision>
        | "nat" <precision>
        | "bool"
        | "char"
        | "unicode"
        | "string" [ <intconst> ]
        | "word" <precision>
        | "float" <precision>
        | "fixed" <precision>
              [ @ ( "2" | "10" | "16" ) [ "**" <intconst> ] ]
        | "exact"

<precision> ::=
        "1" | ... | "64"

<range> ::=
        <intconst1> ".." <intconst2>
```

Expression Language

```
<expr> ::=
          <constant>
        | <varid>                        variable/constant
        | <expr1> <op> <expr2>           binary operator
        | <varid> <expr1> ... <exprn>    function appl., n >= 1
        | <conid> <expr1> ... <exprn>    constructor appl., n >= 0
        | "[" <exprs> "]"                list
        | "()"                           empty tuple
        | "(" <expr> "," <exprs> ")"     tuple
        | "<" <exprs> ">"                vector
        | "case" <expr> "of" <matches>  case expression
        | "if" <expr1> "then" <expr2>    conditional
                     "else" <expr3>
        | "let" <fundecls> "in" <expr>  local definition
        | <expr> "::" <exprtype>         type cast/view
        | <expr> "as" <exprtype>         type coercion
        | "raise" <exnid> <expr>         raise an exception
        | <expr> "within" <cexpr>        timeout
        | "(" <expr> ")"                 grouping
        | "{" <expr> "}"                 macro expansion
        | "*"                            ignored output


<cexpr> ::= <expr>                       constant expression

<exprs> ::=
        <expr0> "," ... "," <exprn>      n >= 0

<matches> ::=
        <match1> "|" ... "|" <matchn>    n >= 1

<match> ::=
       <patt> "->" <expr>
```

Constants

```
<constant> ::=
          <intconst>
        | <floatconst>
        | <boolconst>
        | <charconst>
        | <stringconst>
        | <wordconst>
        | <timeconst>
```

Patterns

```
<patt> :: =
          <constant>
        | <varid>                        variable
```

```
        | <conid>                        nullary constructor
        | "_"                            wildcard
        | "[" <patts> "]"                list pattern
        | "<" <patts> ">"                vector pattern
        | "()"                           empty tuple pattern
        | "(" <patt> "," <patts> ")"     tuple pattern
        | <conid> <patt1> ... <pattn>    discr. pattern, n >= 1
        | "(" <patt> ")"                 grouping
        | "*"                            ignored input

<patts> ::=
        <patt0> "," ... "," <pattn>      n >= 0
```

Coordination language

```
<boxdecl> ::= "box" <boxid> <prelude> <body>

<prelude> ::=
        "in"  <inoutlist>
        "out" <inoutlist>
        [ "handles" <exnidlist> ]

<inoutlist> ::=
        <inout1> "," ... "," <inoutn>           n >= 1

<inout> ::=
        <varid> "::" <exprtype>
```

Boxes

```
<body> ::=
        ("match" | "fair" )
        <boxmatches>
        [ "timeout" <cexpr> ]
        [ "handle" <handlers> ]

<handlers> ::=
        <handler1> "|" ... "|" <handlern>       n >= 1

<boxmatches> ::=
        <matches>
```

```
<handler> ::=
        <hpatt> "->" <cexpr>

<hpatt> ::=
        <exnid> <patt1> ... <pattn>             n >= 1
```

Wiring Macro Language

```
<wiringdecl> ::=
          "replicate" <wireid> "as" <wireid> [ "*" <intconst> ]
        | "instantiate" <wireid> "as" <boxid> [ "*" <intconst> ]
        | "macro" <mid> <ids> "=" <expr>
        | "initial" <wireid> <inits>
        | <templatedecl>
        | <wiredecl>
        | "for" <id> "=" <expr> "to" <expr> [ "except" <excepts> ]
            <wiringdecl>

<inits> ::=
          "(" <init1> "," ... "," <initn> ")"           n >= 1

<init> ::=  <wireid> "=" <expr>

<templateecl> ::= "template" <templateid> <prelude> <body>

<excepts> ::=
          "(" <expr1> "," ... "," <exprn> ")’’          n >= 1
        | <id>
```

Wiring

```
<wiredecl> ::=
          "wire" <wireid> <sources> <dests>
        | "wire" <wireid> <idlist> "=" <wireid> <sources> <dests>

<sources>/<dests> ::=
        "(" <link1> "," ... "," <linkn> ")"     n >= 0

<link> ::=
          <linkspec> <linkprops>

<linkspec> ::=
          <connection>
        | <streamid>
        | <portid>

<connection> ::= <boxid> "." <varid>

<linkprops> ::=
          <linkprop1> ... <linkpropn>

<linkprop> ::=
```

```
          "initially" <expr>
        | "trace"
```

Identifiers

```
<id>     ::= [ <modid> "." ] <localid>

<idlist> ::= <id1> "," ... "," <idn>            n >= 1

<ids> ::= <id1> ... <idn>                       n >= 0

<varids> ::= <varid1> ... <varidn>              n >= 0

<exnidlist> ::= <exnid1> "," ... "," <exnidn>   n >= 1

<boxid>    ::= <id>
<modid>    ::= <id>
<exnid>    ::= <id>
<conid>    ::= <id>
<typeid>   ::= <id>
<varid>    ::= <id>
<streamid> ::= <id>
<portid>   ::= <id>

<wireid>   ::= <id> | <id> "{" <expr> "}" | "{" <id> "}"
```

Lexical Syntax

```
<localid> ::= <letter> ( <letter> | <digit> ) *

<op> ::= ( "+" | "-" | "*" | "/" ... ) *

<intconst>    ::= <digit> +
<floatconst>  ::= <intconst> "." <intconst> [ "e" <intconst> ]
<boolconst>   ::= "true" | "false"
<charconst>   ::= "'" <char> "'"
<stringconst> ::= """ <char> * """
<wordconst>   ::= "0x" <hexdigit> +
<timeconst>   ::= <intconst> <timedes>
<timedes>     ::= "ps" | "ns" | "us" | "ms" | "s" | "min"
<char>        ::= "A" | ... | "Z" | " " | "\t" | "\n" | "\\" |
  "\" <digit> + | "\0x" <hexdigit> +
```

Appendix B

Static Semantics

This appendix defines the static semantics of Hume, giving formal type rules etc.

B.1 Static Semantics: Notation

Except where noted, we use the same notation as the definition of Standard ML [?].

Our static semantics is given in terms of the semantic domain *SemVal* defined below. The notation $D^{(k)}$ is used to denote a sequence of k instances of D, DD, …, DD^{k-1}.

BasVal and *BasCon* are fully defined in Section B.8.

	BasVal =	$\{\ (+), (==), \ldots\ \}$	Basic Values
	BasCon =	$\{\ (:), \mathrm{Nil}, \mathrm{True}, \mathrm{False}, \ldots\ \}$	Basic Constructors
	Con =	BasCon + con	Constructors
	Var =	BasCon + var	Variables
id ∈	Id =	BasVal + Con + Var	Identifiers
E, E' ∈	Env =	⟨ VarEnv, TyVarEnv ⟩	Environments
IE, VE, VE', SE ∈	VarEnv =	{ var ↦ PolyType }	Variable Environments
TE ∈	TypeEnv =	$\{\ \chi\ \}$	Type Environments
AE ∈	TyVarEnv =	$\{\ \alpha\ \}$	Type Variable Environments
$\alpha, \beta \in$	TyVar		Type Variables
$\chi \in$	TyCon		Type Constructors
$\tau, \tau' \in$	Type =	TyVar + TyConType$^{(k)}$ +Type → Type	Monomorphic Types
$\sigma, \sigma' \in$	PolyType =	∀ TyVar$^{(k)}$. Type	Polymorphic Types

Environments are unique maps. They are used by applying the environment to an identifier to give the corresponding entry in the map, for example if E is the environment { var ↦ v }, then E (var) = v. The $m_1 \oplus m_2$ operation updates an environment mapping m_1 by the new mapping m_2. The domains of m_1 and m_2 must be disjoint (this introduces an implicit side-condition on each semantic rule that uses the $\oplus$ operation). The $m_1 \overrightarrow{\oplus} m_2$ operation is similar, but allows values in m_1 to be "shadowed" by those in m_2. It is therefore unnecessary for the domains of m_1 and m_2 to be disjoint. There are two degenerate environments, type environments (which are sets of type constructors) and type variable environments (which are sets of type variables). These environments simply record the presence or absence of their components in the environment, and are used as TE (χ), for example. Where an environment contains sub-environment, the notation E' E is used to select the sub-environment E' from E. The notation E $\oplus_{E'}$ E" updates subenvironment E' of E with the value E".

B.2 Static Semantics: Declarations

Declarations are processed to generate a *variable environment* (VE) mapping identifiers to types, and a *type environment* (TE) recording the arity of type constructors. Declarations may be self-recursive or mutually recursive.

$$\boxed{\mathrm{E} \vdash \mathrm{decls} \Rightarrow \mathrm{E}}$$

$$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \oplus \bigoplus_{j=1}^{n} \mathrm{E'}_j \vdash \mathrm{decl}_i \Rightarrow \mathrm{VE}_i, \mathrm{TE}_i}{\mathrm{E} \vdash \mathrm{decl}_1 \ \ldots\ \mathrm{decl}_n \Rightarrow \mathrm{E} \oplus_{VE} (\bigoplus_{i=1}^{n} \mathrm{VE}_i) \oplus_{TE} (\bigoplus_{i=1}^{n} \mathrm{TE}_i)} \quad (1)$$

$$\boxed{\mathrm{E} \vdash \mathrm{decl} \Rightarrow \mathrm{VE}, \mathrm{TE}}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}{\mathrm{E} \vdash \texttt{constant}\ \mathrm{var} = \mathrm{exp} \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \}, \{\ \}} \quad (2)$$

$$\frac{(\mathrm{SE}\ \textit{of}\ \mathrm{E})\ \mathrm{var'} = \forall\, \alpha_1 \ \ldots\ \alpha_n.\, \tau \qquad [\mathrm{E} \vdash \mathrm{cexpr} \Rightarrow \tau] \qquad \mathrm{E} \vdash \mathbf{Port}\ \tau/\mathbf{Stream}\ \tau \Rightarrow \sigma}{\mathrm{E} \vdash \texttt{port/stream}\ \mathrm{var}\ \texttt{from}\ \mathrm{var'}\ [\texttt{initial}\ \mathrm{cexpr}] \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \}, \{\ \}} \quad (3)$$

$$\frac{\mathrm{E} \vdash \mathrm{matches} \Rightarrow \sigma \qquad \textit{primrec}\ (\ \mathrm{var\ matches}\ )}{\mathrm{E} \vdash \mathrm{var\ matches} \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \}, \{\ \}} \quad (4)$$

$$\frac{\mathrm{E} \vdash \mathrm{op}\ \mathrm{pat}_1\ \mathrm{pat}_2 = \mathrm{exp} \Rightarrow \mathrm{VE}, \mathrm{TE}}{\mathrm{E} \vdash \mathrm{pat}_1\ \mathrm{op}\ \mathrm{pat}_2 = \mathrm{exp} \Rightarrow \mathrm{VE}, \mathrm{TE}} \quad (5)$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}{\mathrm{E} \vdash \mathrm{var} = \mathrm{exp} \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \}, \{\ \}} \quad (6)$$

$$\frac{\mathrm{E} \vdash \mathrm{var} \Rightarrow \sigma \qquad \mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \mathrm{var} :: \mathrm{type} \Rightarrow \{\ \}, \{\ \}} \quad (7)$$

$$\frac{\mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathbf{Exn}\ \tau \Rightarrow \sigma}{\mathrm{E} \vdash \texttt{exception}\ \mathrm{exnid\ type} \Rightarrow \{\ \mathrm{exnid} \mapsto \sigma\ \}, \{\ \}} \quad (8)$$

$$\frac{(\ \mathrm{E}\ \vec{\oplus}_{VE}\ (\bigoplus_{i=1}^{n} \{\ \mathrm{var}_i\ \mapsto\ \alpha_i\ \}\ ))\vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E}\vdash \tau \Rightarrow \sigma}{\mathrm{E}\vdash \texttt{type}\ \mathrm{typeid}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n\ = \mathrm{type}\ \Rightarrow\ \{\ \mathrm{typeid}\ \mapsto\ \sigma\ \},\{\ \}} \quad (9)$$

$$\frac{\begin{array}{c}\sigma\ =\ \forall\,\alpha_1\ \ldots\ \alpha_n.\,\chi\ \alpha_1\ \ldots\ \alpha_n \qquad \mathrm{VE}\ =\ \{\ \mathrm{typeid}\ \mapsto\ \sigma\ \} \qquad \mathrm{TE}\ =\ \{\ \chi\ \}\\ ((\ \mathrm{E}\ \vec{\oplus}_{VE}\ (\bigoplus_{i=1}^{n}\{\ \mathrm{var}_i\ \mapsto\ \alpha_i\ \}\ ))\ \oplus\ \mathrm{VE}\ ),\tau\vdash \mathrm{constrs}\Rightarrow \mathrm{VE'}\end{array}}{\mathrm{E}\vdash \texttt{union}\ \mathrm{typeid}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n = \mathrm{constrs}\Rightarrow (\mathrm{VE}\ \oplus\ \mathrm{VE'}),\mathrm{TE}} \quad (10)$$

$$\frac{\forall i.\ 1\le i\le n,\ (\mathrm{IE}\ \mathit{of}\ \mathrm{E})\ (\mathrm{var}_i)\ =\ \sigma_i}{\mathrm{E}\vdash \texttt{import}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n\Rightarrow \bigoplus_{i=1}^{n}\{\ \mathrm{var}_i\ \mapsto\ \sigma_i\ \},\{\ \}} \quad (11)$$

$$\frac{}{\mathrm{E}\vdash \texttt{export}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n\Rightarrow \{\ \},\{\ \}} \quad (12)$$

$$\boxed{\mathrm{E}\ \vdash\ \mathrm{constrs}\ \Rightarrow\ \mathrm{VE}}$$

$$\frac{\forall i.\ 1\le i\le n,\ \mathrm{E},\tau\vdash \mathrm{constr}_i\Rightarrow \mathrm{VE}_i}{\mathrm{E},\tau\vdash \mathrm{constr}_1\ |\ldots|\ \mathrm{constr}_n\Rightarrow \bigoplus_{i=1}^{n}\mathrm{VE}_i} \quad (13)$$

$$\boxed{\mathrm{E}\ \vdash\ \mathrm{constr}\ \Rightarrow\ \mathrm{VE}}$$

$$\frac{\forall i.\ 1\le i\le n,\ \mathrm{E}\vdash \mathrm{type}_i\Rightarrow \tau_i' \qquad \mathrm{E}\vdash \tau_1'\to\ldots\to\tau_n'\to\tau\Rightarrow\sigma}{\mathrm{E},\tau\vdash \mathrm{conid}\ \mathrm{type}_1\ \ldots\ \mathrm{type}_n\Rightarrow \{\ \mathrm{conid}\ \mapsto\ \sigma\ \}}\ n\ \ge\ 0 \quad (14)$$

B.3 Static Semantics: Programs and Wiring

$$\boxed{\mathrm{IE} \vdash \text{program}}$$

$$\frac{\mathrm{E}_0 \ \oplus_{VE} \ \mathrm{IE} \vdash \text{decls} \ \Rightarrow \ \mathrm{E} \qquad (((\mathrm{E}_0 \ \vec{\oplus} \ \mathrm{E}) \oplus_{VE} \mathrm{IE}) \oplus_{VE} \mathrm{SE}) \vdash \text{boxes} \ \Rightarrow \ \mathrm{VE} \qquad \mathrm{VE} \vdash \text{wires}}{\mathrm{IE} \vdash \texttt{program} \ \text{decls boxes wires}} \tag{15}$$

$$\boxed{\mathrm{E} \vdash \text{boxes} \ \Rightarrow \ \mathrm{VE}}$$

$$\frac{\forall i.\ 1 < i \leq n, \ \ \vdash \text{box}_i \ \Rightarrow \ \mathrm{VE}_i}{\vdash \text{box}_1 \ \ldots \ \text{box}_n \ \Rightarrow \ \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{16}$$

$$\boxed{\vdash \text{box} \ \Rightarrow \ \mathrm{VE}}$$

$$\frac{\mathrm{E} \vdash \text{body} \ \Rightarrow \ \tau \to \tau' \qquad \mathrm{E} \vdash \text{ins} \ \Rightarrow \ \tau \qquad \mathrm{E} \vdash \text{outs} \ \Rightarrow \ \tau' \qquad \mathrm{E} \vdash \tau \to \tau' \ \Rightarrow \ \sigma}{\vdash \texttt{box} \ \text{boxid ins outs body} \ \Rightarrow \ \{ \ \text{boxid} \ \mapsto \ \sigma \ \}} \tag{17}$$

$$\boxed{\mathrm{E} \vdash \text{wires}}$$

$$\frac{\forall i.\ 1 < i \leq n, \ \ \vdash \text{wire}_i}{\vdash \texttt{wiring} \ \text{wire}_1 \ \ldots \ \text{wire}_n} \tag{18}$$

$$\boxed{\vdash \text{wire}}$$

$$\frac{\mathrm{E} \vdash \text{sources} \ \Rightarrow \ \tau \qquad \mathrm{E} \vdash \text{dests} \ \Rightarrow \ \tau' \qquad \mathrm{E} \vdash \text{boxid} \ \Rightarrow \ \tau \to \tau'}{\vdash \ \text{boxid sources dests}} \tag{19}$$

$$\boxed{\mathrm{E} \vdash \text{body} \ \Rightarrow \ \tau}$$

$$\frac{\mathrm{E} \vdash \text{time} \ \Rightarrow \ \textbf{Time} \qquad \mathrm{E} \vdash \text{matches} \ \Rightarrow \ \tau \to \tau' \qquad \mathrm{E} \vdash \text{handlers} \ \Rightarrow \ \tau'}{\mathrm{E}, \text{vs} \vdash \ [\texttt{loop}] \ \text{matches} \ \texttt{timeout} \ \text{time} \ \texttt{handle} \ \text{handles} \ \Rightarrow \ \tau \to \tau'} \tag{20}$$

B.4 Static Semantics: Expressions

The first rule generalises the types of expressions from monotypes to polytypes. The second looks up the type of a variable from the variable environment.

$$\boxed{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}$$

$$\frac{(\mathrm{VE}\ \mathit{of}\ \mathrm{E})\ (\mathrm{id})\ =\ \sigma}{\mathrm{E} \vdash \mathrm{id} \Rightarrow \sigma} \tag{21}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}\ \mathrm{exp} \notin \mathrm{Id} \tag{22}$$

$$\boxed{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau}$$

$$\frac{\mathrm{E} \vdash \mathrm{id} \Rightarrow \forall\, \alpha_1\ \ldots\ \alpha_n.\, \tau \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \tau_i}{\mathrm{E} \vdash \mathrm{id} \Rightarrow \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]} \tag{23}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{char} \Rightarrow \mathbf{Char}} \tag{24}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{string} \Rightarrow \mathbf{String}} \tag{25}$$

$$\frac{\mathrm{E} \vdash \mathrm{con/var} \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau' \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \tau_i}{\mathrm{E} \vdash \mathrm{con/var}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \tau'}\ n > 0 \tag{26}$$

$$\frac{\mathrm{E} \vdash \forall i.\ 1 \leq i \leq n,\ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \tau \Rightarrow \mathbf{List}\ \tau}{\mathrm{E} \vdash [\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ] \Rightarrow \mathbf{List}\ \tau}\ \mathrm{n} \geq 1 \tag{27}$$

$$\frac{}{\mathrm{E} \vdash [\,] \Rightarrow \mathbf{List}\ \tau} \tag{28}$$

$$\frac{}{\mathrm{E} \vdash () \Rightarrow \mathbf{Tuple}_0} \tag{29}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i\ \Rightarrow\ \tau_i}{\mathrm{E} \vdash (\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ )\ \Rightarrow\ \mathbf{Tuple}_n\ \tau_1\ \ldots\ \tau_n} \tag{30}$$

$$\frac{}{\mathrm{E} \vdash <\ \ >\Rightarrow\ \mathbf{Vector}\ \tau} \tag{31}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i\ \Rightarrow\ \tau}{\mathrm{E} \vdash <\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ \ >\Rightarrow\ \mathbf{Vector}\ \tau} \tag{32}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \tau \qquad \mathrm{E} \vdash \mathrm{match}\ \Rightarrow\ \tau \rightarrow \tau'}{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}\ \texttt{of}\ \mathrm{match}\ \Rightarrow\ \tau'} \tag{33}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}_1\ \Rightarrow\ \mathbf{Bool} \qquad \mathrm{E} \vdash \mathrm{exp}_2\ \Rightarrow\ \tau \qquad \mathrm{E} \vdash \mathrm{exp}_3\ \Rightarrow\ \tau}{\mathrm{E} \vdash \texttt{if}\ \mathrm{exp}_1\ \texttt{then}\ \mathrm{exp}_2\ \texttt{else}\ \mathrm{exp}_3\ \Rightarrow\ \tau} \tag{34}$$

$$\frac{\mathrm{E} \vdash \mathrm{decls}\ \Rightarrow\ \mathrm{E'} \qquad \mathrm{E}\ \overrightarrow{\oplus}\ \mathrm{E'} \vdash \mathrm{exp}\ \Rightarrow\ \tau}{\mathrm{E} \vdash \texttt{let}\ \mathrm{decls}\ \texttt{in}\ \mathrm{exp}\ \Rightarrow\ \tau} \tag{35}$$

$$\frac{\mathrm{E} \vdash \mathrm{type}\ \Rightarrow\ \tau \qquad \mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \tau}{\mathrm{E} \vdash \mathrm{exp} :: \mathrm{type}\ \Rightarrow\ \tau} \tag{36}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \tau \qquad \mathrm{E} \vdash \mathrm{type}\ \Rightarrow\ \tau' \qquad \mathit{coerceable}\ (\ \tau,\ \tau'\ )}{\mathrm{E} \vdash \mathrm{exp\ as\ type}\ \Rightarrow\ \tau'} \tag{37}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \tau \qquad \mathrm{E} \vdash \mathrm{exnid}\ \Rightarrow\ \mathbf{Exn}\ \tau}{\mathrm{E} \vdash \texttt{raise}\ \mathrm{exnid\ exp}\ \Rightarrow\ \tau'} \tag{38}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}_2\ \Rightarrow\ \mathbf{Time} \qquad \mathrm{E} \vdash \mathrm{exp}_1\ \Rightarrow\ \tau}{\mathrm{E} \vdash \mathrm{exp}_1\ \texttt{within}\ \mathrm{exp}_2\ \Rightarrow\ \tau} \tag{39}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}\ \Rightarrow\ \tau}{\mathrm{E} \vdash (\ \mathrm{exp}\ )\ \Rightarrow\ \tau} \tag{40}$$

B.5 Static Semantics: Matches

$$\boxed{\mathrm{E} \vdash \mathrm{match} \Rightarrow \tau}$$

$$\frac{\mathrm{E} \vdash \{\ \mathrm{match}\ \} \Rightarrow \tau \qquad \mathrm{E} \vdash \{\ \mathrm{matches}\ \} \Rightarrow \tau}{\mathrm{E,v} \vdash \{\ \mathrm{match} \mid \mathrm{matches}\ \} \Rightarrow \tau} \tag{41}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{match}_i \Rightarrow \tau_i, \mathrm{VE}_i \qquad \mathrm{E}\ \vec{\oplus}_{VE}\ (\bigoplus_{i=1}^{n} \mathrm{VE}_i) \vdash \mathrm{exp} \Rightarrow \tau'}{\mathrm{E} \vdash \{\ \mathrm{pat}_1\ \ldots\ \mathrm{pat}_n\ \rightarrow\ \mathrm{exp}\} \Rightarrow \tau_1 \rightarrow\ \ldots \rightarrow \tau_n \rightarrow \tau'} \tag{42}$$

$$\boxed{\mathrm{E} \vdash \mathrm{pat} \Rightarrow \tau, \mathrm{VE}}$$

$$\frac{}{\mathrm{E} \vdash \_ \Rightarrow \tau, \{\ \}} \tag{43}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{var} \Rightarrow \tau, \{\ \mathrm{var}\ \mapsto\ \tau\ \}} \tag{44}$$

$$\frac{\mathrm{E} \vdash \mathrm{con} \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau' \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{pat}_i \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{E} \vdash \mathrm{con}\ \mathrm{pat}_1\ \ldots\ \mathrm{pat}_n \Rightarrow \tau', \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{45}$$

$$\frac{}{\mathrm{E} \vdash () \Rightarrow \mathbf{Tuple}_0\ , \{\ \}} \tag{46}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{pat}_i \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{E} \vdash (\ \mathrm{pat}_1,\ \ldots,\ \mathrm{pat}_n\ ) \Rightarrow \mathbf{Tuple}_n\ \tau_1\ \ldots\ \tau_n, \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{47}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{pat}_i \Rightarrow \tau, \mathrm{VE}_i}{\mathrm{E} \vdash <\ \mathrm{pat}_1,\ \ldots,\ \mathrm{pat}_n\ > \Rightarrow \mathbf{Vector}\ \tau, \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{48}$$

B.5.1 Static Semantics: Exception Handler Matches

$$\frac{\mathrm{E} \vdash \text{exnid} \Rightarrow \mathbf{Exn}\ \tau \qquad \mathrm{E} \vdash \text{pat} \Rightarrow \tau, \mathrm{VE} \qquad \mathrm{E} \overset{\rightarrow}{\oplus}_{VE}\ \mathrm{VE} \vdash \text{exp} \Rightarrow \tau'}{\mathrm{E} \vdash \text{exnid pat} \ \rightarrow\ \text{exp} \Rightarrow \tau'} \tag{49}$$

$$\frac{\mathrm{E} \vdash \text{handler} \Rightarrow \tau \qquad \mathrm{E} \vdash \text{handlers} \Rightarrow \tau}{\mathrm{E} \vdash \text{handler} \mid \text{handlers} \Rightarrow \tau} \tag{50}$$

B.6 Static Semantics: Type Expressions

$$\boxed{\mathrm{E} \vdash \text{type} \Rightarrow \tau}$$

$$\frac{(\mathrm{VE}\ \mathit{of}\ \mathrm{E})\ (\mathit{tyvar}) = \alpha}{\mathrm{E} \vdash \text{tyvar} \Rightarrow \alpha} \tag{51}$$

$$\frac{(\mathrm{VE}\ \mathit{of}\ \mathrm{E})\ (\mathit{tycon}) = \forall\ \alpha_1\ \ldots\ \alpha_n.\chi\ \alpha_1\ \ldots\ \alpha_n \qquad \forall i.\ 1 \leq i \leq n,\ \mathrm{E} \vdash \text{type}_i \Rightarrow \tau_i}{\mathrm{E} \vdash \text{tycon type}_1\ \ldots\ \text{type}_n \Rightarrow \chi\ \tau_1\ \ldots\ \tau_n} \tag{52}$$

$$\frac{\mathrm{E} \vdash \mathit{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathit{type} \Rightarrow \tau'}{\mathrm{E} \vdash \text{type} \ \rightarrow\ \text{type'} \Rightarrow \tau \rightarrow \tau'} \tag{53}$$

$$\frac{\mathrm{E} \vdash \mathit{type} \Rightarrow \tau}{\mathrm{E} \vdash (\text{type}) \Rightarrow \tau} \tag{54}$$

B.7 Static Semantics: Types

$$\boxed{\mathrm{E} \vdash \tau}$$

$$\frac{(\mathrm{AE}\ \mathit{of}\ \mathrm{E})\ \alpha = n}{\mathrm{E} \vdash \alpha} \tag{55}$$

$$\frac{(\mathrm{TE}\ \mathit{of}\ \mathrm{E})\ \chi = n \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \tau_i}{\mathrm{E} \vdash \chi\ \tau_1\ \ldots\ \tau_n} \tag{56}$$

$$\frac{\mathrm{E} \vdash \tau \Rightarrow \qquad \mathrm{E} \vdash \tau' \Rightarrow}{\mathrm{E} \vdash \tau \rightarrow \tau' \Rightarrow} \tag{57}$$

$$\boxed{\mathrm{E} \vdash \sigma}$$

$$\frac{\mathrm{E} \ \oplus_{AE} \ \{ \alpha_1, \ \ldots, \ \alpha_n \} \vdash \tau}{\mathrm{E} \vdash \forall \, \alpha_1 \ \ldots \ \alpha_n . \, \tau} \tag{58}$$

$$\boxed{\mathrm{E} \vdash \tau \Rightarrow \sigma}$$

$$\frac{\mathrm{E} \vdash \forall \, \alpha_1 \ \ldots \ \alpha_n . \, \tau}{\mathrm{E} \vdash \tau \Rightarrow \forall \, \alpha_1 \ \ldots \ \alpha_n . \, \tau} \tag{59}$$

B.8 Static Semantics: The Initial Environment

The initial environment used in the static comprises type bindings for all values defined in the module *Prelude*, including functions, data constructors, type constructors and exceptions, plus bindings for basic values as given below.

The initial variable environment contains types for the following functions (*BasVal*):

PrimPlusInt $\mapsto$ **Int** $\rightarrow$ **Int**
PrimMulInt $\mapsto$ **Int** $\rightarrow$ **Int**
...

plus types for the standard constructors (*BasCon*):

0 $\mapsto$ **Int** , 1 $\mapsto$ **Int** , ..., 0.0 $\mapsto$ **Float** , 0.1 $\mapsto$ **Float** , ..., True $\mapsto$ **Bool** , False $\mapsto$ **Bool** , 'a' $\mapsto$ **Char** , ..., (:) $\mapsto$ $\forall \, \alpha . \, \alpha \rightarrow \mathbf{List} \ \alpha \rightarrow \mathbf{List} \ \alpha$, Nil $\mapsto$ $\mathbf{List} \ \alpha$

Appendix C

Dynamic Semantics

This appendix defines the Hume dynamic semantics using an axiomatic style. It is divided into five parts: i) overview and definitions; ii) the semantics of declarations; iii) the semantics of processes; iv) the semantics of expressions; and v) the semantics of pattern matches. The semantics assumes that all static checks and translations defined by the static semantics are valid and have been properly carried out.

C.1 Limitations

There are a number of limitations on the semantics given here. Firstly, we do not consider the semantics of imported values. This can be added straightforwardly by extending the initial value environment with bindings for the imported values. Secondly, the semantics of processes assumes that all active processes are scheduled for precisely one step. The status of all processes is then reassessed to determine whether each process is active or inactive. A more flexible semantics would schedule precisely one active process. This modification should not be too hard to incorporate into the semantics. Thirdly, we have not provided a semantics for space overflow. This can be introduced in the same way as the rules for time costs. Fourthly, we need to define the semantics of the *timecost* and *coerce* functions which are used to calculate timeouts and type coercions respectively. We anticipate that the semantics of type coercions can be defined without great difficulty. We anticipate that defining the *timecost* function will prove to be somewhat more difficult, but our experience with defining cost semantics for non-strict languages suggests that this should be tractable for Hume. Finally, the semantics is currently defined only for the synchronous language (i.e. omitting fair matches and $*$). We anticipate extending the semantics to cover these constructs in due course.

C.2 Dynamic Semantics: Notation

The dynamic semantics uses a similar style to that used for the static semantics in Appendix B. Our semantics is given in terms of the semantic domain *SemVal* defined below. We use $\langle \ldots \rangle$ to enclose semantic tuples in the *SemVal* domain. This avoids confusion with the syntactic tuple domains, and allows the direct representation of 1-tuples where necessary. The notation D^* is used to define the domain of all tuples of D: $\langle\rangle$, $\langle D \rangle$, $\langle D, D \rangle$, $\ldots$

BasVal and *BasCon* are fully defined in Section C.7.

	$\mathrm{BasVal} =$	$\{\ \mathrm{PrimPlusInt}, \mathrm{PrimEqInt}, \ldots\ \}$	Basic Values
	$\mathrm{BasCon} =$	$\{\ (:), \mathrm{Nil}, \mathrm{True}, \mathrm{False}, \ldots\ \}$	Basic Constructors
$c \in$	$\mathrm{Con} =$	$\mathrm{BasCon} + \mathrm{con}$	
$t, v, vs \in$	$\mathrm{SemVal} =$	$\mathrm{BasVal} + \mathrm{Con}\ \mathrm{SemVal}^* +$ $\mathrm{SemVal}^* + \mathrm{Exn}$	Semantic Values
$\mathrm{E} \in$	$\mathrm{Env} =$	$\langle\ \mathrm{VarEnv},\ \mathrm{SysEnv}\ \rangle$	Environments
$\mathrm{IE}, \mathrm{VE} \in$	$\mathrm{VarEnv} =$	$\{\ \mathrm{var} \mapsto \mathrm{SemVal}\ \}$	Value Environments
$\mathrm{SE} \in$	$\mathrm{SysEnv} =$	$\{\ \mathrm{var} \mapsto \mathrm{SemVal}^*\ \}$	System Environment
$b \in$	$bool =$	$\{\ true, false\ \}$	Booleans
$\mathrm{W} \in$	$\mathrm{Wire} =$	$\{\ \mathrm{var} \mapsto \langle \mathrm{var}^*, \mathrm{var}^*\ \rangle\ \}$	Wires
$\mathrm{I}, \mathrm{A}, \mathrm{P} \in$	$\mathrm{Process} =$	$\{\ \mathrm{Proc}\ \}$	Processes
	$\mathrm{Proc} =$	$\langle\ \mathrm{var}, \mathrm{SemVal}^*, \mathrm{SemVal}^*, \mathrm{exp}\ \rangle$	Process
$\mathrm{x} \in$	$\mathrm{Exn} =$	$\langle\ \mathrm{var}, \mathrm{SemVal}\ \rangle$	Exceptions

Environments are unique maps from identifiers to values. They are used by applying the environment to an identifier to give the corresponding entry in the map, for example if E is the environment $\{\ \mathrm{var} \mapsto \mathrm{v}\ \}$, then $E\ (\mathrm{var}) = \mathrm{v}$. The $m_1 \oplus m_2$ operation updates an environment mapping m_1 by the new mapping m_2. The $m_1 \overset{\rightarrow}{\oplus} m_2$ operation is similar, but allows values in m_1 to be "shadowed" by those in m_2. Conversely, $e \ominus m$ removes the mapping m from an environment e.

C.3 Dynamic Semantics: Declarations

Declarations are processed to generate an initial environment mapping identifiers to initial values. This environment is used in the dynamic semantics for expressions to determine the value of identifers in function applications and variable expressions (rules 91, 86) and in the semantics of boxes to determine the value attached to a port or stream (rule 82). Declarations may be self-recursive or mutually recursive.

$$\boxed{\mathrm{E} \vdash \mathrm{decls} \Rightarrow \mathrm{E}}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E} \oplus \bigoplus_{j=1}^{n} \mathrm{E'}_j \vdash \mathrm{decl}_i \Rightarrow \mathrm{E'}_i}{\mathrm{E} \vdash \mathrm{decl}_1 \ \ldots \ \mathrm{decl}_n \Rightarrow \mathrm{E} \oplus \bigoplus_{i=1}^{n} \mathrm{E'}_i} \quad (60)$$

Each declaration is processed to produce a corresponding value environment.

$$\boxed{\mathrm{E} \vdash \mathrm{decl} \Rightarrow \mathrm{VE}}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \texttt{constant}\ \mathrm{id} = \mathrm{exp} \Rightarrow \{\ \mathrm{id} \mapsto \mathrm{v}\ \}} \quad (61)$$

$$\frac{(\mathrm{SE}\ \mathit{of}\ \mathrm{E})\ \mathrm{id'} = \mathrm{vs}}{\mathrm{E} \vdash \texttt{port/stream}\ \mathrm{id}\ \texttt{from}\ \mathrm{id'} \Rightarrow \{\ \mathrm{id} \mapsto \langle\ \texttt{true}, \mathrm{vs}\ \rangle\ \}} \quad (62)$$

$$\frac{\mathrm{E} \vdash \mathrm{cexpr} \Rightarrow \mathrm{v} \qquad (\mathrm{SE}\ \mathit{of}\ \mathrm{E})\ \mathrm{id'} = \mathrm{vs} \qquad \mathrm{vs'} = \langle\ \mathrm{v},\ \mathrm{vs}\ \rangle}{\mathrm{E} \vdash \texttt{port/stream}\ \mathrm{id}\ \texttt{from}\ \mathrm{id'}\ \texttt{initial}\ \mathrm{cexpr} \Rightarrow \{\ \mathrm{id} \mapsto \langle\ \texttt{true}, \mathrm{vs'}\ \rangle\ \}} \quad (63)$$

$$\frac{}{\mathrm{E} \vdash \mathrm{var\ matches} \Rightarrow \{\ \mathrm{var} \mapsto \mathrm{matches}\ \}} \quad (64)$$

$$\frac{\mathrm{E} \vdash \mathrm{op}\ \mathrm{pat}_1\ \mathrm{pat}_2 = \mathrm{exp} \Rightarrow \mathrm{E'}}{\mathrm{E} \vdash \mathrm{pat}_1\ \mathrm{op}\ \mathrm{pat}_2 = \mathrm{exp} \Rightarrow \mathrm{E'}} \quad (65)$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \mathrm{var} = \mathrm{exp} \Rightarrow \{\ \mathrm{var} \mapsto \mathrm{v}\ \}} \quad (66)$$

$$\frac{}{\mathrm{E} \vdash \mathrm{var} :: \mathrm{type} \Rightarrow \{\ \}} \quad (67)$$

...

C.4 Dynamic Semantics: Processes

The dynamic semantics of a Hume program is given by the dynamics semantics of the boxes that are defined in the program. This semantics is produced in the context of the declarations and wirings that are specified in the same program. The result of a Hume program is a new environment reflecting the state of new bindings in the system or value environments.

$$\boxed{\mathrm{SE}, \mathrm{IE} \vdash \mathrm{program} \Rightarrow \mathrm{E}}$$

$$\frac{\begin{array}{c}\mathrm{E}_0 \oplus \mathrm{IE} \vdash \mathrm{decls} \Rightarrow \mathrm{E} \qquad \vdash \mathrm{boxes} \Rightarrow \mathrm{P} \qquad \vdash \mathrm{wires} \Rightarrow \mathrm{W} \\ ((\mathrm{E}_0 \overset{\rightarrow}{\oplus} \mathrm{E}) \oplus \mathrm{IE} \oplus \mathrm{SE}),\ \mathrm{W} \vdash \mathrm{P} \Rightarrow \mathrm{E'}\end{array}}{\mathrm{SE}, \mathrm{IE} \vdash \texttt{program}\ \mathrm{decls\ boxes\ wires} \Rightarrow \mathrm{E'}} \tag{68}$$

Box declarations are processed to give a set of initial processes, P.

$$\boxed{\vdash \mathrm{boxes} \Rightarrow \mathrm{P}}$$

$$\frac{\forall i.\ 1 < i \leq n,\quad \vdash \mathrm{box}_i \Rightarrow \mathrm{P}_i}{\vdash \mathrm{box}_1\ \ldots\ \mathrm{box}_n \Rightarrow \bigcup_{i=1}^{n} \mathrm{P}_i} \tag{69}$$

$$\boxed{\vdash \mathrm{box} \Rightarrow \mathrm{P}}$$

$$\frac{}{\vdash \texttt{box}\ \mathrm{boxid\ ins\ outs\ body} \Rightarrow \{\ \langle\ \mathrm{boxid}, \mathrm{ins}, \mathrm{outs}, \mathrm{body}\ \rangle\ \}} \tag{70}$$

`wiring` declarations are processed to give the wiring layout mapping the outputs of boxes or ports/streams to the inputs of other boxes.

$$\boxed{\vdash \mathrm{wires} \Rightarrow \mathrm{W}}$$

$$\frac{\forall i.\ 1 < i \leq n,\quad \vdash \mathrm{wire}_i \Rightarrow \mathrm{W}_i}{\vdash \texttt{wiring}\ \mathrm{wire}_1\ \ldots\ \mathrm{wire}_n \Rightarrow \bigcup_{i=1}^{n} \mathrm{W}_i} \tag{71}$$

$$\boxed{\vdash \mathrm{wire} \Rightarrow \mathrm{W}}$$

$$\frac{\mathrm{W} = \{\ \mathrm{boxid} \mapsto \langle\ \mathrm{sources}, \mathrm{dests}\ \rangle\ \}}{\vdash \mathrm{boxid\ sources\ dests} \Rightarrow \mathrm{W}} \tag{72}$$

The set of processes is split into active (A) and inactive processes (I). A process is active if input is available on all its input channels, *or* if a timeout has been raised on any input channel.

$$\boxed{\mathrm{E,W} \vdash \mathrm{P} \Rightarrow \mathrm{P,P}}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E,W} \vdash \mathrm{P}_i \Rightarrow \mathrm{I}_i, \mathrm{A}_i \qquad \mathrm{I} = \bigcup_{i=1}^{n} \mathrm{I}_i \qquad \mathrm{A} = \bigcup_{i=1}^{n} \mathrm{A}_i}{\mathrm{E,W} \vdash \{\ \mathrm{P}_1, \ldots, \mathrm{P}_n\ \} \Rightarrow \mathrm{I,A}} \tag{73}$$

Rules 74–77 determine whether individual inputs are available or have timed out.

$$\boxed{\mathrm{E,W} \vdash \mathrm{P} \Rightarrow \mathrm{P,P}}$$

$$\frac{\begin{array}{c}\mathrm{P} = \langle\ \mathrm{boxid, ins, outs, body}\ \rangle \qquad \mathrm{W\ (boxid)} = \langle\ \mathrm{wins, wouts}\ \rangle \\ \mathrm{E} \vdash \mathrm{wins} \Rightarrow \mathrm{b, b'} \qquad \mathrm{I,A} = \mathit{if}\, b \vee b'\ \mathit{then}\ \{\ \}, \{\ \mathrm{P}\ \} \mathit{else}\ \{\ \mathrm{P}\ \}, \{\ \}\end{array}}{\mathrm{E,W} \vdash \mathrm{P} \Rightarrow \mathrm{I,A}} \tag{74}$$

$$\boxed{\mathrm{E} \vdash \mathrm{ids} \Rightarrow \mathit{bool, bool}}$$

$$\frac{\mathrm{E} \vdash \mathrm{id}_1 \Rightarrow \mathit{true}, \mathrm{b} \qquad \mathrm{E} \vdash \mathrm{id}_2\ \ldots\ \mathrm{id}_n \Rightarrow \mathrm{b', b''}}{\mathrm{E} \vdash \mathrm{id}_1\ \ldots\ \mathrm{id}_n \Rightarrow \mathrm{b'},\ (\mathrm{b} \vee \mathrm{b''})} \tag{75}$$

$$\frac{\mathrm{E} \vdash \mathrm{id}_1 \Rightarrow \mathit{false}, \mathrm{b} \qquad \mathrm{E} \vdash \mathrm{id}_2\ \ldots\ \mathrm{id}_n \Rightarrow \mathrm{b', b''}}{\mathrm{E} \vdash \mathrm{id}_1\ \ldots\ \mathrm{id}_n \Rightarrow \mathit{false},\ (\mathrm{b} \vee \mathrm{b''})} \tag{76}$$

$$\boxed{\mathrm{E} \vdash \mathrm{id} \Rightarrow \mathit{bool, bool}}$$

$$\frac{\mathrm{E\ (id)} = \langle\ \mathrm{b,\ vs}\ \rangle \qquad \mathrm{b'} = \mathit{if} \mathrm{vs} = \langle\rangle \vee \mathit{hd}\ \mathrm{vs} \neq \langle\ \mathrm{Timeout}, \langle\ \rangle\ \rangle\ \mathit{then\ false else\ true}}{\mathrm{E} \vdash \mathrm{id} \Rightarrow \mathrm{b, b'}} \tag{77}$$

Processes are split into active/inactive sets, and all active processes are scheduled for one step, yielding a new environment.

$$\boxed{\mathrm{E, W} \vdash \mathrm{P} \Rightarrow \mathrm{E}}$$

$$\frac{\mathrm{E, W} \vdash \mathrm{P} \Rightarrow \mathrm{I, A} \qquad \mathrm{E, W} \vdash \mathrm{I, A} \Rightarrow \mathrm{E'}}{\mathrm{E, W} \vdash \mathrm{P} \Rightarrow \mathrm{E'}} \tag{78}$$

Processes are scheduled repeatedly until the set of active processes becomes empty.

$$\boxed{\mathrm{E, W} \vdash \mathrm{P, P} \Rightarrow \mathrm{E}}$$

$$\frac{\mathrm{E, W} \vdash \mathrm{I, A} \Rightarrow \mathrm{E', I', A'} \qquad \mathrm{E', W} \vdash \mathrm{I', A'} \Rightarrow \mathrm{E''}}{\mathrm{E, W} \vdash \mathrm{I, A} \Rightarrow \mathrm{E'', I', A'}} \ \mathrm{A} \neq \{\,\} \tag{79}$$

When there are no further active processes, the program terminates.

$$\frac{}{\mathrm{E, W} \vdash \mathrm{I}, \{\,\} \Rightarrow \mathrm{E}} \tag{80}$$

Each active process is executed for one step and the output redirected to the input specified in the wiring specification. All processes are then reassessed to determine their new activity status.

$$\boxed{\mathrm{E, W} \vdash \mathrm{P, P} \Rightarrow \mathrm{E, P, P}}$$

$$\frac{\begin{array}{c} \forall i.\ 1 \leq i \leq \mathit{card}(\mathrm{A}),\ \mathrm{E, W} \vdash \mathrm{A}_i \Rightarrow \mathrm{outs}_i,\ \mathrm{E}_i^I, \mathrm{E}_i^O \\ \mathrm{E'} = \bigcup_{i=1}^{\mathit{card}(\mathrm{A})} \mathrm{E}_i^I \ \vec{\oplus} \bigcup_{i=1}^{\mathit{card}(\mathrm{A})} \mathrm{E}_i^O \\ \mathrm{E} \ \vec{\oplus} \ \mathrm{E', W} \vdash \mathrm{I} \cup \mathrm{A} \Rightarrow \mathrm{I', A'} \end{array}}{\mathrm{E, W} \vdash \mathrm{I, A} \Rightarrow (\mathrm{E} \ \vec{\oplus} \ \mathrm{E'}), \mathrm{I', A'}} \tag{81}$$

A process is executed by determining the value of each of its inputs, and then executing the body of the process in the context of those values. The new values of the inputs and outputs are returned.

$$\boxed{\mathrm{E} \vdash \mathrm{P} \Rightarrow \mathrm{v}, \mathrm{E}, \mathrm{E}}$$

$$\frac{\begin{array}{c} \mathrm{W}\ (\mathrm{boxid}) \ = \ \langle\ \mathrm{wins}, \mathrm{wouts}\ \rangle \\ n \ = \ \mathit{card}(\mathrm{wins}) \\ \mathrm{SE} \ = \ \mathit{SE\ of\ E} \\ \mathrm{vs} \ = \ \langle\ \mathit{snd}(\mathit{SE}\ (\mathrm{wins}_1)),\ \ldots,\ \mathit{snd}(\mathit{SE}\ (\mathrm{wins}_n))\ \rangle \\ \mathrm{E}, \mathrm{vs} \vdash \mathrm{body} \Rightarrow \mathrm{vs'} \\ \mathrm{SE}^I \ = \ \{\ \forall i.\ 1 \le i \le n,\ \mathrm{wins}_i \ \mapsto\ \langle\ \mathit{isport}\ \mathrm{wins}_i, \mathit{tl}\ \mathrm{vs}_i\ \rangle\ \} \\ \mathrm{SE}^O \ = \ \{\ \forall i.\ 1 \le i \le \mathit{card}(\mathrm{wouts}),\ \mathrm{wouts}_i \ \mapsto\ \langle\ \mathit{true}, [\mathrm{vs'}_i]\ \rangle\ \} \end{array}}{\mathrm{E}, \mathrm{W} \vdash \langle\ \mathrm{boxid}, \mathrm{ins}, \mathrm{outs}, \mathrm{body}\ \rangle \Rightarrow \mathrm{vs'}, \mathrm{SE}^I, \mathrm{SE}^O} \tag{82}$$

The final set of process rules define the semantics of executing a single box body. There are three cases, corresponding to normal execution, an exception or a timeout respectively.

$$\boxed{\mathrm{E}, \mathrm{v} \vdash \mathrm{body} \Rightarrow \mathrm{v}}$$

$$\frac{\mathrm{E} \vdash \mathrm{time} \Rightarrow \mathrm{t} \qquad \mathit{timecost}\ (\ \mathrm{E}, \mathrm{matches}\ (\mathrm{vs})\ ) \ < \ \mathrm{t} \qquad \mathrm{E}, \mathrm{vs} \vdash \mathrm{matches} \Rightarrow \mathrm{v}}{\mathrm{E}, \mathrm{vs} \vdash [\texttt{loop}]\ \mathrm{matches}\ \texttt{timeout}\ \mathrm{time}\ \texttt{handle}\ \mathrm{handles} \Rightarrow \mathrm{v}} \quad v \notin \mathrm{Exn} \tag{83}$$

$$\frac{\begin{array}{c} \mathrm{E} \vdash \mathrm{time} \Rightarrow \mathrm{t} \qquad \mathit{timecost}\ (\ \mathrm{E}, \mathrm{matches}\ (\mathrm{vs})\ ) \ < \ \mathrm{t} \qquad \mathrm{E}, \mathrm{vs} \vdash \mathrm{matches} \Rightarrow \mathrm{v} \\ \mathrm{E}, \mathrm{v} \models \mathrm{handle} \Rightarrow \mathrm{v'} \end{array}}{\mathrm{E}, \mathrm{vs} \vdash [\texttt{loop}]\ \mathrm{matches}\ \texttt{timeout}\ \mathrm{time}\ \texttt{handle}\ \mathrm{handles} \Rightarrow \mathrm{v'}} \quad v \in \mathrm{Exn} \tag{84}$$

$$\frac{\mathrm{E} \vdash \mathrm{time} \Rightarrow \mathrm{t} \qquad \mathit{timecost}\ (\ \mathrm{E}, \mathrm{matches\ vs}\ ) \ \ge \ \mathrm{t} \qquad \mathrm{E}, \langle\ \mathrm{Timeout},\ \langle\,\rangle\ \rangle \models \mathrm{handle} \Rightarrow \mathrm{v}}{\mathrm{E}, \mathrm{vs} \vdash [\texttt{loop}]\ \mathrm{matches}\ \texttt{timeout}\ \mathrm{time}\ \texttt{handle}\ \mathrm{handles} \Rightarrow \mathrm{v}} \tag{85}$$

C.5 Dynamic Semantics: Expressions

$$\boxed{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}$$

The first few rules handle the semantics for simple expressions, including variables, basic values, nullary constructors, characters, and strings.

$$\frac{\mathrm{E}\ (\mathrm{var}) \ = \ \mathrm{v}}{\mathrm{E} \vdash \mathrm{var} \Rightarrow \mathrm{v}} \tag{86}$$

$$\frac{\mathrm{E}_0\ (\mathrm{b}) \ = \ \mathrm{exp} \qquad \mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \mathrm{b} \Rightarrow \mathrm{v}} \tag{87}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{con} \Rightarrow \mathrm{con}\ \langle\rangle} \tag{88}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{char} \Rightarrow \mathrm{char}} \tag{89}$$

$$\frac{\mathit{sval}\ (\mathrm{string}) \ = \ \mathrm{v}}{\mathrm{E} \vdash \mathrm{string} \Rightarrow \mathrm{v}} \tag{90}$$

The next rule defines the semantics of function applications as the application of the body of the function to a tuple of the arguments. There is no semantics of partial application.

$$\frac{\mathrm{E}\ (\mathrm{var}) \ = \ \mathrm{matches} \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \mathrm{v}_i \qquad \mathrm{E}, \langle\ \mathrm{v}_1, \ldots, \mathrm{v}_n\ \rangle \vdash \mathrm{matches} \Rightarrow \mathrm{v'}}{\mathrm{E} \vdash \mathrm{var}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \mathrm{v'}} \tag{91}$$

Rules 92–93 deal with constructors by interpreting their arguments as a tuple. If the value of the tuple is an exception, then this is the value of the expression; otherwise the value of the expression is constructed as the combination of the constructor and the semantic tuple of arguments.

$$\frac{\mathrm{E} \vdash (\mathrm{exp}_1 \ldots, \mathrm{exp}_n) \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \mathrm{con}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \mathrm{con}\ \mathrm{v}}\ \mathrm{v} \notin \mathrm{Exn} \tag{92}$$

$$\frac{\mathrm{E} \vdash (\mathrm{exp}_1 \ldots, \mathrm{exp}_n) \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \mathrm{con}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \mathrm{v}}\ \mathrm{v} \in \mathrm{Exn} \tag{93}$$

The next set of rules define the semantics for primitive constructors, including lists, tuples and vectors. The semantics of non-empty lists is given in terms of that for the constructors `(:)` and `Nil`, while that for non-empty vectors is given in terms of that for tuples.

$$\frac{\mathrm{E} \vdash (:)\ \mathrm{exp}_1\ (\ldots((:)\ \mathrm{exp}_n\ [\,])\ldots) \Rightarrow \mathrm{v}}{\mathrm{E} \vdash [\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ] \Rightarrow \mathrm{v}}\ \mathrm{n} \geq 1 \tag{94}$$

$$\frac{}{\mathrm{E} \vdash [\,] \Rightarrow \texttt{Nil}\ \langle\ \rangle} \tag{95}$$

$$\frac{}{\mathrm{E} \vdash () \Rightarrow \langle\ \rangle} \tag{96}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \mathrm{v}_i \qquad \vdash \langle\ \mathrm{v}_1,\ \ldots,\ \mathrm{v}_n\ \rangle \Rightarrow \mathrm{v'} \qquad \mathrm{v'} \notin \mathrm{Exn}}{\mathrm{E} \vdash (\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ) \Rightarrow \langle\ \mathrm{v}_1,\ \ldots,\ \mathrm{v}_n\ \rangle} \tag{97}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \mathrm{v}_i \qquad \vdash \langle\ \mathrm{v}_1,\ \ldots,\ \mathrm{v}_n\ \rangle \Rightarrow \mathrm{v'} \qquad \mathrm{v'} \in \mathrm{Exn}}{\mathrm{E} \vdash (\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ) \Rightarrow \mathrm{v'}} \tag{98}$$

$$\frac{\mathrm{E} \vdash (\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ) \Rightarrow \mathrm{v}}{\mathrm{E} \vdash <\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ > \Rightarrow \langle\ v_1,\ \ldots,\ v_n\ \rangle} \tag{99}$$

The semantics of case-expressions is defined by matching the value of the expression against the match. Note that the semantics for conditional expressions (rule 101) is defined in terms of the semantics for case-expressions (rule 100).

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v} \qquad \mathrm{E}, \mathrm{v} \vdash \mathrm{match} \Rightarrow \mathrm{v'}}{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}\ \texttt{of}\ \mathrm{match} \Rightarrow \mathrm{v'}} \tag{100}$$

$$\frac{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}_1\ \texttt{of}\ \{\ \mathrm{True}\ \rightarrow\ \mathrm{exp}_2 \mid \mathrm{False}\ \rightarrow\ \mathrm{exp}_3\ \} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \texttt{if}\ \mathrm{exp}_1\ \texttt{then}\ \mathrm{exp}_2\ \texttt{else}\ \mathrm{exp}_3 \Rightarrow \mathrm{v}} \tag{101}$$

Let-expressions have a simple semantics.

$$\frac{\mathrm{E} \vdash \mathrm{decls} \Rightarrow \mathrm{E'} \qquad \mathrm{E}\ \overset{\rightarrow}{\oplus}\ \mathrm{E'} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \texttt{let}\ \mathrm{decls}\ \texttt{in}\ \mathrm{exp} \Rightarrow \mathrm{v}} \tag{102}$$

Type signatures have no dynamic component.

$$\frac{\mathrm{E} \vdash \mathrm{exp} \;\Rightarrow\; \mathrm{v}}{\mathrm{E} \vdash \mathrm{exp} :: \mathrm{type} \;\Rightarrow\; \mathrm{v}} \tag{103}$$

The semantics of type coercion is defined in terms of an auxiliary *coerce* function that implements the semantics of coercion defined earlier. This function is still to be specified.

$$\frac{\mathrm{E} \vdash \mathrm{exp'} \;\Rightarrow\; \mathrm{v} \qquad \mathit{coerce}(\mathrm{v}, \mathrm{type}) \;=\; \mathrm{v'}}{\mathrm{E} \vdash \mathrm{exp}\ \texttt{as}\ \mathrm{type} \;\Rightarrow\; \mathrm{v'}} \tag{104}$$

Raising an exception simply involves returning it as the value of the expression.

$$\frac{\mathrm{E} \vdash \mathrm{exp} \;\Rightarrow\; \mathrm{v}}{\mathrm{E} \vdash \texttt{raise}\ \mathrm{exnid}\ \mathrm{exp} \;\Rightarrow\; \langle\ \mathrm{exnid},\ \mathrm{v}\ \rangle} \tag{105}$$

The next two rules define the semantics of timeouts. If the cost of evaluating the expression (as given by function *timecost* is greater than the timeout, then the Timeout exception is raised, otherwise the value of the within-expression is the same as the encapsulated expression.

$$\frac{\mathrm{E} \vdash \mathrm{exp}_2 \;\Rightarrow\; \mathrm{t} \qquad \mathit{timecost}\ (\mathrm{E},\ \mathrm{exp}_1\ ) \;<\; \mathrm{t} \qquad \mathrm{E} \vdash \mathrm{exp}_1 \;\Rightarrow\; \mathrm{v}}{\mathrm{E} \vdash \mathrm{exp}_1\ \texttt{within}\ \mathrm{exp}_2 \;\Rightarrow\; \mathrm{v}} \tag{106}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}_2 \;\Rightarrow\; \mathrm{t} \qquad \mathit{timecost}\ (\mathrm{E},\ \mathrm{exp}_1\ ) \;\geq\; \mathrm{t}}{\mathrm{E} \vdash \mathrm{exp}_1\ \texttt{within}\ \mathrm{exp}_2 \;\Rightarrow\; \langle\ \mathrm{Timeout},\ \langle\ \rangle\ \rangle} \tag{107}$$

The next expression rule defines the semantics of bracketed expressions in terms of the enclosed expression.

$$\frac{\mathrm{E} \vdash \mathrm{exp} \;\Rightarrow\; \mathrm{v}}{\mathrm{E} \vdash (\ \mathrm{exp}\ ) \;\Rightarrow\; \mathrm{v}} \tag{108}$$

Rules 109–114 extract exceptions from constructed values such as lists or tuples. The rules are applied to a value that is being matched in order to ensure that any exception that is embedded within the matched value is raised as a result of a match. If there are multiple exceptions, then the rightmost-outermost is returned. – the first such exception working from right-to-left is used the value of the constructed item. If there is no exception, this is signalled by the value $\langle\rangle$.

$$\boxed{\vdash \mathrm{v} \Rightarrow \mathrm{v}}$$

$$\frac{}{\vdash \mathrm{v} \Rightarrow \mathrm{v}} \quad \mathrm{v} \in \mathrm{Exn} \tag{109}$$

$$\frac{}{\vdash \langle\, \mathrm{v}_1 \ldots, \mathrm{v}_n \,\rangle \Rightarrow \mathrm{v}_n} \quad \mathrm{v}_n \in \mathrm{Exn} \tag{110}$$

$$\frac{\vdash \langle\, \mathrm{v}_1,\ \ldots,\ \mathrm{v}_{n-1} \,\rangle \Rightarrow \mathrm{v'}}{\vdash \langle\, \mathrm{v}_1,\ \ldots,\ \mathrm{v}_n \,\rangle \Rightarrow \mathrm{v'}} \quad \mathrm{v}_n \notin \mathrm{Exn} \tag{111}$$

$$\frac{\vdash \langle\, \mathrm{v}_1,\ \ldots,\ \mathrm{v}_{n-1} \,\rangle \Rightarrow \mathrm{v'}}{\vdash \mathrm{con}\ \mathrm{v}_1\ \ldots \mathrm{v}_n \Rightarrow \mathrm{v'}} \quad \mathrm{v}_n \notin \mathrm{Exn} \tag{112}$$

$$\frac{}{\vdash \langle\rangle \Rightarrow \langle\rangle} \tag{113}$$

$$\frac{}{\vdash \mathrm{v} \Rightarrow \langle\rangle} \quad \mathrm{v} \in \mathrm{BasVal} \tag{114}$$

The final expression rules are used in constructing matches for case-expressions and function applications. If the expression to be matched is an exception, then the result of the match is an exception; otherwise the matching rules defined below are used.

$$\boxed{\mathrm{E}, \mathrm{v} \vdash \mathrm{match} \Rightarrow \mathrm{v}}$$

$$\frac{\vdash \mathrm{v} \Rightarrow \mathrm{v'}}{\mathrm{E}, \mathrm{v} \vdash \mathrm{match} \Rightarrow \mathrm{v'}} \quad \mathrm{v'} \in \mathrm{Exn} \tag{115}$$

$$\frac{\vdash \mathrm{v} \Rightarrow \langle\rangle \qquad \mathrm{E}, \mathrm{v} \models \mathrm{match} \Rightarrow \mathrm{v'}}{\mathrm{E}, \mathrm{v} \vdash \mathrm{match} \Rightarrow \mathrm{v'}} \tag{116}$$

C.6 Dynamic Semantics: Matches

For clarity, we use a different kind of turnstile ($\models$) for match inference rules. E,v $\models$ e $\Rightarrow$ v' defines the meaning of *match* with respect to a single matched value *e*. The semantics for definitions and applications ensures that matches are curried appropriately.

The semantics for pattern-matching is derived from that presented in the Haskell report for **case** expressions (where the semantics was defined as a translation into a Haskell kernel). This gives a less direct semantics than that of, e.g., Standard ML.

Rules (117–118) define sequences of matches. The first rule applies when the first match in a sequence succeeds, the second when it fails. Failure of the last match in a sequence is as defined by the specific case below, e.g. in the rule for non-matching constructors (Rule 125). Since Hume requires matches to be complete, it should not be possible for this to happen, however.

$$\boxed{\mathrm{E,v} \models \mathrm{match} \Rightarrow \mathrm{v}/FAIL}$$

$$\frac{\mathrm{E,v} \models \{\ \mathrm{match}\ \} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\ \mathrm{match} \mid \mathrm{matches}\ \} \Rightarrow \mathrm{v'}} \tag{117}$$

$$\frac{\mathrm{E,v} \models \{\ \mathrm{match}\ \} \Rightarrow FAIL \qquad \mathrm{E,v} \models \{\ \mathrm{matches}\ \} \Rightarrow \mathrm{v}}{\mathrm{E,v} \models \{\ \mathrm{match} \mid \mathrm{matches}\ \} \Rightarrow \mathrm{v}} \tag{118}$$

Rule (119) simplifies multi-argument matches to single-argument matches.

$$\frac{\begin{array}{c}\forall i.\ 0 < i \leq m,\ \ \mathrm{var}_i \notin (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_{ij} \cup \mathrm{fv}(\mathrm{exp}_i))) \\[1em] \mathrm{E,v} \models \left\{ \begin{array}{l} (\mathrm{var}_1, \ldots, \mathrm{var}_n) \ \rightarrow \\ \quad \texttt{case}\ (\mathrm{var}_1, \ldots, \mathrm{var}_n)\ \texttt{of} \\ \qquad \{\ \ (\mathrm{pat}_{11}, \ldots, \mathrm{pat}_{1n}) \ \rightarrow \ \mathrm{exp}_1 \\ \qquad\quad \mid \ldots \\ \qquad\quad \mid (\mathrm{pat}_{m1}, \ldots, \mathrm{pat}_{mn}) \ \rightarrow \ \mathrm{exp}_m\ \} \end{array} \right\} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{\ \mathrm{pat}_{11} \ldots \mathrm{pat}_{1n} \ \rightarrow \ \mathrm{exp}_1 \mid \ldots \mid \mathrm{pat}_{m1} \ldots \mathrm{pat}_{mn} \ \rightarrow \ \mathrm{exp}_m\ \} \Rightarrow \mathrm{v'}}\ m \geq 1, n \geq 2 \tag{119}$$

Rule (120) simplifies matches into matches of the form { pat $\rightarrow$ exp | var $\rightarrow$ exp' }.

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n,\ \ \mathrm{var}_i \notin (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_j) \cup \mathrm{fv}(\mathrm{exp}_i)) \\[2ex] \mathrm{E,v} \models \left\{ \begin{array}{l} \mathrm{pat}_1 \ \to\ \mathrm{exp}_1 \\ \mid \mathrm{var}_1 \ \to\ \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{ \\ \quad \mathrm{pat}_2 \ \to\ \mathrm{exp}_2 \\ \quad \mid \mathrm{var}_2 \ \to\ \texttt{case}\ \mathrm{var}_2\ \texttt{of}\ \{ \\ \qquad \ldots \\ \qquad \mid \mathrm{var}_{n-1} \ \to\ \texttt{case}\ \mathrm{var}_{n-1}\ \texttt{of}\ \{\ \mathrm{pat}_n \ \to\ \mathrm{exp}_n\ \} \ldots\ \}\ \} \end{array} \right\} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{\ \mathrm{pat}_1 \ \to\ \mathrm{exp}_1 \mid \ldots \mid \mathrm{pat}_n \ \to\ \mathrm{exp}_n\ \} \Rightarrow \mathrm{v'}}\ n \geq 1 \tag{120}$$

Rules (121)–(122) define the semantics of wildcard and variable matches.

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\ \_ \ \to\ \mathrm{exp}\ \} \Rightarrow \mathrm{v'}} \tag{121}$$

$$\frac{\mathrm{E} \stackrel{\rightarrow}{\oplus} \{\ \mathrm{var} \ \mapsto\ \mathrm{v}\ \} \vdash \mathrm{exp} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\ \mathrm{var} \ \to\ \mathrm{exp}\ \} \Rightarrow \mathrm{v'}} \tag{122}$$

Rules (123)–(127) define the semantics of matches against constructor patterns. Rules (123) and (127) are simplification rules, simplifying general constructor matches and tuple matches, respectively; the remaining rules define the matching semantics. The simplification rules are used to simplify deep pattern matches (such as [1,2]) into single-level matches.

$$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n,\ \ \mathrm{var}_i \ \notin\ (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_j) \cup \mathrm{fv}(\mathrm{exp})) \\[2ex] \mathrm{E,v} \models \left\{ \begin{array}{l} \mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n \ \to \\ \quad \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{\ \mathrm{pat}_1 \ \to\ \ldots \\ \quad \texttt{case}\ \mathrm{var}_n\ \texttt{of}\ \{\ \mathrm{pat}_n \ \to\ \mathrm{exp}\ \} \ldots\ \} \end{array} \right\} \Rightarrow \mathrm{v'}\end{array}}{\mathrm{E,v} \models \{\ \mathrm{con}\ \mathrm{pat}_1 \ldots \mathrm{pat}_n \ \to\ \mathrm{exp}\ \} \Rightarrow \mathrm{v'}} \tag{123}$$

$$\frac{\mathrm{v} \ =\ \mathrm{con}\ < \mathrm{v}_1, \ldots, \mathrm{v}_n > \qquad \mathrm{E} \stackrel{\rightarrow}{\oplus} \{\ \forall i.\ 1 \leq i \leq n,\ \ \mathrm{var}_i \ \mapsto\ \mathrm{v}_i\ \} \vdash \mathrm{exp} \Rightarrow \mathrm{v'}}{\mathrm{E,v} \models \{\ \mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n \ \to\ \mathrm{exp}\ \} \Rightarrow \mathrm{v'}} \tag{124}$$

$$\frac{\mathrm{v} \ \neq\ \mathrm{con}\ \langle\ \mathrm{v}_1,\ \ldots,\ \mathrm{v}_n\ \rangle}{\mathrm{E,v} \models \{\ \mathrm{con}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n \ \to\ \mathrm{exp}\ \} \Rightarrow FAIL} \tag{125}$$

$$\frac{\mathrm{v} \ = \ <> \qquad \mathrm{E} \vdash \mathrm{exp} \ \Rightarrow \ \mathrm{v'}}{\mathrm{E}, \mathrm{v} \ \models \ \{ \ () \ \rightarrow \ \mathrm{exp} \ \} \ \Rightarrow \ \mathrm{v'}} \tag{126}$$

$$\frac{\begin{array}{c} \forall i.\ 0 < i \leq n,\ \mathrm{var}_i \notin \ (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_j) \cup \mathrm{fv}(\mathrm{exp})) \\ \mathrm{E}, \mathrm{v} \ \models \left\{ \begin{array}{l} (\mathrm{var}_1, \ldots, \mathrm{var}_n) \ \rightarrow \\ \quad \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{\ \mathrm{pat}_1 \ \rightarrow \ \ldots \\ \quad \texttt{case}\ \mathrm{var}_n\ \texttt{of}\ \{\ \mathrm{pat}_n \ \rightarrow \ \mathrm{exp}\ \} \ldots \ \} \end{array} \right\} \Rightarrow \ \mathrm{v'} \end{array}}{\mathrm{E}, \mathrm{v} \ \models \ \{ \ (\mathrm{pat}_1, \ldots, \mathrm{pat}_n) \ \rightarrow \ \mathrm{exp} \ \} \ \Rightarrow \ \mathrm{v'}} \tag{127}$$

$$\frac{\mathrm{v} \ = \ \langle \ \mathrm{v}_1, \ \ldots, \ \mathrm{v}_n \ \rangle \qquad \mathrm{E} \stackrel{\rightarrow}{\oplus} \bigcup_{i=1}^{n} \{ \ \mathrm{var}_i \ \mapsto \ \mathrm{v}_i \ \} \vdash \mathrm{exp} \ \Rightarrow \ \mathrm{v'}}{\mathrm{E}, \mathrm{v} \ \models \ \{ \ (\ \mathrm{var}_1, \ \ldots, \ \mathrm{var}_n \ ) \ \rightarrow \ \mathrm{exp} \ \} \ \Rightarrow \ \mathrm{v'}} \tag{128}$$

C.6.1 Exception Handler Matches

Rules 129–130 match against sequences of exception handlers.

$$\frac{\mathrm{E}, \mathrm{v} \ \models \ \mathrm{handler} \ \Rightarrow \ \langle \ \rangle \qquad \mathrm{E}, \mathrm{v} \ \models \ \mathrm{handlers} \ \Rightarrow \ \mathrm{v'}}{\mathrm{E}, \mathrm{v} \ \models \ \mathrm{handler} \mid \mathrm{handlers} \ \Rightarrow \ \mathrm{v'}} \tag{129}$$

$$\frac{\mathrm{E}, \mathrm{v} \ \models \ \mathrm{handler} \ \Rightarrow \ \mathrm{v'}}{\mathrm{E}, \mathrm{v} \ \models \ \mathrm{handler} \mid \mathrm{handlers} \ \Rightarrow \ \mathrm{v'}} \tag{130}$$

Finally, rules 131–132 handle matches against individual exceptions, either success or failure.

$$\frac{\mathrm{v} \ = \ \langle \ \mathrm{exnid}, \ \mathrm{v'} \ \rangle \qquad \mathrm{E}, \mathrm{v'} \ \models \ \mathrm{pat} \ \Rightarrow \ \mathrm{v''}}{\mathrm{E}, \mathrm{v} \ \models \ \mathrm{exnid}\ \mathrm{pat} \ \rightarrow \ \mathrm{exp} \ \Rightarrow \ \mathrm{v''}} \tag{131}$$

$$\frac{\mathrm{v} \ = \ \langle \ \mathrm{exnid'}, \ \mathrm{v'} \ \rangle}{\mathrm{E}, \mathrm{v} \ \models \ \mathrm{exnid}\ \mathrm{pat} \ \rightarrow \ \mathrm{exp} \ \Rightarrow \ \langle \ \rangle} \tag{132}$$

C.7 Dynamic Semantics: The Initial Environment

The initial environment comprises definitions for all functions and constructors defined in the module *Prelude*. These values must be available in all Hume programs. The meanings of other *Prelude* functions is defined by reference to Appendix D, which provides a source language definition. We assume that the meaning of basic operations (such as addition on numbers) is obvious. To define this formally would be tedious in the extreme.

The initial environment contains the following functions (*BasVal*)

$$
\begin{array}{lll}
\text{PrimPlusInt} \mapsto & (a, b) \texttt{ -> } a + b & + \text{ is fixed-precision integer addition} \\
\text{PrimMulInt} \mapsto & (a, b) \texttt{ -> } a \times b & \times \text{ is fixed-precision integer multiplication} \\
\ldots & &
\end{array}
$$

plus the standard constructors (*BasCon*):

0, 1, ..., 0.0, 0.1, ..., *True*, *False*, 'a', ..., (:), *Nil*

The characters correspond to those defined by the ASCII character set. The mapping from syntactic variables to semantic constructors is the obvious one, that is, E_0 (SetEnv) $=$ SetEnv ....

Appendix D

Standard Prelude

Summary of Standard Hume Functions and Operators

Int

```
+, -, *, div :: Int -> Int -> Int
=, /=, <=, <, >, >= :: Int -> Int -> Bool
```

Nat

```
+, -, *, div :: Int -> Int -> Int
=, /=, <=, <, >, >= :: Int -> Int -> Bool
```

Word

```
+, - :: Word -> Nat -> Word
=, /=, <=, <, >, >= :: Word -> Word -> Bool
```

Vectors

```
range :: vector
```

Tuples

```
=, /=, <=, <, >, >= :: (a1, .., an) -> (a1, ..., an) -> Bool
```

Lists

```
++ :: [a] -> [a] -> [a]
member :: [a] -> a -> Bool
=, /=, <=, <, >, >= :: [a] -> [a] -> Bool
```

Bibliography

[1] J.-R. Abrial, E. Börger, and H. Langmaack (Eds.), "Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control", Springer-Verlag, LNCS 1165, October 1996.

[2] W. Ackermann, *Solvable Cases of the Decision Problem*, North Holland, 1954.

[3] J.G.P. Barnes, *High Integrity Ada: the Spark Approach*, Addison-Wesley, 1997.

[4] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of the IEEE*, **79**(9), Sept. 1991, pp. 1270-1282.

[5] A. Benveniste and P.L. Guernic, "Synchronous Programming with Events and Relations: the Signal Language and its Semantics", *Science of Computer Programming*, **16**, 1991, pp. 103–149.

[6] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation", *Science of Computer Programming*, **19**(2), 1992, pp. 87–152.

[7] G. Blair, L. Blair, H. Bowman and A. Chetwynd, *Formal Specification of Distributed Multimedia Systems*, UCL Press, 1998.

[8] F. Boussinot and R. de Simone, "The Esterel Language", *Proceedings of the IEEE*, **79**(9), Sept. 1991, pp. 1293–1304.

[9] R. Burstall, "Inductively Defined Functions in Functional Programming Languages", LFCS, Department of Computer Science, University of Edinburgh, ECS-LFCS-87-25, April, 1987.

[10] P. Caspi, D. Pilaud, N. Halbwachs and J. Place, "Lustre: a Declarative Language for Programming Synchronous Systems", *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL '87)*, München, Germany, 1987.

[11] C. Grelck and S.-B. Scholz, "HPF vs. SAC — a Case Study", *Proc. EuroPar 2000*, Munich, Germany, Springer Verlag, LNCS, September 2000.

[12] K. Hammond, H.-W. Loidl and A.S. Partridge, "Improving Granularity for Parallel Functional Programs: a Graphical Winnowing System for Haskell", *Proc. 1995 Conf. on High Performance Functional Computing (HPFC '95)*, Denver, Co., Apr. 1995, pp. 208–221.

[13] K. Hammond and G.J. Michaelson (Eds.), *Research Directions in Parallel Functional Programming*, Springer-Verlag, November 1999.

[14] K. Hammond, "Hume: a Bounded Time Concurrent Language", *Proc. IEEE International Conf. on Electronic Control Systems (ICECS '2K)*, Kaslik, Lebanon, December 2000.

[15] K. Hammond and Greg Michaelson, "Wiring your Hume", *Internal Report*, University of St Andrews and Heriot-Watt University, April 2001, `http://www-fp.dcs.st-and.ac.uk/hume/docs/wiring.ps`.

[16] D. Harel, "Statecharts: a Visual Approach to Complex Systems", *Science of Computer Programming*, **8**(3), 1987, pp. 231–274.

[17] D. Harel, O. Lichtenstein and A. Pnueli, "Explicit Clock Temporal Logic", *Proc. 5th IEEE Symposium on Logic in Computer Science*, 1990, pp. 402–413.

[18] I. Holyer and E. Spiliopoulou, "Concurrent Monadic Interfacing", *Proc. 10th. Intl. Workshop on Implementation of Functional Programming (IFL '98)*, Springer-Verlag, LNCS 1595, 1998, pp. 73–89.

[19] R.J.M. Hughes, L. Pareto, and A. Sabry, "Proving the Correctness of Reactive Systems using Sized Types", *Proc. 1996 ACM Symposium on Principles of Programming Languages (POPL '96)*, St Petersburg, Fl., Jan. 1996.

[20] R.J.M. Hughes and L. Pareto, "Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming", *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, 1999.

[21] International Organisation for Standardisation, "ISO: Information Processing Systems — Open Systems Interconnection — LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour", ISO 8807, Geneva, August 1988.

[22] International Organisation for Standardisation, "ISO: Information Processing Systems — Open Systems Interconnection — Estelle, A Formal Description Technique based on an Extended State Transition Model", ISO 9074, Geneva, May 1989.

[23] International Telecommunication Union, *[Z.100] Recommendation Z.100 (11/99) – Specification and description language (SDL)*, 1999.

[24] F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Trans. on Software Eng.*, **12**(9), 1986, pp. 890–904.

[25] F. Kamareddine and F. Monin, "On Formalised Proofs of Termination of Recursive Functions", *PPDP'99*, LNCS 1702, pp 29-46, 1999.

[26] F. Kamareddine and F. Monin, "On Automating Inductive and Non-inductive Termination Methods", *ASIAN'99*, LNCS 1742, pp. 177-189, 1999.

[27] P.H.J. Kelly and F.A. Taylor, "Coordination Languages", In [13].

[28] R. Koymans, "Specifying Real-Time Properties with Metric Temporal Logic", *Real-Time Systems*, **2**, 1990, pp. 255-299.

[29] N.G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[30] H-W. Loidl, *Granularity in Large-Scale Parallel Functional Programming*, PhD Thesis, Department of Computer Science, University of Glasgow, 1998.

[31] H-W. Loidl, R. Morgan, S.L. Peyton Jones, R. Garagliano, P.W. Trinder and C. Cooper, "Parallelising a Large Functional Program. Or Keeping Lolita Busy", *Proc. 1997 International Workshop on Implementing Functional Languages (IFL '97)*, St Andrews, Scotland, Sept. 1997, Springer-Verlag LNCS 1467.

[32] D. Le Métayer, "ACE: An Automatic Complexity Evaluator", *ACM TOPLAS*, **10**(2), April 1988.

[33] H.-W. Loidl, A.J. Rebón Portillo and Kevin Hammond, "A Sized Time System for a Parallel Functional Language (Revised)", *Draft Proc. 2nd Scottish Functional Programming Workshop*, St Andrews, July 2000.

[34] M. Lücker and T. Noll, "The Implementation of the TRUTH Model Checker in Haskell", *Draft Proc. International Workshop on Implementation of Functional Programming (IFL 2000)*, Aachen, Germany, RWTH Fachgruppe Informatik Research Report 00-7, M. Mohnen and P. Koopman (Eds.), September 2000, pp. 363-380.

[35] G.J. Michaelson, "Interpreter prototypes from formal language definitions", PhD thesis, Dept of Computing and Electrical Engineering, Heriot-Watt University, 1993.

[36] G.J. Michaelson, "Constraints on Recursion in the Hume Expression Language", *Draft Proc. International Workshop on Implementation of Functional Programming (IFL 2000)*, Aachen, Germany, RWTH Fachgruppe Informatik Research Report 00-7, M. Mohnen and P. Koopman (Eds.), September 2000, pp. 231–246.

[37] G.J. Michaelson and K.Hammond, "The Hume Language Definition and Report, Version 0.1", Heriot-Watt University and University of St Andrews, July 2000.

[38] G.J. Michaelson and N. Scaife , "Prototyping a parallel vision system in Standard ML", *Journal of Functional Programming*, special issue on Applications of Functional Programming, **5**(3), pp. 345–382, July 1995

[39] G. Michaelson, N. Scaife, P. Bristow and P. King, "Nested algorithmic skeletons from higher order functions", *Parallel Algorithms and Applications*, special issue on High Level Models and Languages for Parallel Processing, to appear, September 2000

[40] C. Miguel, *Extended LOTOS Definition*, OSI 95 (Esprit Project 5341), Report OSI95/DIT/B5/8/TR/R/V0, Depto. Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, Madrid, Spain.

[41] A.J.R.G. Milner, J. Parrow and D. Walker, "A Calculus of Mobile Processes, Parts I and II", *Information and Computation*, **100**(1), 1992, pp. 1–77.

[42] A.J.R.G. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, *MIT Press*, 1997.

[43] R. Mirani, and P. Hudak, "First-Class Schedules and Virtual Maps", *Proc. FPCA '95 — Functional Prog. Langs. and Computer Architecture*, La Jolla, CA, June, 1995, pp. 78–85.

[44] J.W. Peterson, K. Hammond (eds.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, E. Meijer, S.L. Peyton Jones, A. Reid and P.L. Wadler, "Report on the Programming Language Haskell: a Non-Strict Purely Functional Language, Version 1.4", April 1997.

[45] *Report on the Non-Strict Functional Language, Haskell* S.L. Peyton Jones (ed.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, K. Hammond, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, E. Meijer, J.C. Peterson, A. Reid, and P.L. Wadler, Yale University, 1999.

[46] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain and P.L. Wadler, "The Glasgow Haskell Compiler: a Technical Overview", *Proc. JFIT '93*, Keele, March 1993.

[47] R. Peter, *"Recursive Functions"*, Academic Press, 1967.

[48] A. Pnueli, "The Temporal Logic of Programs", *Proc. 18th. IEEE Symposium on Foundations of Computer Science*, Los Alamitos, CA, 1977, pp. 46-57.

[49] B. Randall, "Facing up to faults", Computer Journal, **43**(2), 2000, pp. 95–106.

[50] R. Rangaswami, "Compile-Time Cost Analysis for Parallel Programming", *Proc. EUROPAR '96*, Lyon, France, 1996.

[51] G.M. Reed and A.W. Roscoe, "A Timed Model for Communicating Sequential Processes", *Theoretical Computer Science*, **58**, 1988, pp. 249–261.

[52] B. Reistad and D.K. Gifford., "Static Dependent Costs for Estimating Execution Time", *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pp. 6578, Orlando, Fl., June 2729, June 1994.

[53] A.W. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall, 1998.

[54] D. Sands, "Complexity Analysis for a Lazy Higher-Order Language", *Proc. 1990 European Symposium on Programming (ESOP '90)*, Springer-Verlag LNCS 432, May 1990.

[55] D.B. Skillicorn, "Deriving Parallel Programs from Specifications using Cost Information", *Science of Computer Programming*, **20**(3), June 1993.

[56] S. Stepney, *High Integrity Compilation: a Case Study*, Prentice-Hall, 1993.

[57] Sun Microsystems, "EmbeddedJava Technical Overview", URL: `http://java.sun.com/products/embeddedjava/spec/1.0/eJavaTechnicalOverview.html`, 1998.

[58] M. Tofte and J.-P. Talpin, "Region-based Memory Management", *Information and Control*, **132**(2), 1997, pp. 109–176.

[59] P.W. Trinder, K. Hammond, H.-W. Loidl and S.L. Peyton Jones, "Algorithm + Strategy = Parallelism", *Journal of Functional Programming*, **8**(1), Jan. 1998, pp. 23–60.

[60] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge and S.L. Peyton Jones, "GUM: a Portable Parallel Implentatiion of Haskell", *Proc. 1996 ACM. Conf. on Programming Language Design and Implementation (PLDI '96)*, Philadelphia, May 1996, pp. 79–90.

[61] D.A. Turner, "Elementary Strong Functional Programming", *Proc. 1st. Symposium on Functional Programming Languages in Education — FPLE '95*, Springer-Verlag LNCS No. 1022, December 1995.

[62] W. Yi, "Real-Time Behaviour of Asynchronous Agents", *Proc. CONCUR '90*, Springer-Verlag, LNCS 458, 1990, pp. 502–520.