Automatic Amortised Resource Analysis for Hume

Notes for the 2009 Advanced Programming Languages Summer School,
Heriot-Watt University, Edinburgh, 25-28 August 2009

Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl
School of Computing Science, University of St Andrews, Scotland
<kh,jost,hwloidl@cs.st-andrews.ac.uk>

Pedro Vasconcelos
Universidade do Porto, Porto, Portugal
<pbv@dcc.fc.up.pt>

6th August 2009

Revision: 1.13

EU Framework VI FET-Open Project IST-510255 (**EmBounded**)

To our muses: Vicki, Natalie and Sofia.

Contents

List of Figures

Chapter 1

Introduction

Steffen Jost, Kevin Hammond and Hans-Wolfgang Loidl

Abstract

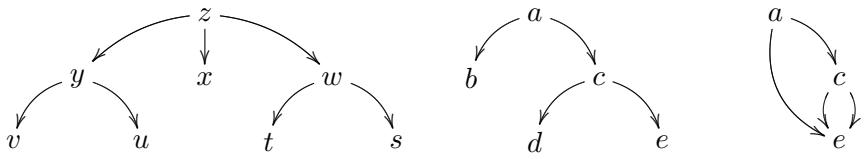
This chapter gives an overview of the amortised analysis approach that we have used for Hume. We introduce an analogy with carbon emissions, describe amortised analysis in general terms, outline how we will develop an automatic analysis and introduce the Hume programming language that will be used as the basis for undertaking our analysis.

1.1 Predicting Software "Emissions"

Programs often produce undesirable "emissions", such as littering the memory with garbage[1]. Our work is aimed at predicting limits on such emissions in advance of execution. "Emissions" here refer to any quantifiable resource that is used by the program. In these notes, we will focus on the key resources of worst-case execution time, heap allocations, and stack usage. Predicting emissions limits is clearly desirable in general, and can be vital in safety-critical, embedded systems.

Our method can be explained by analogy to an attempted countermeasure to global warming: some governments are attempting to reduce industrial pollution by issuing tradable carbon credits. The law then dictates that each CO_2 emission must be offset by expending an appropriate number of carbon credits. It follows that the total amount of emissions is *a priori* bounded by the number of carbon credits that have been previously issued by the authorities. Following this analogy, we will similarly issue credits for computer programs. The "emissions" of each program operation must then be *immediately* justified by spending a corresponding number of credits. The use of "carbon credits" for software analysis does, however, have several advantages over the political situation: i) we can *prove* that each and every emission that occurs is legitimate and that it has been properly paid for by spending credits; ii) we have zero bureaucratic overhead, since we use an efficient *compile-time* analysis, there need be no modifications whatever to the original program, and we therefore do not change actual execution costs; and iii) we provide an *automatic static analysis* that, when successful, provides a *guaranteed* upper bound on the number of credits that must be issued initially to ensure that a program can run to completion, rather than using a heuristic to determine the requirements. The amount of credits a program is allowed to spend is specified as part of its type. This allows the absolute number of credits to vary in relation to the actual input, as shown below.

Example: Tree Processing. Consider a tree-processing function *mill*, whose argument has been determined by our analysis to have type $\mathsf{tree}(\mathsf{Node}\langle 7\rangle \mid \mathsf{Leaf}\langle 0.5\rangle)$. Given this type, we can determine that processing the first tree below requires at most $23 = \lfloor 23.5 \rfloor$ credits: 7 credits per node and 0.5 credits for each leaf reference[2]; and that processing either of the other trees requires at most $\lfloor 15.5 \rfloor$ credits, regardless of aliasing.



In fact, the type given by our analysis allows us to easily determine an upper bound on the cost of *mill* for any input tree. For example, for a tree of 27 nodes and 75 leaves, we can compute the credit quota from the type as $7 \cdot 27 + 0.5 \cdot 75 = 226.5$, without needing to consider the actual node or leaf values. The crucial point is that while we are analysing *mill*, our analysis only needs to keep track of this single number. Indeed, the entire dynamic state of the program at any time during its execution *could* be abstracted into such a number, representing the total unspent credits at that point in its execution. Because the number of credits must always be non-negative, this then establishes an upper bound on the total future execution costs (time or space, etc.) of the program. Note that since this includes the cost incurred by all subsequent function calls, recursive or otherwise, it follows that our analysis will also deal with outsourced emissions.

[1] Some material in this section, including this analogy, previously appeared in [55]

[2] Note while only whole credits may be spent, fractional credits can be accumulated.

1.2 Automatic Analysis Approach for Resource Consumption

We aim to automate the process of determining a bound on the number of credits that are needed to enforce emissions limits. The automatic analysis that we will develop in Chapter 8 is a variation of the *amortised cost analysis* approach that was first described by Tarjan [143]. Amortised cost analysis is a manual technique, which works as follows: Using ingenuity, we devise a mapping from all possible machine memory states to a non-negative rational number, henceforth referred to as the *potential* of that state. The map must be constructed in such a way that the actual cost of each machine operation is amortised by the difference in the state potentials before and after the execution of each operation. For example, for heap space, an operation allocating n memory units always leads to states whose potential is decreased by n when compared to the state before that operation. Therefore the cost of each operation, including entire loops or recursive calls, becomes zero, and the overall execution cost is then equal to the potential of the initial state.

Devising a useful mapping from machine states to the number representing potential is, unfortunately, quite a difficult task in general. Okasaki notes in his excellent book [118] that

> *"As we have seen, amortized data structures are often tremendously effective in practice. Unfortunately, traditonal methods of amortization break in presence of persistence."*

Our *type-based* variant of amortised analysis solves both of these issues at once: We can automatically determine the abstraction through efficient linear programming techniques and deal with the persistent data structures that are commonly found in a functional setting by assigning potential on a per-reference basis rather than resorting to a lazy-evaluation strategy as Okasaki does. The price is that our method is currently limited to linear cost formulas, a restriction which is not inherent to amortised cost analysis. However, we believe that an efficient automatic analysis at the press of a button is a major advantage over a complex, cumbersome, error-prone manual analysis. Note that in practice, our method can be quite simple to implement: It does does not involve reference counting, we only need to determine the points at which aliases are introduced. Furthermore, the automatically inferred potential mappings always allow easy determination of the initial potential for large classes of inputs, and can thus be transformed to simple closed cost formulas.

1.3 Hume: A New Box-Based Notation for Embedded Systems

We apply our work in the context of the Hume language. Since late 2000, we have been exploring a new *cost-driven*, transformational approach to software construction from certified components which is highly suited to dynamic, reconfigurable embedded systems [47, 48, 49, 50, 105]. This approach builds on the modern *layered* programming language *Hume* [13, 52, 54, 56, 59, 60, 61, 62, 63, 64, 82, 106], whose strengths lie in the explicit separation of *coordination* and *control* concerns. Hume is based on autonomous *boxes* linked by *wires* and controlled by generalised transitions. Boxes and wires are defined in the finite state *coordination layer* with transitions defined in the purely functional *expression layer* through pattern matching and associated recursive actions. Both coordination and expression layers share a rich polymorphic type system, comparable to contemporary functional languages like Haskell [39] and Standard ML [107].

Finite State Automata. *Finite state automata* provide a basis for constructing simple state-changing systems. They may also be used to give a natural model of concurrency. Finite state automata comprise a set of linked states, with transitions showing the changes from one state to another based on the inputs that are received. Because pure finite state automata are so simple, there is a natural fit between finite state automata and hardware, and it is easy to show that such automata have bounded

time and space costs. Some low-level programming languages, such as Esterel [16] also use an essentially pure finite state approach, and mechanistic systems such as lexers and parsers are also commonly automaton-based. The primary deficiencies of finite state approaches are:

- there may be a huge number of states for even fairly simple programs;
- it may be necessary to decompose programming problems into very low-level abstractions;
- there are theoretical limitations on the classes of problems that can be solved using finite state automata.

While each state may be simple in itself, the explosion in the number of states means even small, simple programs can be too complex to understand easily. Moreover, it can be cumbersome to write simple functions and other operations as automata. Hume attempts to systematically address these objections as follows:

- finite state automata are used to structure concurrency only – computations are written using conventional programming notations;
- high-level programming notations are used to collapse complex sets into a few manageable automata;
- combining high-level programming notations with automata greatly extends the classes of problems that can be solved to cover all *computable* problems.

In Hume, programs are formed from concurrent *boxes*, which respond to inputs and produce outputs on one or more *wires*. Computations within boxes are described using normal high-level programming notations rather than as automata. While there is a broad analogy with the use of high-level *objects* as concurrent agents, and object-based designs can thus be exploited at the high-level, the analogy should not be stretched too far – boxes are much more structured than objects, in particular in restricting communication patterns, in relating inputs directly to outputs, and in providing a static rather than dynamic process network. This discipline allows us to automate testing, to demonstrate deadlock-freedom using an automatic analysis, and to enforce strong bounds on program costs.

Functional Programming. *Purely functional programming* provides a good basis for constructing software with excellent formal properties. Because functional programs are both *declarative* and *deterministic,* they are much easier to reason about using mathematically-derived techniques than either object-based or imperative approaches [75]. In fact, many advanced compiler optimisations work on an internal representation that is purely functional, and compilers can go to great lengths to isolate parts that are not purely functional, so that they can take advantage of these techniques.

However, to obtain these advantages:

a) programmers must be trained to exploit high-level functional abstractions, which some programmers find difficult;

b) there may be a poor match between program and machine implementation, making it difficult to construct software that must access low-level features; and

c) performance can be significantly worse than with the best imperative implementations (though performance may be better than, say, with C++ or even Fortran in some cases [131]).

Hume attempts to systematically address these objections as follows:

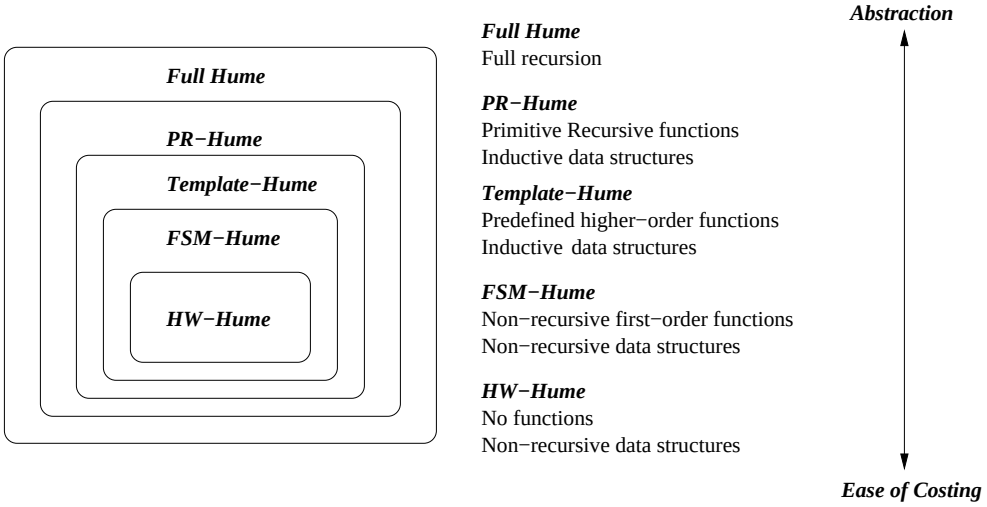


Figure 1.1: Hume Design Space

a) finite state automata are a natural way to decompose concurrent programs, and provide analogies to the higher levels of object-based program decomposition, without the structures of overly low-level objects;

b) state changes are made explicit through finite state automata, and explicit operating system interactions;

c) it is possible to provide a straightforward translation from Hume source to the corresponding machine code, whether direct or through an intermediate abstract machine; and

While we have not yet fully optimised our implementations, Hume has been designed to allow good compiler optimisations to be exploited. In particular, although the Hume model exposes explicit wires and boxes to the programmer, many of these may be compiled to simple memory accesses, or even compiled away. Currently, time performance of Hume programs *without* optimisation is roughly 10 times faster than that for Sun's embedded KVM Java Virtual Machine or about half the speed of optimised C++, while dynamic space usage is only a fraction of that required by either Java or C++, and is guaranteed to be bounded. For example, we have constructed a complete implementation for a Renesas M32C bare development board using less than 16KB RAM, including all run-time, operating system, user code and library support. The combination of finite state automata with functional programming therefore gives a powerful programming basis without sacrificing crucial low-level capabilities.

Levels of Expressivity. Hume offers programmers different programming *levels* where expressivity is balanced against accuracy of behavioural modelling (Figure 1.1). *Full-Hume* is a general purpose, Turing-complete language with undecidable correctness, termination and resource bounds. *PR-Hume* restricts Full-Hume expressions to primitive recursive constructs, enabling decidable termination and bounded resource prediction. *Template-Hume* further restricts expressions to higher-order functions with precise cost models, enabling stronger resource prediction. In *FSM-Hume*, types are restricted to those of fixed size and expressions to conditions over base operations, enabling highly accurate resource bounds. Finally, *HW-Hume* is a basic finite state language over tuples of bits, offering decidable correctness and termination, and exact resource analysis. However, rather than requiring programmers to choose a level from the outset, we have elaborated an iterative methodology based on cost-driven transformation. An initial Hume program, designed to meet its specification, is then analysed to establish resource bounds. Where established bounds are unacceptable, the offending program constructs are transformed, usually to lower levels, and the program is again analysed, with the cycle continuing until the required analytic precision is reached. Furthermore, this approach is applicable where Hume

programs are partially or fully built from unitary components of known bounds guaranteed by certificate, where components can be written in *any* certifiable language: whether Hume, C, or some other notation.

1.4 Structure of these Notes

In this document, we will develop all the theoretical underpinnings that are required to produce an amortised resource analysis for Hume. We consider Full Hume programs, but produce cost information only where a program has cost that is linear in the sizes of its input values. We will focus on worst-case execution time, since this involves significant care, both in determining correct cost points in the analysis, in developing a good operational semantics, and in obtaining detailed and accurate measurements. In Chapter 2, we will survey approaches to cost modelling and analysis, covering type-based systems, including recent sized type approaches as well as our previous work on amortised analysis. We will also consider approaches based on automatic complexity analysis, as well as recent work on worst-case execution time analysis. In Chapter 3, we describe the structure and operation of the Hume Abstract Machine. In Chapter 4, we describe how Hume programs can be compiled to Hume Abstract Machine (HAM) instructions, including a formal description of the compilation process. Chapter 5 develops a high-level cost model for Hume that is used in Chapter 6 to develop a resource-aware operational semantics for the Hume Abstract Machine. Chapter 7 develops the Schopenhauer and Core-Hume intermediate notations that will simplify the specification of our amortised analysis, and provides a generic framework for a type-based amortised analysis. Chapter 8 develops the amortised analysis for worst-case execution time. Finally, Chapter **??** evaluates our amortised analysis against a number of significant examples, for both worst-case execution time, stack-space and heap-space metrics.

Acknowledgements and Apologies

At this stage, this document represents a compilation of material from various sources. In particular much of the material was produced in the course of the EU Framework VI EmBounded project (`http://www.embounded.org`), extracts from which have been published in the form of various technical conference and journal papers, as shown on the project web site. We apologise in advance for any errors or omissions that have been introduced as a result of editing this document.

We would like to thank our sponsors from the European Union (EmBounded Project IST-510255), the UK's Engineering and Physical Sciences Research Council (EPSRC Project EP/C 001346 and EP/F 030657 (Islay)), Selex PLC, the Royal Society of Edinburgh and the British Council. We would like to thank our collaborators on the EmBounded project, including Greg Michaelson, Martin Hofmann, Jocelyn Sérot, Robert Pointon, Norman Scaife, Christian Ferdinand, Reinhold Heckmann for their contributions to numerous technical points. We would also like to thank Armelle Bonenfant and Christoph Herrmann for their hard work in determining time costs for the Renesas M32C/85U and for providing detailed time measurements for various programs. Finally, but not least, we would like to thank Hugo Simoes for his knowledged and detailed comments on our amortised analysis.

Chapter 2

Cost Modelling and Analysis

Pedro Vasconcelos, Kevin Hammond and Steffen Jost

Abstract

This chapter describes previous approaches to cost modelling and analysis. The material is largely taken from Vasconcelos' thesis. We first review some basic principles of two frameworks of program analysis, namely, *type and effect systems* and *abstract interpretation* that form the basis of our cost analysis. We then present a comparative review of the most relevant work on determining bounds for time or space usage.

2.1 Overview

Program analysis concerns the study of automatic techniques for obtaining predictive information about the dynamic behaviour of programs. The usual requirement is that program analysis should obtain *sound* information with respect to the program semantics, that is, obtain *approximations* that hold for all executions. This means that any approximation must be conservative, i.e. err by over-estimation of the dynamic behaviour.

The traditional motivation for program analysis is to gather information for enabling optimisations in compilers. More recently, program analysis has been applied for verifying that software respect both *safety properties* ("something bad will not happen") and *liveness properties* ("something good will happen") [119]. Applications of program analysis in this context include aiding detecting errors, validating software received from sub-contractors, allowing execution of foreign code in an untrusted environment and aiding in transformations of data formats (e.g. the Y2K problem).

A complete survey of program analysis is beyond the scope of this document; we refer the reader to the textbook of Nielson, Nielson, and Hankin [114] for a comprehensive presentation of the area. In the remaining of this chapter we will focus on the basic principles of the two approaches used in our work, namely, *type and effect systems* and *abstract interpretation*.

2.2 Type and effect systems

Type and effect systems are program analysis that extend types with annotations describing properties of values or evaluations [112]. Analyses based on effects were first introduced to control the combination of imperative features with functional languages [97, 98, 140]; in this setting, *effects* are abstract descriptions of impure side-effects occurring during evaluation, e.g. accesses to imperative references or input-output actions. Other uses of type and effects analysis include exception tracking, inferring region annotations [139, 146], analysing communication in concurrent systems [2] and predicting execution costs [38, 73, 127]; the latter will be reviewed in detail in Section 2.4 onwards.

2.2.1 A simply-typed language

For concreteness we will consider an analysis for tracking exceptions raised during evaluation of a simple applicative language; our presentation is based on [114]. The syntax of terms is the simply-typed lambda-calculus extended with constants, conditionals and exception raising and handling:

$$\begin{aligned} e \quad ::= \quad & c \mid x \mid \lambda x.\, e \mid (@e_1 e_2) \\ & \mid \mathsf{if} \quad \mathsf{then}\ e\ \mathsf{else}_{\,0}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{raise}\ \epsilon \mid \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 \end{aligned} \tag{2.2.1}$$

Exceptions are identified by tokens ϵ taken from some finite set; they can be raised by the evaluation of a raise expression and trapped by the expression '$\mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2$'; the latter evaluates to e_2 *unless* the exception ϵ is raised, in which case it evaluates to e_1. For simplicity we have omitted primitive operations and recursive definitions; extending the exception analysis for these is straightforward.

The semantics of expressions is given in Table 2.1 by a call-by-value big-step evaluation relation $\vdash e \longrightarrow v$ meaning that expression e evaluates to the value v; values are a proper subset of expressions, namely: constants, (closed) lambda-abstractions or raised exceptions.

$$v \quad ::= \quad c \mid \lambda x.\, e \mid \mathsf{raise}\ \epsilon \tag{2.2.2}$$

We use the notation $e[x \mapsto e']$ to substitute a variable x for e' in an expression e. It will be the case that e' is a closed expression (i.e. without free variables) whenever we use substitutions so that we need not concern with the possibility of variable capture.

$$\vdash c \longrightarrow c$$

$$\vdash \lambda x.\, e \longrightarrow \lambda x.\, e$$

$$\vdash \mathsf{raise}\ \epsilon \longrightarrow \mathsf{raise}\ \epsilon$$

$$\frac{\vdash e_1 \longrightarrow \mathsf{raise}\ \epsilon}{\vdash (@e_1 e_2) \longrightarrow \mathsf{raise}\ \epsilon}$$

$$\frac{\vdash e_1 \longrightarrow \lambda x.\, e' \qquad \vdash e_2 \longrightarrow \mathsf{raise}\ \epsilon}{\vdash (@e_1 e_2) \longrightarrow \mathsf{raise}\ \epsilon}$$

$$\frac{\vdash e_1 \longrightarrow \lambda x.\, e' \qquad \vdash e_2 \longrightarrow v_2 \qquad \vdash e'[x \mapsto v_2] \longrightarrow v}{\vdash (@e_1 e_2) \longrightarrow v} \quad v_2 \neq \mathsf{raise}\ \epsilon$$

$$\frac{\vdash e_0 \longrightarrow \mathsf{raise}\ \epsilon}{\vdash \mathsf{if}\ \ \mathsf{then}\ e\ \mathsf{else}_{\,0}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \mathsf{raise}\ \epsilon}$$

$$\frac{\vdash e_0 \longrightarrow \mathsf{true} \qquad \vdash e_1 \longrightarrow v_1}{\vdash \mathsf{if}\ \ \mathsf{then}\ e\ \mathsf{else}_{\,0}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow v_1}$$

$$\frac{\vdash e_0 \longrightarrow \mathsf{false} \qquad \vdash e_2 \longrightarrow v_2}{\vdash \mathsf{if}\ \ \mathsf{then}\ e\ \mathsf{else}_{\,0}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow v_2}$$

$$\frac{\vdash e_2 \longrightarrow \mathsf{raise}\ \epsilon \qquad \vdash e_1 \longrightarrow v}{\vdash \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 \longrightarrow v}$$

$$\frac{\vdash e_2 \longrightarrow v}{\vdash \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 \longrightarrow v} \quad v \neq \mathsf{raise}\ \epsilon$$

Table 2.1: Big-step semantics for exceptions.

$$\Gamma \vdash c : \tau_c$$

$$\Gamma \cup \{x : \tau\} \vdash x : \tau$$

$$\Gamma \vdash \mathsf{raise}\ \epsilon : \tau$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\quad \mathsf{then}\ e\ \mathsf{else}_{\,1}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1\ e_2) : \tau}$$

$$\frac{\Gamma \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \tau' \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 : \tau}$$

Table 2.2: Underlying type system rules.

The objective of the exception analysis is to approximate what exceptions (if any) may the evaluation of an expression yield.

2.2.2 Underlying type system

The type and effect analysis will extend a standard type system with annotations. This underlying type system includes types of integers, booleans and and functions:

$$\tau \quad ::= \quad \mathsf{int} \ \mid \ \mathsf{bool} \ \mid \ \tau_1 \to \tau_2$$

The typing relation is presented as judgements $\Gamma \vdash e : \tau$ where Γ is a set of assumptions for free variables (i.e. pairs $x : \tau$). We use τ_c for the type of a constant c (an integer or boolean). The typing rules are presented in Table 2.2.

2.2.3 Effects and annotated types

The key idea of type and effect systems is to annotate function types with an *effect* φ that delimits side-effects that may be triggered during the evaluation of the function. For the exception analysis, effects represent sets of exception tokens:

$$\varphi \quad ::= \quad \emptyset \ \mid \ \{\epsilon\} \ \mid \ \varphi_1 \cup \varphi_2 \tag{2.2.3}$$

Equality of effects is taken modulo axioms for commutativity, associativity and idempotency of $\cup$ and with null element $\emptyset$. We will therefore abuse notation and sometime write effects as finite sets $\{\epsilon_1, \ldots, \epsilon_n\}$.

The syntax of annotated types is

$$\tau \quad ::= \quad \mathsf{int} \ \mid \ \mathsf{bool} \ \mid \ \tau_1 \xrightarrow{\varphi} \tau_2 \tag{2.2.4}$$

where int and bool are the types of constants and $\tau_1 \xrightarrow{\varphi} \tau_2$ is the type of a function that may raise exceptions *only* in φ. For example, the integer division operation can be given the annotated type $\mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\{\mathsf{div0}\}} \mathsf{int}$ meaning that it might raise a division-by-zero exception.[1] By contrast, the addition operation can be given the type $\mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}$ manifesting that it cannot raise exceptions

2.2.4 Subeffecting and subtyping

Effects can be ordered by a *subeffecting* relation $\subseteq$. Informally, $\varphi \subseteq \varphi'$ means that φ can be safely approximated by φ'. For the simple example of exception analysis, the subeffecting relation is just set-containment.

Subeffecting can be used to ensure that a type and effect system is a "conservative extension" of the underlying type system, i.e. to be able to derive an analysis for any expression that is typeable in the underlying type system. Consider, for example, the expression

$$\begin{aligned}\lambda y.\,\mathsf{if}\ \ \mathsf{then}\ y\ \mathsf{else}\ >0\ \mathsf{then}\ (\lambda x.\,\mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ >0\ \mathsf{then\ raise\ pos\ else}\ x)\\ \mathsf{else}\ (\lambda x.\,\mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ <0\ \mathsf{then\ raise\ neg\ else}\ x)\end{aligned} \tag{2.2.5}$$

The two x-abstractions may raise distinct exceptions and so admit different annotated types:

$$(\lambda x.\,\mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ >0\ \mathsf{then\ raise\ pos\ else}\ x)\ :\ \mathsf{int} \xrightarrow{\{\mathsf{pos}\}} \mathsf{int}$$

$$(\lambda x.\,\mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ <0\ \mathsf{then\ raise\ neg\ else}\ x)\ :\ \mathsf{int} \xrightarrow{\{\mathsf{neg}\}} \mathsf{int}$$

Without subeffecting expression (2.2.5) would not admit an effect annotated type even though it is admits a type the underlying type system. Subeffecting will allow us to "enlarge" the effects of both abstractions to

$$(\lambda x.\,\mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ >0\ \mathsf{then\ raise\ pos\ else}\ x)\ :\ \mathsf{int} \xrightarrow{\{\mathsf{neg},\,\mathsf{pos}\}} \mathsf{int}$$

$$(\lambda x.\,\mathsf{if}\ \ \mathsf{then}\ x\ \mathsf{else}\ <0\ \mathsf{then\ raise\ neg\ else}\ x)\ :\ \mathsf{int} \xrightarrow{\{\mathsf{neg},\,\mathsf{pos}\}} \mathsf{int}$$

and obtain the type

$$\mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\{\mathsf{neg},\,\mathsf{pos}\}} \mathsf{int}$$

for the whole expression (2.2.5).

Since types are annotated with effects, subeffecting induces a *subtyping* relation $\leqslant$ on annotated types. Informally, $\tau \leqslant \tau'$ means that τ can be safely approximated by τ'. The subtyping relation is formally defined by rules (2.2.6):

$$\tau \leqslant \tau \qquad \frac{\tau_1' \leqslant \tau_1 \qquad \tau_2 \leqslant \tau_2' \qquad \varphi \subseteq \varphi'}{\tau_1 \xrightarrow{\varphi} \tau_2 \leqslant \tau_1' \xrightarrow{\varphi'} \tau_2'} \tag{2.2.6}$$

Note that subtyping is *shape conformant* (or *structural*) i.e. if $\tau \leqslant \tau'$ then τ and τ' have the same underlying type but possibly distinct annotations; this is unlike more general kinds of subtyping that relate types with distinct constructors (e.g. a relation such as $\mathsf{int} \leqslant \mathsf{float}$ modelling coercion between numeric types).

Also note that the definition (2.2.6) is *covariant* on the right of the arrow but *contravariant* on the left, i.e. the ordering is reversed on the domains of functions. This is indeed the correct definition

[1] Note that currying allows distinguishing the effect of partial application (which can not raise exceptions) from the full application which can.

$$\Gamma \vdash c : \tau_c \;\&\; \emptyset \tag{2.2.7}$$

$$\Gamma \cup \{x : \tau\} \vdash x : \tau \;\&\; \emptyset \tag{2.2.8}$$

$$\Gamma \vdash \mathsf{raise}\ \epsilon : \tau \;\&\; \{\epsilon\} \tag{2.2.9}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \;\&\; \varphi_1 \qquad \Gamma \vdash e_2 : \tau \;\&\; \varphi_2 \qquad \Gamma \vdash e_3 : \tau \;\&\; \varphi_3}{\Gamma \vdash \mathsf{if}\quad \mathsf{then}\ e\ \mathsf{else}_{\,1}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi_3} \tag{2.2.10}$$

$$\frac{\Gamma \vdash e_1 : \tau' \xrightarrow{\varphi_0} \tau \;\&\; \varphi_1 \qquad \Gamma \vdash e_2 : \tau' \;\&\; \varphi_2}{\Gamma \vdash (e_1\ e_2) : \tau \;\&\; \varphi_0 \cup \varphi_1 \cup \varphi_2} \tag{2.2.11}$$

$$\frac{\Gamma \cup \{x : \tau'\} \vdash e : \tau \;\&\; \varphi}{\Gamma \vdash \lambda x.\, e : \tau' \xrightarrow{\varphi} \tau \;\&\; \emptyset} \tag{2.2.12}$$

$$\frac{\Gamma \vdash e_1 : \tau \;\&\; \varphi_1 \qquad \Gamma \vdash e_2 : \tau \;\&\; \varphi_2}{\Gamma \vdash \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 : \tau \;\&\; \varphi_1 \cup (\varphi_2 \setminus \{\epsilon\})} \tag{2.2.13}$$

$$\frac{\Gamma \vdash e : \tau \;\&\; \varphi}{\Gamma \vdash e : \tau \;\&\; \varphi'} \qquad \text{if } \varphi \subseteq \varphi' \tag{2.2.14}$$

Table 2.3: Type and effect rules for exception analysis.

regardless of the precise semantics of side-effects; the intuition for this is that $A \xrightarrow{\varphi} B$ is interpreted as an implication $A \implies (B \wedge \varphi)$ and $\leqslant$ as logical consequence.

Finally, we remark that subeffecting alone is sufficient to ensure that the exception analysis is a conservative extension of the underlying type system; this is because the only type annotations are effects, unlike more complex type-based analysis [2, 73, 127]. However, adding subtyping can still be useful to improve precision: while subeffecting requires enlarging effects at the point of definition, subtyping allows enlarging types at the points of use, thus limiting any precision loss to specific contexts.

2.2.5 Type and effect rules

The type and effect analysis is formulated in Table 2.3 as a set of typing rules that derive judgements with the form

$$\Gamma \vdash e : \tau \;\&\; \varphi$$

where Γ is a set of type assumptions for free identifiers, e is an expression, τ is an annotated type and φ is the effect associated with e. We describe each rule informally:

- Rules (2.2.7) and (2.2.8) specify the type and an empty effect $\emptyset$ for constants and identifiers: under a call-by-value semantics evaluation of these expressions cannot raise exceptions.
- Rule (2.2.9) specifies the most precise effect for an explicit raise, namely, the singleton exception raised; note that the result type τ is arbitrary.
- Rule (2.2.10) overestimates the effect of a conditional as the union of the effects of all sub-expressions.

- Rule (2.2.11) specifies the combination of effects for an application: the result is a union of the effects φ_1, φ_2 for evaluating both sub-expressions plus the "latent effect" φ_0 for the function itself.

- Conversely, rule (2.2.12) transposes the effect φ of an expression e into the annotation in the arrow type for the abstraction $\lambda x.\, e$. The evaluation the lambda-abstraction has an empty effect; this is because the effects of the function are delayed until the point of application.

- Rule (2.2.13) specifies the type and effect of the handle construct: an exception ϵ raised in e_2 is caught and therefore the result effect masks it using an "effect-difference" operation $\varphi_2 \setminus \{\epsilon\}$ (the definition is straightforward and we omit it). Note that exceptions other than ϵ raised in e_2 or those raised by e_1 are propagated to the outer scope.

- Rule (2.2.14) allows subeffecting only; it would be possible to allow subtyping as well [as in 114]:

$$\frac{\Gamma \vdash e : \tau \;\&\; \varphi}{\Gamma \vdash e : \tau' \;\&\; \varphi'} \qquad \text{if } \tau \leqslant \tau' \text{ and } \varphi \subseteq \varphi'$$

  We do not consider the extended rule here to avoid the treatment of subtyping in the inference algorithm. In any case, subtyping would only improve precision; subeffecting alone is sufficient to ensure that the exception analysis is a conservative extension of the underlying type system.

2.2.6 Semantic correctness

The correctness of the type and effect analysis can be formulated as a "subject reduction" property: if a type and effect can is inferred for an expression, it is also admissible for the result of evaluation.

For the particular exception analysis, this is formulated in the following theorem.

Theorem 2.1. If $\emptyset \vdash e : \tau \;\&\; \varphi$ and $\vdash e \longrightarrow v$, then $\emptyset \vdash v : \tau \;\&\; \varphi$.

In particular, if $\emptyset \vdash e : \tau \;\&\; \varphi$ and $\vdash e \longrightarrow \mathsf{raise}\ \epsilon$, then applying the above theorem we obtain $\emptyset \vdash \mathsf{raise}\ \epsilon : \tau \;\&\; \varphi$. By inspection of Table 2.3 we can see that the only type rules that could be applied to raise are (2.2.9) and (2.2.14). We conclude that $\{\epsilon\} \subseteq \varphi$, i.e. the analysis obtains an upper-approximation of the exceptions raised.

The proof of Theorem 2.1 is by induction on the big-step reduction $\vdash e \longrightarrow v$ together with a standard "substitution lemma" to allow replacing variables with expressions of the correct type. We omit the proof which is similar to the one presented in Nielson et al. [114, pages 295–297].

2.2.7 Type and effect polymorphism

For simplicity the language and type rules considered so far did not include polymorphic definitions. We will now add let-bound polymorphism by extending terms with an expression '$\mathsf{let}\ \ = x\ \mathsf{in}\ = e_1\ \mathsf{in}\ e_2$' and a type rule that allows quantified types for x in e_2. Moreover, it is possible to use polymorphism to obtain a more precise analysis by quantifying over effects as well as types. The extended syntax of

$$\frac{\Gamma \vdash e_1 : \sigma_1 \;\&\; \varphi_1 \qquad \Gamma \cup \{x : \sigma_1\} \vdash e_2 : \tau_2 \;\&\; \varphi_2}{\Gamma \vdash \mathsf{let} \;\; = x \;\mathsf{in}\; = e_1 \;\mathsf{in}\; e_2 : \tau_2 \;\&\; \varphi_1 \cup \varphi_2} \tag{2.2.15}$$

$$\frac{\Gamma \vdash e : \sigma \;\&\; \varphi}{\Gamma \vdash e : \forall\gamma.\, \sigma \;\&\; \varphi} \quad \text{if } \gamma \text{ does not occur free in } \Gamma \text{ and } \varphi \tag{2.2.16}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\, \sigma \;\&\; \varphi}{\Gamma \vdash e : \sigma[\alpha \mapsto \tau'] \;\&\; \varphi} \tag{2.2.17}$$

$$\frac{\Gamma \vdash e : \forall\beta.\, \sigma \;\&\; \varphi}{\Gamma \vdash e : \sigma[\beta \mapsto \varphi'] \;\&\; \varphi} \tag{2.2.18}$$

Table 2.4: Extensions for type and effect polymorphism.

terms and types is

$$\begin{array}{lcl}
e & ::= & \cdots \mid \mathsf{let} \;\; = x \;\mathsf{in}\; = e_1 \;\mathsf{in}\; e_2 \\
\varphi & ::= & \beta \mid \emptyset \mid \{\epsilon\} \mid \varphi_1 \cup \varphi_2 \\
\tau & ::= & \alpha \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \\
\sigma & ::= & \tau \mid \forall\gamma.\, \sigma \\
\gamma & ::= & \alpha \mid \beta \\
\alpha & ::= & \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \mid \ldots \\
\beta & ::= & '0 \mid '1 \mid '2 \mid \ldots
\end{array}$$

where α are *type variables*, β are *effect variables* and σ are *quantified types* (i.e. type schemes).

Table 2.4 lists the new type rules for the let-expression 2.2.15 and for introduction and elimination of type quantifiers (2.2.16), (2.2.17), (2.2.18); the rules of Table 2.3 remain unchanged.

Example 2.2 Consider a program that starts with the definition of a higher-order composition function,

$$\mathsf{let} \;\; = \;\mathsf{in}\; \mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x)\ \mathsf{in}\ e$$

where the expression e is the remaining of the program. Using type and effect polymorphism, we can derive a quantified type for *compose*

$$\forall\mathsf{a}\,\forall\mathsf{b}\,\forall\mathsf{c}\,\forall'0\,\forall'1.\,(\mathsf{b} \xrightarrow{'0} \mathsf{c}) \xrightarrow{\emptyset} (\mathsf{a} \xrightarrow{'1} \mathsf{b}) \xrightarrow{\emptyset} \mathsf{a} \xrightarrow{'0 \cup '1} \mathsf{c}$$

which specifies the "most general" annotated type with instances such as

$$(\mathsf{int} \xrightarrow{\emptyset} \mathsf{int}) \xrightarrow{\emptyset} (\mathsf{int} \xrightarrow{\emptyset} \mathsf{int}) \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}$$

and

$$(\mathsf{int} \xrightarrow{\{\mathsf{neg}\}} \mathsf{bool}) \xrightarrow{\emptyset} (\mathsf{int} \xrightarrow{\{\mathsf{pos}\}} \mathsf{int}) \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\{\mathsf{neg},\,\mathsf{neg}\}} \mathsf{bool}\ .$$

Note that with type but not effect polymorphism we would only be able to derive a type for *compose* annotated with the effects of *all* uses; this would lead to an over-estimation where each use shares the effects of all others.

By contrast, quantification on effects allows the analysis of polymorphic functions to be *polyvariant*: each application is given a range of effects specific to its arguments; this is particularly important for generic functions such as higher-order combinators that are typically used in very different contexts.

Note also that effect polymorphism allows expressing the analysis of *compose* without advance knowledge of its uses. This means that the analysis is *compositional*: it does require that the whole program be present and can be applied to separately-compiled modules or libraries.

We remark that the addition of polymorphism to the exception analysis is uncharacteristically simple. In general, the combination of polymorphism and imperative side-effects (e.g. references) requires restrictions on the use of the generalisation rule to retain soundness of type inference [145, 162].

2.2.8 Inference algorithms

Tables 2.3 and 2.4 present the type and effect analysis as proof systems that require guessing suitable types for sub-expressions; to obtain an automatic analysis we need an algorithm for *type and effect reconstruction*.

A first problem is the presence of non-structural type rules such as those for subeffecting (2.2.14), subtyping, generalisation (2.2.16) and instantiation (2.2.17), (2.2.18): these rules can occur in arbitrary points of a derivation and therefore some "canonical" choice has to be made; this is usually done by showing a *proof normalisation* result i.e. that uses of non-structural rules can be restricted to specific syntax points without a incurring a loss of typeability.

Let-bound polymorphism is a suitable choice for normalising the uses of generalisation and instantiation: generalise all suitable type and effect variables of let-bound identifiers[2] and instantiate all quantified variables just after the use of a variable. Subeffecting can be normalised by allowing over-approximation of effects in all rules i.e. by adding an arbitrary effect $\ldots \cup \varphi'$ to the conclusions of (2.2.7), (2.2.8), (2.2.9) and (2.2.12).

Type inference for type and effect systems with subeffecting but not subtyping can be implemented as an extension of the well-known algorithm W of Damas [35]. The key hindsight is to restrict types to the subset of *simple types* $\widehat{\tau}$ whose annotations must be variables:

$$\widehat{\tau} \quad ::= \quad \alpha \quad | \quad \mathsf{int} \quad | \quad \mathsf{bool} \quad | \quad \widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2$$

To allow expressing complex effects (i.e. non-variables) the algorithm collects separate *lower-bound constraints* C over effect variables:

$$C \quad ::= \quad \emptyset \quad | \quad \{\beta \supseteq \varphi\} \quad | \quad C_1 \cup C_2$$

$$\varphi \quad ::= \quad \emptyset \quad | \quad \beta \quad | \quad \{\epsilon\} \quad | \quad \varphi_1 \cup \varphi_2$$

The reason for restricting the algorithm to simple types is that these form a free algebra in which equality constraints can be solved by first-order unification [128] just as in ordinary Damas-Milner type inference. By contrast, the algebra of effects is non-free (e.g. $\cup$ is associative, commutative and has a empty element $\emptyset$). By segregating effects to separate constraints, it becomes possible to use the simple unification to solve type equalities and deal with the non-free algebra of effects in a separate constraint solver.

Table 2.6 presents an excerpt of the reconstruction algorithm as judgements

$$\widehat{\Gamma} \vdash_{\mathsf{RA}} e : (\widehat{\tau},\, \varphi,\, C,\, [])/$$

] where $\widehat{\Gamma}$ is a set of (simple) type assumptions, e is an expression and the output is a 4-tuple of: a simple type $\widehat{\tau}$, an effect φ, a set of lower-bound constraints C and a substitution [/.] For simplicity, we include only the rules for constants, abstraction and application; the omitted cases (conditionals and exception handling) are straightforward but tedious.

[2] That is, those that do not occur free in the type assumption or effect.

$$
\begin{aligned}
\mathcal{U}(\mathsf{int},\mathsf{int}) &= id \\
\mathcal{U}(\mathsf{bool},\mathsf{bool}) &= id \\
\mathcal{U}(\widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2, \widehat{\tau}_1' \xrightarrow{\beta'} \widehat{\tau}_2') = \quad \text{let} \quad & [/0] = [\beta \mapsto \beta'] \\
& [/1] = \mathcal{U}([/0]\widehat{\tau}_1, [/0]\widehat{\tau}_1') \\
& [/2] = \mathcal{U}([/1][/0]\widehat{\tau}_2, [/1][/0]\widehat{\tau}_2') \\
\text{in} \quad & [/2] \circ [/1] \circ [/0] \\
\mathcal{U}(\alpha, \widehat{\tau}) = \mathcal{U}(\widehat{\tau}, \alpha) &= \begin{cases} [\alpha \mapsto \widehat{\tau}] & \text{if } \alpha \text{ does not occur in } \widehat{\tau} \\ \text{fails} & \text{otherwise} \end{cases} \\
\mathcal{U}(\widehat{\tau}, \widehat{\tau}') &\quad \text{fails in all other cases}
\end{aligned}
$$

Table 2.5: Unification of simple types.

$$\widehat{\Gamma} \vdash_{\mathsf{RA}} c : (\tau_c,\, \emptyset,\, \emptyset,\, id)$$

$$\widehat{\Gamma} \cup \{x : \widehat{\tau}\} \vdash_{\mathsf{RA}} x : (\widehat{\tau},\, \emptyset,\, \emptyset, id)$$

$$\widehat{\Gamma} \vdash_{\mathsf{RA}} \mathsf{raise}\ \epsilon : (\alpha,\, \{\epsilon\},\, \emptyset, id)$$

$$\frac{\widehat{\Gamma} \cup \{x : \alpha\} \vdash_{\mathsf{RA}} e : (\widehat{\tau},\, \varphi,\, C,\, [])/}{]\widehat{\Gamma} \vdash_{\mathsf{RA}} \lambda x.\, e : ([\alpha/ \xrightarrow{]} \beta\widehat{\tau},\, \emptyset,\, \{\beta \supseteq \varphi\} \cup C,\, [])/}] \qquad \alpha,\, \beta \text{ are fresh variables}$$

$$\frac{\begin{array}{l}\widehat{\Gamma} \vdash_{\mathsf{RA}} e_1 : (\widehat{\tau}_1,\, \varphi_1,\, C_1,\, [/1]) \qquad [/1]\widehat{\Gamma} \vdash_{\mathsf{RA}} e_2 : (\widehat{\tau}_2,\, \varphi_2,\, C_2,\, [/2]) \\ [/3] = \mathcal{U}(\widehat{\tau}_2 \xrightarrow{\beta} \alpha, [/2]\widehat{\tau}_1)\end{array}}{\widehat{\Gamma} \vdash_{\mathsf{RA}} (@e_1 e_2) : ([/3]\alpha,\, [/3][/2]\varphi_1 \cup [/3]\varphi_2 \cup \{[/3]\beta\},\, [/3][/2]C_1 \cup [/3]C_2,\, [/3] \circ [/2] \circ [/1])}$$

Table 2.6: Algorithmic typing judgements for exception analysis (excerpt).

The main difference between the proof systems of Table 2.3 and Table 2.6 is that the latter does not require guessing types of sub-expressions; instead, it uses "fresh" variables for both types and effects and uses unification to impose equality constraints between (simple) types.

The unification algorithm $\mathcal{U}$ in Table 2.5 takes two simple types $\widehat{\tau}$, $\widehat{\tau}'$ and yields the "smallest" substitution [/s]uch that $[\widehat{/}\tau] \equiv [\widehat{/}\tau]'$ (or fails, if no such substitution exists). Note that substitutions bind both type and effect variables and therefore are applied to types, effects and constraints.

Each rule of Table 2.6 is applicable to a single expression syntax node; thus, the rules can be read as an algorithm for reconstructing the type and effect of an expression.

Extending the inference algorithm with let-bound polymorphism is straightforward: quantification of variables is done at the let =a in nd instantiation is done at the use of variables by introducing fresh type and effect variables. The type and effect system for region inference of Talpin and Jouvelot [139] combines polymorphism and effects (but not subtyping).

Type reconstruction algorithms for subtyping usually require extending the proof system with explicit type inequality constraints [43, 108]; this is needed to obtain syntactic completeness i.e. an algorithm that computes a principal solution from which any valid typing can be derived. This approach is followed in Nielson et al. [113, 115] although completeness of the algorithm is left as an open problem. For shape conformant subtyping typical of type and effect systems it is possible to employ a simpler two-stage approach: first the underlying types are inferred and then the subtyping inequalities are translated to constraints on the annotations [127]; such an algorithm will not be complete, i.e. it may compute a type and effect that is not minimal.

2.2.9 Concluding remarks

The type and effect discipline has some strengths compared to the other main approaches for program analysis (e.g. based on data-flow or abstract interpretation): it deals naturally with higher-order functions by means of annotations on arrow types; it allows separate analysis of modules by communicating information via extended type signatures; the latter also provide a natural mechanism for reporting the results of analysis to the user.

Effect systems where initially developed to deal the problems caused by implicit side-effects in functional languages with call-by-value semantics (e.g. in the LISP and ML families). As such, the type and effect framework does not naturally fit languages with lazy evaluation; this can be witnessed in the type rule (2.2.11) for application:

$$\frac{\Gamma \vdash e_1 : \tau' \xrightarrow{\varphi_0} \tau \;\&\; \varphi_1 \qquad \Gamma \vdash e_2 : \tau' \;\&\; \varphi_2}{\Gamma \vdash (e_1\ e_2) : \tau \;\&\; \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

The cumulative effect $\varphi_0 \cup \varphi_1 \cup \varphi_2$ always includes the argument effect φ_2, thus modelling a strict application.

Moreover, lazy languages do not typically employ *implicit* side effects since the order of evaluation is driven by demand. Side effects must then be controlled explicitly e.g. using *monads* [9, 157]. However, the two approaches are not completely apart: Wadler [158] has shown that monads can be parameterised by effects and that the inference rules and algorithms of type and effect systems carry over to the monadic translation.

2.3 Abstract interpretation

Abstract interpretation [28, 29] is a framework for program analysis based on approximating computations on *concrete values* by computations on *abstract properties* of values. The key idea is to approximate the concrete domain of values by a more coarse domain of abstract properties and lift the

concrete operations to sound abstract approximations. The static analysis is constructed by interpreting the program in the non-standard abstract domain.

2.3.1 Concrete and abstract domains

The theory abstract interpretation is concerned with establishing sound approximations independently of the syntactical characteristics of the programming language (or, more generally, model of computation). It is therefore usual to start by defining sets $\mathcal{P}^\natural$ of *concrete properties* and $\mathcal{P}^\sharp$ of *abstract properties*.

The concrete properties are derived from some semantic description of the language (e.g. they can be sets of values, states or given by a "collecting" semantics); the requirement for $\mathcal{P}^\natural$ is that is a complete proof method for the intended properties under analysis; therefore elements of $\mathcal{P}^\natural$ are usually not machine representable. Contrariwise, the abstract properties $\mathcal{P}^\sharp$ will be typically be both finitely representable and computable.

The sets of concrete and abstract properties are then instrumented with partial orders representing the relative precision of descriptions; a common scenario is to consider properties to be lattices $(\mathcal{P}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$. The convention is that precision is lost when moving upwards in the lattice: if $p, p' \in \mathcal{P}$ satisfy $p \sqsubseteq p'$ then any value described by p is also described by p', i.e. p entails p'. The bottom element $\bot$ represents the most precise property (i.e. divergent or non-reachable computations); the top element $\top$ represent the least precise property (absence of information). Note that it is possible for properties to be incomparable i.e. when neither $p \sqsubseteq p'$ nor $p' \sqsubseteq p$ holds.

2.3.2 Correspondence between concrete and abstract properties

The correspondence between concrete and abstract properties $c \in \mathcal{P}^\natural$ and $a \in \mathcal{P}^\sharp$ can be established in many ways; one of the most common scenarios is to require the existence of two monotone functions $\alpha : \mathcal{P}^\natural \to \mathcal{P}^\sharp$ and $\gamma : \mathcal{P}^\sharp \to \mathcal{P}^\natural$ called *abstraction* and *concretisation*, respectively. Informally, $\alpha(c)$ should be the "smallest" abstract representative of a concrete property c; dually, $\gamma(a)$ should be the "largest" concrete property described by an abstraction a.

The classical *Galois connection* framework requires that the abstraction and concretisation functions satisfy (2.3.1) and (2.3.2):

$$\forall c \in \mathcal{P}^\natural \quad c \sqsubseteq^\natural \gamma(\alpha(c)) \tag{2.3.1}$$

$$\forall a \in \mathcal{P}^\sharp \quad \alpha(\gamma(a)) \sqsubseteq^\sharp a \tag{2.3.2}$$

Condition (2.3.1) states that we may loose precision moving from concrete to abstract lattice and back again (though we do not loose safety); condition (2.3.2) states that we do not loose precision in the inverse direction.

In the presence of a Galois connection we can state the soundness relation between a concrete and abstract property using either the concrete or abstract orders. More precisely, $a \in \mathcal{P}^\sharp$ is a sound approximation of $c \in \mathcal{P}^\natural$ if either (2.3.3) or (2.3.4) holds:

$$c \sqsubseteq^\natural \gamma(a) \tag{2.3.3}$$

$$\alpha(c) \sqsubseteq^\sharp a \tag{2.3.4}$$

However, we shall consider the slightly more general setting where not every concrete value has a "best" abstraction e.g. when the domain of abstract properties is an incomplete lattice [30, 32]. In that situation we can do without the abstraction function and define the soundness relation using condition (2.3.3).

2.3.3 Approximation of fixed points

We now introduce a further assumption that the lattice $\mathcal{P}^\natural$ of concrete properties is *complete* so that we can define the invariant properties of recursive or iterative computations as fixed points.

Assume then that the semantics of a "basic block" of computation (e.g. a state transition or the body of a loop or recursive computation) is given by a continuous function $f : \mathcal{P}^\natural \to \mathcal{P}^\natural$ on concrete properties. The semantics of an iterative computation can then be obtained as the limit of an ascending chain of concrete iterates $\{(c_n)_{n\in\mathbb{N}}\}$:

$$c_0 := \bot^\natural \qquad c_{n+1} := f(c_n) \tag{2.3.5}$$

By completeness of $\mathcal{P}^\natural$ the limit $\bigsqcup\{(c_n)_{n\in\mathbb{N}}\}$ exists; by continuity of f we have $\bigsqcup\{(c_n)_{n\in\mathbb{N}}\} = \bigsqcup_{n\geq 0} f^n(\bot^\natural) = \mathit{fix}(f)$.

It will usually be the case that the least fixed point $\mathit{fix}(f)$ is incomputable and so we are interested in obtaining a computable approximation using a monotone abstract semantics function $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ describing a transition in abstract properties. Note that the requirement for monotonicity is quite natural for program analysis: it merely amounts to saying that $f^\sharp$ cannot yield more precise results from less precise initial approximations. Note also that we do not assume that the lattice $\mathcal{P}^\sharp$ is complete nor that $f^\sharp$ is continuous.

The soundness relation between f and $f^\sharp$ can specified using a Galois connection, i.e. an abstraction and concretisation functions. In that case $f^\sharp$ is completely determined by f, α and γ:

$$f^\sharp = \alpha \circ f \circ \gamma \tag{2.3.6}$$

Informally, condition (2.3.6) says that computing with $f^\sharp$ should yield the same result as first using γ to project into the concrete domain, computing with f and then using α to get back to the abstract domain.

In the absence of a Galois connection, it is possible to employ the concretisation function alone to specify the soundness relation using (2.3.3) in a pointwise manner:

$$\forall a \in \mathcal{P}^\sharp \quad f(\gamma(a)) \sqsubseteq^\natural \gamma(f^\sharp(a)) \tag{2.3.7}$$

Condition (2.3.7) says that computing with $f^\sharp$ and then projecting into the concrete domain must yield a upper-approximation of first projecting into the concrete domain and then computing with f.

Consider now the sequence of abstract iterates $\{(a_n)_{n\in\mathbb{N}}\}$ generated by $f^\sharp$:

$$a_0 := \bot^\sharp \qquad a_{n+1} := f^\sharp(a_n) \tag{2.3.8}$$

It is immediate that $\{(a_n)_{n\in\mathbb{N}}\}$ is an ascending chain in $\mathcal{P}^\sharp$ by the monotonicity of $f^\sharp$. In the simple situation where the abstract lattice of properties $\mathcal{P}^\sharp$ is *finite* the iteration (2.3.8) will converge to the least fixed point in a finite number of iterations.

However, this is not the case for lattices that capture numerical properties e.g. the lattices of integer intervals or convex polyhedra [32]. If the abstract lattice is incomplete or does not satisfy the finite ascending chain condition (see [152]) then the limit of (2.3.8) might not exist or the iteration might not converge finitely; in either case the abstract iteration does not give an effective computational method for obtaining a sound abstraction.

The solution is to replace (2.3.8) by another iteration that is an upper bound of the original and does stabilise in a finite number of steps; this can be done using a *widening operator*.

Widening operators

When the lattice of abstract values does not satisfy the ascending chain condition, we need some heuristic for extrapolating fixed point iterations. This is designated a *widening operator*.[3]

[3]There is some flexibility in the definition of widening operators; we use the formulation of [114].

Definition 2.3.1. A total function $\nabla : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ on a partially ordered set $(\mathcal{P}, \sqsubseteq)$ is a *widening operator* if and only if:

a) For all $x, y \in \mathcal{P}$ we have $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$;

b) If for all n, $x_n \in \mathcal{P}$ and $x_n \sqsubseteq x_{n+1}$, then the sequence $y_n \in \mathcal{P}$ defined by

$$y_0 := x_0 \qquad y_{n+1} := y_n \nabla x_{n+1}$$

eventually stabilises, i.e. there exists $k \geq 0$ such that for all n, $n \geq k$ implies $y_n = y_k$.

The first condition of this definition requires that ∇ is an upper bound operator (though not necessarily the *least* upper bound); the second condition requires that ∇ transforms ascending iterations into finitely stabilising iterations.

Assume we have a widening operator ∇ for the lattice of abstract properties $\mathcal{P}^\sharp$; then the following iteration

$$\begin{array}{rcll} a_0 & = & \bot^\sharp & \\ a_{n+1} & = & a_n & \text{if } f^\sharp(a_n) \sqsubseteq^\sharp a_n \\ a_{n+1} & = & a_n \nabla f^\sharp(a_n) & \text{otherwise} \end{array} \tag{2.3.9}$$

eventually stabilises [114, pages 227–228]. Therefore, we can compute successive iterates a_0, a_1, ... until the condition $f^\sharp(a_k) \sqsubseteq^\sharp a_k$ is satisfied; by the properties of the widening, this is guaranteed to happen in a finite number of iterations; then

$$\begin{array}{rr} f^\sharp(a_k) \sqsubseteq^\sharp a_k & \text{by the termination condition} \\ \Longrightarrow \gamma(f^\sharp(a_k)) \sqsubseteq^\natural \gamma(a_k) & \text{by monotonicity of } \gamma \\ \Longrightarrow f(\gamma(a_k)) \sqsubseteq^\natural \gamma(a_k) & \text{by hypothesis (2.3.7)} \\ \Longrightarrow \mathit{fix}(f) \sqsubseteq^\natural \gamma(a_k) & \text{by the fixed point theorem} \end{array}$$

The last line says that the abstract property a_k obtained from (2.3.9) is a sound approximation of $\mathit{fix}(f)$ as required.

2.3.4 Abstract interpretation of numerical properties

We shall now review some well-known lattices used in abstract interpretation of numerical properties. Such analysis approximate sets of integer vectors, i.e. elements of $\wp(\mathbb{Z}^n)$ (or equivalently, predicates over integer tuples).

The two lattices considered are the *intervals of integers* [27] and *convex polyhedra* [32]. Unlike elements of $\wp(\mathbb{Z}^n)$, both intervals and polyhedra are machine representable and the required operations on them are computable.

2.3.5 Lattice of intervals

The set of integer intervals is defined by

$$\mathbf{Interval} = \{\bot\} \cup \{\, [z_1,\, z_2] \,:\, z_1 \in \mathbb{Z} \cup \{-\infty\},\, z_1 \in \mathbb{Z} \cup \{+\infty\},\, z_1 \leq z_2 \,\}$$

where the order on the integers is extended to $-\infty$ and $+\infty$ by $-\infty \leq z$, $z \leq +\infty$ and $-\infty \leq +\infty$ for all $z \in \mathbb{Z}$; $\bot$ represents the empty interval and $[z_1, z_2]$ represents the interval from z_1 to z_2 including the endpoints if they are in $\mathbb{Z}$. The partial order $\sqsubseteq$ of *interval containment* is defined by

$$\mathit{int}_1 \sqsubseteq \mathit{int}_2 \overset{\text{def}}{\iff} \inf \mathit{int}_2 \leq \inf \mathit{int}_1 \wedge \sup \mathit{int}_1 \leq \sup \mathit{int}_2$$

where

$$\inf \bot = +\infty \qquad \sup \bot = -\infty$$
$$\inf [z_1,\, z_2] = z_1 \quad \sup [z_1,\, z_2] = z_2\,.$$

Then (**Interval**, $\sqsubseteq$, $\bot$, $[-\infty, +\infty]$, $\sqcup$, $\sqcap$) is a complete lattice [114, pages 221–222], where the join $\sqcup$ and meet $\sqcap$ are defined by

$$\begin{aligned}
[z_1,\, z_2] \sqcup [z_1',\, z_2'] &:= [\min(z_1, z_1'),\, \max(z_2, z_2')] \\
\bot \sqcup \mathit{int} &:= \mathit{int} \sqcup \bot := \mathit{int} \\
[z_1,\, z_2] \sqcap [z_1',\, z_2'] &:= \begin{cases} [\max(z_1, z_1'),\, \min(z_2, z_2')]\,, & \text{if } \max(z_1, z_1') \leq \min(z_2, z_2') \\ \bot & \text{otherwise} \end{cases} \\
\bot \sqcap \mathit{int} &:= \mathit{int} \sqcap \bot := \bot
\end{aligned}$$

and min and max extend to $-\infty$ and $+\infty$.

Operations on intervals

The best approximation of a non-empty set $X \subseteq \mathbb{Z}$ of integers is the interval $[\inf X, \sup X]$; dually, an interval $[l, r]$ represents the set $\{n \in \mathbb{Z} : l \leq n \leq r\}$; the interval $\bot$ represents the empty set. More generally, we define the abstraction and concretisation functions as follows:

$$\begin{array}{cc}
\alpha : \wp(\mathbb{Z}) \to \textbf{Interval} & \gamma : \textbf{Interval} \to \wp(\mathbb{Z}) \\
\alpha(\emptyset) := \bot & \gamma(\bot) := \emptyset \\
\alpha(X) := [\inf X, \sup X] \quad (X \neq \emptyset) & \gamma([l, r]) := \{n \in \mathbb{Z} : l \leq n \leq r\}
\end{array}$$

It is straightforward to verify that α, γ form a Galois connection. This allows lifting operations to intervals in a pointwise-manner; for example, the addition operation can be approximated by:

$$\bot + \mathit{int} = \mathit{int} + \bot = \bot$$
$$[l_1, r_1] + [l_2, r_2] = [l_1 + l_2, r_1 + r_2]$$

Similar approximations can be derived for other arithmetic operations.

Widening operators for intervals

Since the interval lattice has chains of infinite height we need a widening operator to ensure the termination of fixed point iterations.

Widening operators for intervals go back to one of the first publications on abstract interpretation; the widening operator ∇ proposed by Cousot and Cousot [27] was:

$$\begin{aligned}
\bot \nabla \mathit{int} := \mathit{int} \nabla \bot &:= \mathit{int} \\
[z_1,\, z_2] \nabla [z_1',\, z_2'] &:= [l,\, u] \\
\text{where } l &= \begin{cases} z_1 & \text{if } z_1 \leq z_1' \\ -\infty & \text{otherwise} \end{cases} \\
u &= \begin{cases} z_2 & \text{if } z_2' \leq z_2 \\ +\infty & \text{otherwise} \end{cases}
\end{aligned}$$

Example 2.3 Consider the simple imperative program (where **read** performs input yielding a boolean result):

$$\begin{array}{l} i := 0;\ c := \mathbf{true} \\ \mathbf{while}\ c\ \mathbf{do} \\ \quad i := i + 2 \\ \quad \mathbf{read}(c) \end{array} \tag{2.3.10}$$

We employ abstract interpretation on the lattice of intervals to approximate the range of values of the variable i inside the loop (2.3.10). The concrete transition function $f : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ associated with one iteration of the loop is

$$f(X) = \{0\} \cup \{n + 2 : n \in X\}$$

and the concrete semantics is $\mathit{fix}(f) = \bigcup_{n \geq 0} f^n(\emptyset) = \{0,\ 2,\ 4, \ldots\} = 2\mathbb{N}$.

The abstract transition function $f^\sharp : \mathbf{Interval} \to \mathbf{Interval}$ is

$$f^\sharp(\mathit{int}) = [0,\ 0] \sqcup (\mathit{int} + [2,\ 2])$$

where $+$ is the addition of intervals. The abstract iteration with widening $a_0 := \bot$ and $a_{n+1} := a_n \nabla f^\sharp(a_n)$ yields the following approximations:

$$\begin{aligned} a_0 &= \bot \\ a_1 &= \bot \nabla f^\sharp(a_0) = \bot \nabla [0,\ 0] = [0,\ 0] \\ a_2 &= [0,\ 0] \nabla f^\sharp(a_1) = [0,\ 0] \nabla [0,\ 2] = [0,\ +\infty] \end{aligned}$$

This stabilises after two iterations because $f^\sharp(a_2) = [0,\ +\infty] = a_2$; the limit $\bigsqcup_{n \geq 0} a_n = a_2 = [0,\ +\infty]$ is a sound (albeit imprecise) approximation of the range of values of i.

2.3.6 Lattice of convex polyhedra

Intervals approximate elements of $\wp(\mathbb{Z})$, i.e. sets of integers. To obtain an analysis for multiple components (e.g. ranges of two or more variables) it is possible to use products of intervals; this is designated an *independent attribute* combination because it does capture any interplay between components [114, pages 249–250].

Example 2.4 Consider the following imperative program:

$$\begin{array}{l} i := 0;\ j := 1;\ c := \mathbf{true} \\ \mathbf{while}\ c\ \mathbf{do} \\ \quad i := i + 1 \\ \quad j := j + 2 \\ \quad \mathbf{read}(c) \end{array}$$

The abstract interpretation of the ranges of i and j using the abstract domain $\mathbf{Interval} \times \mathbf{Interval}$ will obtain the ranges $i \in [0,\ +\infty]$ and $j \in [1,\ +\infty]$ but no relation between i and j.

To obtain relational information we need to approximate elements of $\wp(\mathbb{Z}^n)$, i.e. sets of tuples of integers. One of the most successful abstract domains for capturing linear numerical relations is the lattice of *convex polyhedra* [32]; this forms the basis for many verification tools and program analysis in use today.

A *closed convex polyhedron*, or simply a *polyhedron*, is the solution-set $P \subseteq \mathbb{R}^n$ of a system of linear inequations [132]

$$P = \{\, x \in \mathbb{R}^n \ :\ Ax \leq b \,\} \tag{2.3.11}$$

where $A \in \mathbb{Q}^{m \times n}$ is a matrix of coefficients and $b \in \mathbb{Q}^m$ is a vector of constants.[4] The set $\mathbb{CP}_n$ of convex polyhedra of dimension n ordered by $\subseteq$ is a lattice:

- the intersection $P \cap Q$ of two polyhedra P, Q is a polyhedron (the conjunction of the two systems of constraints);
- the union of two convex polyhedra is not necessarily convex, so the least upper bound of P, Q is not $P \cup Q$; instead it is the *convex hull* $P \uplus Q$, i.e. the smallest polyhedron that contains both P and Q. In general $P \cup Q \subseteq P \uplus Q$ (and the inclusion is strict when $P \cup Q$ is not a convex set).
- the empty set $\emptyset$ and the universe $\mathbb{R}^n$ are, respectively, the bottom and top elements of $\mathbb{CP}_n$.

We remark that $\mathbb{CP}_n$ is *not* a complete lattice: for example, the sphere is not a polyhedron but can be obtained as the limit of an infinite sequence of polyhedra.

The dual description method

We say that (A, b) of equation (2.3.11) is a *system of constraints* for P. A polyhedron can alternatively be characterised by a *system of generators* (V, R) as the sum of a convex combination of vertices $V = \{v_i \in \mathbb{Q}^n\}$ with a positive combination of rays $R = \{r_j \in \mathbb{Q}^n\}$,

$$P = \left\{ \sum_{i=1}^{|V|} \lambda_i v_i + \sum_{j=1}^{|R|} \mu_j r_j \ : \ \lambda_i \geq 0,\, \mu_j \geq 0,\, \sum_{i=1}^{|V|} \lambda_i = 1 \right\} \tag{2.3.12}$$

The two descriptions (2.3.11) and (2.3.12) are dual of each other in the sense that either one represents the polyhedron and that a single algorithm can switch between representations [22, 110, 155].

The dual description method represents polyhedra both by constraints and generators. This is justified because some operations are more efficient on the constraints while others are more efficient on the generators; others still benefit from *both* representations. Another important property is that the duality allows keeping the representations minimal, i.e. free of redundant constraints or generators. Efficient implementations of polyhedra computations are based on the dual description method, taking special care to avoid unnecessary conversions [8, 160].

Operations on polyhedra

We briefly describe the more common computations on polyhedra, mainly to fix notation. For a thorough description we point the reader to [8]. Let P and Q be two polyhedra of n dimensions $x_1, \ldots, x_n$. The following operations are all computationally effective:

Containment test: $P \subseteq Q$ holds if and only if the system of generators of P satisfies the constraints of Q.

Intersection: the system of constraints for $P \cap Q$ is obtained as the union of the constraints for P with those for Q.

Convex hull: the system of generators of $P \uplus Q$ is obtained as the union of the generators of P with those of Q.

Variable elimination: $\mathsf{ELIM}(x_i, P)$ is the polyhedron resultant from eliminating the dimension x_i from P by Fourier elimination [21]; the system generators of $\mathsf{ELIM}(x_i, P)$ is obtained by adding two rays $\{x_i, -x_i\}$ to the system of generators of P.

[4] Note that we intentionally restrict coefficients to *rational* rather than real numbers to ensure that these are machine representable.

Widening: if $P \subseteq Q$ then $P \nabla Q$ is the *widening* of P and Q. The standard widening for convex polyhedra is due to Halbwachs [51]; more recently, Bagnara et al. [7] proposed a more precise widening operator.

Widening operators for convex polyhedra

Since the lattice of convex polyhedra is incomplete, we need to employ a widening operator to guarantee termination of fixed point approximation (see Section 2.3.3).

The first widening operator for convex polyhedra was proposed by Cousot and Halbwachs [32] for synthesising loop invariants of imperative programs and later formalised by Halbwachs [51] in his PhD thesis. Informally, $P \nabla Q$ is the set of constraints of P that are still satisfied by Q. This is an upper bound operator because $P \nabla Q$ is defined by a subset of the constraints of both polyhedra. It is a widening because the system of constraints of P is finite, therefore it is not possible to keep removing constraints indefinitely.[5]

Example 2.5 Consider again the imperative program:

$$
\begin{array}{l}
i := 0;\ j := 1;\ c := \textbf{true} \\
\textbf{while}\ c\ \textbf{do} \\
\quad i := i + 1 \\
\quad j := j + 2 \\
\quad \textbf{read}(c)
\end{array}
$$

We will perform abstract interpretation using $\mathbb{CP}_2$ to determine loop invariants as linear inequalities in two dimensions. For readability, we represent elements of $\mathbb{CP}_2$ by systems of linear inequations using the same variable names i, j as the program.

The abstract semantic function $f^\sharp : \mathbb{CP}_2 \to \mathbb{CP}_2$ is

$$f^\sharp(P) = \{i = 0,\, j = 1\} \uplus \{(i + 1,\, j + 2) : (i, j) \in P\}$$

The abstract iteration with widening yields

$$
\begin{aligned}
a_0 &= \emptyset \\
a_1 = f^\sharp(a_0) &= \{i = 0,\, j = 1\} \uplus \emptyset \\
&= \{i = 0,\, j = 1\} \\
a_2 = f^\sharp(a_1) &= \{i = 0,\, j = 1\} \uplus \{i = 1,\, j = 3\} \\
&= \{0 \le i \le 1,\, j = 1 + 2i\} \\
f^\sharp(a_2) &= \{i = 0,\, j = 1\} \uplus \{1 \le i \le 2, j = 1 + 2i\} \\
&= \{0 \le i \le 2,\, j = 1 + 2i\} \\
a_3 = a_2 \nabla f^\sharp(a_2) &= \{0 \le i \le 1,\, j = 1 + 2i\} \nabla \{0 \le i \le 2,\, j = 1 + 2i\} \\
&= \{0 \le i,\, j = 1 + 2i\}
\end{aligned}
$$

The widening operator caused the unstable constraint $i \le 1$ to be discarded; the result a_3 is already a sound loop invariant because $f^\sharp(a_3) = \{i = 0,\, j = 1\} \uplus \{1 \le i,\, j = 1 + 2i\} = \{0 \le i,\, j = 1 + 2i\} = a_3$. In general, the iteration might not stabilise after a single application of the widening; the process must be repeated until a post fixed point is reached (this must happen in a finite number of steps by the properties of the widening).

[5] The formal definition is slightly more elaborate to make the widening well-defined for equivalent but syntactically-distinct constraint systems; see [7, 51] for the details.

We obtained not just a lower bound for i but also a linear relation $j = 1 + 2i$ between the two variables. Note also that the lower bound $1 \leq j$ for j is a consequence of the two linear constraints and therefore not explicitly stated in a minimal constraint system.

Halbwachs' widening has been used in most analysis tools employing convex polyhedra and has received the designation of the *standard widening* for convex polyhedra. However, this widening is sometimes too coarse, loosing many constraints before stabilising.

Several techniques have been proposed in the abstract interpretation literature to improve the precision of iteration with widening. One approach is simply to not use the widening of the first k iterations, where k is some fixed small constant. This is in fact what we did in Example 2.5: the widening was not applied until the third iteration. Delaying the application of widening allows accumulating more information during the first iterations; convergence is still ensured from the k-th iteration onwards.

A more precise variant is the "widening with tokens" of Bagnara et al. [7]. The iteration starts with a fixed number of *tokens*; one token is consumed each time the widening would cause a loss of precision and the exact least upper bound is used instead; the standard widening is used when there are no tokens left. The advantage of this technique is that the number of initial tokens specify the delaying of *actual* rather than *potential* losses of precision.

A final approach is to choose a more precise widening operator; this is highly dependent on the lattice of abstract properties and little can be said about how to proceed in general. Halbwachs' widening was the sole proposal for abstract interpretation using convex polyhedra from its inset in the late 1970s for over 20 years. More recently, Bagnara et al. [7] proposed a new widening operator for convex polyhedra which they proved to be no less precise than Halbwachs' widening in the worst-case and more precise in some cases.

2.4 Automatic complexity analysis

Early works in automatic cost analysis follow the methodology for hand analysis of algorithms e.g. as in the seminal textbook by Knuth [86]: first derive recurrence equations expressing the program cost (e.g. number of arithmetic or other primitive operations) in terms of some input metric (e.g. data size) and then solve the recurrences (maybe using approximation) to obtain a closed equation.

The earliest work following this methodology is Wegbreit's METRIC system [159]. METRIC derived complexity equations for list functions written in a first-order subset of LISP with recursive procedures but no side-effects or imperative features. The system obtained metrics such as time, length or size as a 4-tuple $\langle min, max, avg, var \rangle$ of lower bound, upper bound, average and standard deviation; the first two are best and worst-case bounds; the last two measures are derived under the assumption of statistical independence of dynamic tests. The performance measures are expressed symbolically as functions of input size or length and the costs of primitive operations.

METRIC first transformed a recursive function into a step-counting version, i.e. a function with the same domain and whose value is the cost metric (here "cost" can be length, size or time, i.e. number of reduction steps). As an example, consider a function appending two lists:[6]

$$\begin{aligned} APPEND(X, Y) = &\text{ if NULL}(X) \text{ then } Y \\ &\text{else CONS}(\text{CAR}(X),\ APPEND(\text{CDR}(L), Y)) \end{aligned}$$

[6] In this example we use LISP constructors names: *CONS* is the pair constructor; *CAR* and *CDR* project the first and second element of a pair; *NIL* is a constant; and *NULL* tests equality to the *NIL* constant.

The step counting function for *APPEND* under the time metric produced the symbolic cost function

$$\begin{aligned} time(APPEND(X,Y)) = &\text{if NULL}(L) \text{ then } null + 2vref \\ &\text{else } null + 4vref + cdr + car + cons + \\ &\quad time(APPEND(\text{CDR}(L),\, Y)) \end{aligned}$$

where *null*, *car*, *cdr* and *cons* are symbolic constants for the costs for the primitive list operations and *vref* is the cost for accessing a variable.

The system now projects the step-counting function into a recurrence equation over the integers, using either a measure of the *size* (total number of nodes) or *length* (number of nodes along the *cdr*-direction) of arguments. For this example, the recursion involves only the *cdr* of the first argument, so METRIC chooses the length measure and expresses

$$\begin{aligned} time(APPEND(X,Y)) &= T(length(X)) \\ T(0) &= null + 2vref \\ T(n+1) &= null + 4vref + cdr + car + cons + T(n) \end{aligned}$$

The system then attempts to solve the recurrence equation to obtain a closed-form expression. The solution for the example is:

$$\begin{aligned} time(APPEND(X,Y)) = &null + 2vref + \\ &(null + 4vref + cdr + car + cons) \times length(X) \end{aligned}$$

Note that this example is uncharacteristically simple: in general to analyse a function under one measure, METRIC might have to perform sub-analysis under other measures (e.g. length or size of sub-expressions). Solving recurrence equations with more alternatives also requires more sophisticated techniques e.g. generating functions.

METRIC was able to obtain closed forms for simple LISP programs e.g. list reverse, flattening, membership test and union. The analysis could be used to predict heap allocation (e.g. the number of *cons*-cells allocated) by setting the costs of other primitives to zero. However, it could not be used to predict for maximum stack depth because of the underlying assumption that the cost metric is *cumulative*:

$$time(F(G(X))) = time(G(X)) + time(F(Y))\,, \quad \text{where } Y = G(X)$$

While the above assumption holds for time[7] it does not hold for space e.g. stack. The combination of stack costs should be (ignoring the overheads for function applications)

$$stack(F(G(X)) = \max(stack(G(X)),\, stack(F(Y))\,, \quad \text{where } Y = G(X)$$

i.e. the stack depth for the composition is the *maximum* of the stack depths for the sub-expression and outer call. While in theory it suffices to synthesise recurrences with maximums instead of sums, in practice such recurrences are much harder to solve automatically because the maximum is not an analytic function.

METRIC was also limited to list processing: the complexity bound are derived with respect to either the length or the size of *S*-expressions; the system chooses one of the two measures using an heuristic based on the use of arguments in recursive calls. There is no way to use specialised measures as would be desirable e.g. for user-defined data types.

Le Mtayer's ACE system also performed complexity analysis by deriving a recursive step-counting function from each recursive function [90]. However, a first departure from the work by Wegbreit is that

[7] But is accurate only under *call-by-value* reduction strategy; see [156] for a formalisation of the corresponding assumption for *call-by-need* strategy i.e. lazy evaluation.

the costs measure considered is *worst-case* only and *asymptotic*; this means that individual primitive operations are not accounted, only the number of recursive calls.

Second, unlike Wegbreit's approach of projecting the cost function on the integers and solving recurrence equations, ACE obtained closed-form solutions by a series of meaning-preserving program transformations within a functional calculus (a subset of the FP language). The transformations are based on an algebra of applicative program transformations together with McCarthy's recursion induction principle: two functions satisfying the same recurrence equation are equal.[8]

For example, the factorial function can be expressed in FP as follows:

$$\text{fact} = \text{eq0} \rightarrow \text{“1”}; *o[\text{id}, \text{fact} \, o \, \text{sub1}]$$

The definition is in point-free style: $f \rightarrow g; h$ is a conditional with test f and true and false branches g and h; eq0 tests the argument for zero; id is the identity function; o is function composition; “k” is the constant k-valued function; sub1 is the integer predecessor function; $[f_1, \ldots, f_k]$ builds a sequence by applying f_i to an argument; $*$ and $+$ are arithmetic operators which operate on sequences of two values.

The step-counting function *Cfact* mimics the recursion structure of *fact* but adds one unit of cost for each call and zero for constants and primitives:

$$\begin{aligned}\text{Cfact} = &\text{eq0} \rightarrow +o[\text{“0”}, \text{”0”}]; \\ &+ o[\text{“0”}, +o[\text{“0”} \; + o[\text{“0”}, +o[\text{plus1} \, o \, \text{Cfact} \, o \, \text{sub1}, \; \text{“0”}]]]]\end{aligned}$$

To obtain a closed-form solution for the recursive *Cfact*, ACE employs a number of program transformations expressed as rewrite rules. For example, using the rules $+o[\text{“0”}, f] = +o[f \, \text{”0”}] = f$ (i.e. zero is the neutral element for sum) the definition of *Cfact* can be simplified to:

$$\text{Cfact} = \text{eq0} \rightarrow \text{“0”}; \text{plus1} \, o \, \text{Cfact} \, o \, \text{sub1}$$

Using the recursion induction principle, the above definition is matched against a library to find

$$\text{Cfact} = \text{id}$$

i.e. the complexity of factorial is linear in the argument value.

Le Mtayer reports success in analysing numerical programs, sorting algorithms and a parser. However, no indication is given as to the quality of results. Moreover, the quality results appear to be very sensitive to the set of rewrite rules provided: the implementation is said to use over 1,000 rules of various kinds. The extent to which these match specific programs or general programming patterns is not discussed.

The system deals with time in a very abstract manner (since only function calls are accounted) and not heap space or stack depth. The same limitations regarding cumulative cost measure that apply to Wegbreit's METRIC hold here. Asymptotic results would, in any case, be insufficient for bounding time or space in high integrity systems because the lower order terms might dominate the actual worst-case.

Another severe limitation for domains where high integrity is required is that a single incorrect rule in the database compromises the soundness of the results. This is even more problematic if the user is allowed to extend the system with new axioms and rules for specific programs.

Le Mtayer points out that the complexity functions resulting from ACE can sometimes be several lines long; such a result would be unintelligible for a human reader and possibly unusable by a compiler. Finally, the ACE system is very tightly coupled with a particular language: algorithms that are not naturally expressed in FP are very difficult to analyse in ACE.

[8] On the domain of the least solution of the recurrence.

Rosendahl [129] describes a semantic-based method for deriving the step-counting version of recursive first-order functions. The main contribution is the use of abstract interpretation to define a time-bound function whose inputs are partial representations of the original program inputs and whose output is an upper-bound on the original program time. No attempt is made to obtain closed cost expressions.

Liu and Gomez [94] have also presented an automatic time analysis for a first-order subset of Lisp based on obtaining a time-bound function on partial representation of inputs. Instead of trying to obtain closed-form solutions, they use the time-bound function to compute upper bounds. For example, to obtain a bound for sorting a list of n values, they symbolically execute the time-bound function with a list of n "unknown" LISP atoms. Unnikrishnan, Stoller, and Liu [150] have applied this technique to obtain bounds on stack and heap costs.

The limitation of this approach is that it does not yield a closed cost expression: if the original function is recursive, so is the time-bound function. Moreover, symbolic execution of the time-bound program with a partial input will not terminate if the recursion involves an unknown term. To ensure termination the input must be of fixed size (e.g. a list of 10 unknown values); the time-bound function then returns the cost for that specific size. Thus, this approach is closer to profiling than static analysis.

Furthermore, the time-bound function can be exponentially more expensive to compute than the original program, since it has to execute both branches of conditionals that depend on unknowns. This leads to performance problems with even moderate size inputs: Unnikrishnan et al. report a running time exceeding 2 hours to obtain upper bounds for merge sort of 30 elements and were unable to obtain bounds for 1,000 elements.

All works in complexity analysis described so far [90, 129, 159] assumed a cumulative measure of time cost. This is a very coarse overestimate under lazy evaluation, since the arguments may be only partially evaluated. The problem of *compositional* time analysis for lazy evaluation is that the time cost for an expression depends on the context, i.e. the amount of the input that is "needed" by each function.

Wadler [156] proposed a formalism for time analysis for first-order lazy evaluation using *projection transformers* to capture the "neediness" of each function. This work presents a formalism for expressing the step-counting equations but not an algorithm for approximating them.

Bjerner and Holmstrm [12] also proposed a time analysis for first-order lazy functional programs. This approach is based on an abstract representation of demand in result values; they then perform a backwards demand analysis to find out how much of the input is required to produce the required output. One limitation is that the representation of a demand requires knowing in advance much information about the output value.

2.5 Type and effect systems for time

Dornic, Jouvelot, and Gifford [38] presented a polymorphic "time system" for deriving time costs for higher-order call-by-value functional language. This system is an instance of a type and effect analysis where an underlying type system is extended with "effects" that approximate some intentional property of evaluation [84, 139, 140]. In the time system, effects approximate the number of computation steps needed to reduce an expression to normal form.

The starting point is the simply typed lambda-calculus with a strict (i.e. call-by-value) semantics.[9] A type judgement $\Gamma \vdash e : \tau$, where Γ is a typing context, e is a term and τ is a type, is augmented with

[9] To simplify the presentation, we use the lambda-calculus rather than the *CT* language of [38].

an *effect* n:

$$\Gamma \vdash e : \tau \ \$ \ n \tag{2.5.1}$$

The effect n is either a natural number representing an upper-bound on the number of reduction steps for e or a distinguished element `long` representing a potentially unbounded reduction. Thus, the augmented judgement (2.5.1) reads: *under assumptions* Γ*,* e *has type* τ *and cost* n*.*

As in other type and effect systems, functional types in Dornic's time system are annotated with a "latent effect" (designated *latent cost* in this system) that expresses the cost of function evaluation. The latent cost mechanism allows capturing the cost of application of higher-order functions quite naturally, as can be seen in the type rule for application:

$$\frac{\Gamma \vdash e_0 : \tau' \xrightarrow{l} \tau \ \$ \ m \qquad \Gamma \vdash e_1 : \tau' \ \$ \ n}{\Gamma \vdash (e_0 \ e_1) : \tau \ \$ \ m + n + l + 1} \tag{2.5.2}$$

The type rule expresses the cost of an application $(e_0 \ e_1)$ as a sum of: the cost m of reducing e_0 to some lambda-abstraction $\lambda x.\ e'$; the cost n of reducing the argument e_1 to a normal form v; the latent cost l of reducing the *β-reduct* $[x \mapsto v]\, e'$; and one extra unit to account for the application itself.

The dual rule for lambda-abstraction transposes the actual cost of a function body into a latent one:

$$\frac{\Gamma,\, x : \tau' \vdash e : \tau \ \$ \ m}{\Gamma \vdash (\lambda x.\, e) : \tau' \xrightarrow{m} \tau \ \$ \ 1} \tag{2.5.3}$$

Rules (2.5.2) and (2.5.3) reflect the chosen cost model: each application, lambda-abstraction and variable access cost one unit. Different choices could easily be accommodated by choosing different constants in the type rules.

Using these rules we can derive a "timed" type for the higher-order term $twice \equiv \lambda f.\, \lambda x.\, f\ (f\ x)$ as follows:

$$\begin{aligned}
&f : \tau \xrightarrow{l} \tau,\ x : \tau \vdash x : \tau \ \$ \ 1 && \{\text{hypothesis}\}\\
&f : \tau \xrightarrow{l} \tau,\ x : \tau \vdash f : \tau \xrightarrow{l} \tau \ \$ \ 1 && \{\text{hypothesis}\}\\
&f : \tau \xrightarrow{l} \tau,\ x : \tau \vdash (f\ x) : \tau \ \$ \ 3 + l && \{\text{application, arithmetic}\}\\
&f : \tau \xrightarrow{l} \tau,\ x : \tau \vdash (f\ (f\ x)) : \tau \ \$ \ 5 + 2l && \{\text{application, arithmetic}\}\\
&f : \tau \xrightarrow{l} \tau \vdash \lambda x.\, (f\ (f\ x)) : \tau \xrightarrow{5+2l} \tau \ \$ \ 1 && \{\text{abstraction}\}\\
&\vdash \lambda f.\, \lambda x.\, (f\ (f\ x)) : (\tau \xrightarrow{l} \tau) \xrightarrow{1} \tau \xrightarrow{5+2l} \tau \ \$ \ 1 && \{\text{abstraction}\}
\end{aligned}$$

The latent cost in the result type captures the cost duplication of the argument function (plus a constant overhead for *twice* itself) because the function is applied twice. The cost of *twice* is therefore parametric on the cost of its argument.

The full power of the time system is only obtained when the language is extended with polymorphism. Dornic et al. introduce polymorphism via a "polymorphic lambda"; we will do so using let-bound polymorphism. In an expression

$$\mathsf{let} \quad = \quad \mathsf{in}\ twice = \lambda f.\, \lambda x.\, f\ (f\ x)\ \mathsf{in}\ \ldots$$

the identifier *twice* can be given a type quantified over type *and* cost variables:

$$\forall a.\, \forall l.\, (a \xrightarrow{l} a) \xrightarrow{1} a \xrightarrow{5+2l} a$$

Note that the latent cost l is a quantified time variable; this means that distinct uses of *twice* in the remaining program can be typed with different costs. Moreover, the analysis does not require the whole

program: the quantified type of *twice* captures all information needed for future uses, so it is possible to perform separate analysis of libraries and modules.

However, the time system has some important limitations: first, recursive functions are always assigned the unbounded cost `long`. This is because the cost of a recursive function depends on the *sizes* of arguments which are not captured in the time system. The absence of size information also severely limits the precision of the analysis of higher-order functions, since the costs cannot depend on the sizes of arguments.

Second, the type system does not allow *subeffecting* i.e. subsuming a cost by a larger one; this is needed e.g. to be able to type a conditional with different costs in each branch; the extension (adding a maximum function to the cost algebra) is proposed as further work.

Finally, Dornic et al. considered only *checking* timed types, i.e. all time information must be prescribed as type annotations; the problem of *reconstructing* timed types was address in the subsequent work by Reistad and Gifford [127].

Reistad and Gifford extended the time system of Dornic et al. with annotations representing sizes of naturals, lists and vector types and with an algorithm to reconstruct sizes and times based on algebraic reconstruction of effects [84]. This system has been applied to aid dynamically scheduling in a parallelising implementation of the μFX language.

The size annotations represent upper bounds on the dynamic sizes of values. For example, the values of a type $\mathsf{Nat}\ n$ are the naturals less-than or equal to n. More interestingly, annotations in function types describe size changes e.g. the type for the successor function is

$$succ : \forall n.\, \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1)$$

assuming a cost of one unit for the operation. The algebra for sizes and costs includes a value `long` to represent a potentially unbounded sizes, and operations of addition, maximum and multiplication. This *sized* timed type system allows subsuming sizes: for example, the typing rule for natural constants is

$$\frac{nat \leq n}{\Gamma \vdash nat : \mathsf{Nat}\ n\ \$\ C_{num}}$$

where C_{num} is cost associated with naturals. Thus, the natural 1 admits any type $\mathsf{Nat}\ n$ for $1 \leq n$ (including `long`). Without this flexibility a conditional expression like "$\mathsf{if}\ \ \mathsf{then}\ \ldots\ \mathsf{else}\ \mathsf{then}\ 1\ \mathsf{else}\ 2$" would not admit a type because $\mathsf{Nat}\ 1 \neq \mathsf{Nat}\ 2$. The ordering relation $\leq$ on sizes induces a structural subtyping relation $\leqslant$ on the annotated types [108].

Adding size information to types allows specifying more precise costs for e.g. the higher-order *map* that applies an argument function to each element in a list:

$$map : \forall\{a, b, c, l\}.\, (a \xrightarrow{c} b) \times \mathsf{List}\ a\ l \xrightarrow{k_0 + l \times (k_1 + c)} \mathsf{List}\ b\ l$$

Note that the type for *map* expresses not just that the result list has the same length l as the input, but also the cost of *map* as a function of the argument cost c and list length l.[10] Such dependency was not possible in the time system of Dornic et al. because of the absence of size information.

The main limitation of this work is the absence of a treatment of recursion. As in the time system of Dornic et al. recursive functions can only be typed with a `long` cost. To mitigate this, Reistad and Gifford use a fixed set of higher-order functions (such as *map* above) to express primitive recursion schemes.

A minor limitation of the size semantics is that sizes of non-increasing functions must be overestimated. For example, a subtraction operator on the naturals must be given the type

$$sub : \mathsf{Nat}\ n \times \mathsf{Nat}\ m \xrightarrow{C_{sub}} \mathsf{Nat}\ n$$

[10] Constants k_0 and k_1 must be chosen to reflect the overheads associated with a particular implementation.

because we only know that the second argument is at most m (in particular, it could be zero). One solution to this problem would be to extend the size semantics to include lower bounds as well as upper bounds, i.e. intervals.

Another limitation of the system of Reistad and Gifford is an overestimation of sizes caused by insufficient polymorphism in higher-order functions. Consider the following sized timed types[11] for the higher-order function *twice* and the natural successor:

$$\begin{aligned} &\mathit{twice} : \forall a.\, \forall b.\, \forall l.\, \langle (a \xrightarrow{l} b) \xrightarrow{1} a \xrightarrow{5+2l} b,\ b \leqslant a \rangle \\ &\mathit{succ} : \forall n.\, \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1) \end{aligned}$$

To type the application (*twice succ*) we must solve the subtyping constraints:

$$\begin{aligned} \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1) &\leqslant a \xrightarrow{l} b \\ b &\leqslant a \end{aligned}$$

Decomposing the subtyping constraints we get (note the contravariance on the left-side of the arrow):

$$\begin{aligned} a &\leqslant \mathsf{Nat}\ n \\ \mathsf{Nat}\ (n+1) &\leqslant b \\ b &\leqslant a \\ 1 &\leq l \end{aligned}$$

Since the subtyping is shape conformant, we can now substitute $a \equiv \mathsf{Nat}\ i$ and $b \equiv \mathsf{Nat}\ j$ for some sizes variables i, j and obtain the size inequations $i \leq n \wedge n+1 \leq j \wedge 1 \leq l \wedge j \leq i$. It is straightforward to check that the only solution is $n = i = j = \texttt{long}$ and $7 \leq l$. Thus, the application must be typed as

$$\mathit{twice}\ \mathit{succ} : \mathsf{Nat}\ \texttt{long} \xrightarrow{7} \mathsf{Nat}\ \texttt{long}$$

i.e. although the latent cost for the result function is accurate, no size information is known.

This problem was designated *size aliasing* in [124] and is caused by the use of a monomorphic type at two distinct sizes. One solution is to extend the type system with discrete polymorphism i.e. intersection types [136]. In such a system *twice* admits types of the form

$$\mathit{twice} : (\tau \xrightarrow{l} \tau' \wedge \tau' \xrightarrow{l'} \tau'') \xrightarrow{1} \tau \xrightarrow{5+l+l'} \tau''$$

By instantiation, *succ* admits the types

$$\begin{aligned} &\mathit{succ} : \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1) \\ &\mathit{succ} : \mathsf{Nat}\ (n+1) \xrightarrow{1} \mathsf{Nat}\ (n+2) \end{aligned}$$

and therefore

$$\mathit{succ} : (\mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1)) \wedge (\mathsf{Nat}\ (n+1) \xrightarrow{1} \mathsf{Nat}\ (n+2))$$

Finally, @*twicesucc* can be typed as

$$\mathit{twice}\ \mathit{succ} : \mathsf{Nat}\ n \xrightarrow{7} \mathsf{Nat}\ (n+1)$$

[11] We present the type scheme for *twice* in a more general form than that of Reistad and Gifford by allowing subtyping constraints in quantified types [43, 108]. This is done in order to stress that the problem is due to insufficient polymorphism rather than insufficient subtyping.

which accurately expresses both the size and time of the application.

Loidl [95] proposed a type analysis with size and time information for aiding the scheduling of tasks in a parallel implementation of functional languages by providing static granularity information. He proposes extending the size and time analysis of Reistad and Gifford [127] to recursive functions by synthesising recurrence equations; these can then be solved to obtain closed-form cost equations either manually or with the aid of a computer algebra system. To make the analysis automatic, Loidl proposes building a library of known recurrence forms, as was done by Rosendahl [129].

This size and cost analysis was further developed by Vasconcelos and Hammond [154], which also presented results from a prototype implementation. The difficulties that apply to other approaches based on synthesising recurrences [90, 159] hold here: obtaining approximate solutions to recurrence equations automatically is difficult; the use of a library of recurrences mitigates this problem to some extent, but for practical use this must contain a large number of distinct but similar recurrences; furthermore, a single incorrect assumption in this library invalidates the soundness of the analysis, which is particularly relevant for our intended application domain of embedded and real-time systems.[12]

A further limitation is the loss of precision with irregular divide-and-conquer recursions e.g. as in the quicksort algorithm. This is shared by other type analysis where the sizes of data components are independently approximated and will discussed in more detail in the next section.

2.6 Sized types

Some researchers have presented type based analysis for size information alone; this can be useful for proving termination, enabling optimisations in compilers (e.g. eliminating array bounds checks) or for enabling program transformations (e.g. partial evaluation).

Hughes, Pareto, and Sabry [74] presented a type system extended with size information for proving liveness properties of reactive systems, namely termination and productivity.

The term language is purely-functional, non-strict and higher-order with let-bound polymorphism, general recursion and algebraic data types. The type system syntactically distinguishes *inductive* data (e.g. naturals or finite lists) from *co-inductive* data (e.g. infinite lists). The notion of "size" is purely denotational: it is an upper bound on the number of constructors of an inductive data value (and dually, a *lower* bound for co-inductive data). For example, given the declarations for naturals, finite lists and infinite lists (streams),

$$\begin{aligned}
\mathsf{idata}\ \mathsf{Nat} &= \mathsf{Zero} \mid \mathsf{Succ}\ \mathsf{Nat} \\
\mathsf{idata}\ \mathsf{List}\ a &= \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a) \\
\mathsf{codata}\ \mathsf{Stream}\ a &= \mathsf{Mk}\ a\ (\mathsf{Stream}\ a)
\end{aligned}$$

the corresponding sized types for constructors are:

$$\begin{aligned}
\mathsf{Zero} &: \mathsf{Nat}_1 \\
\mathsf{Succ} &: \forall i.\, \mathsf{Nat}_i \to \mathsf{Nat}_{i+1} \\
\mathsf{Nil} &: \forall a.\, \mathsf{List}_1\ a \\
\mathsf{Cons} &: \forall i.\, \forall a.\, a \to \mathsf{List}_i\ a \to \mathsf{List}_{i+1}\ a \\
\mathsf{Mk} &: \forall i.\, \forall a.\, a \to \mathsf{Stream}^i\ a \to \mathsf{Stream}^{i+1}\ a
\end{aligned}$$

The data types are annotated with a subscript or superscript size annotation (for inductive or co-inductive data, respectively). Sizes expressions are built up using naturals, variables and additions;

[12] It is worth remarking that soundness of cost approximations is not critical for the granularity analysis of Loidl, since erroneous cost information could reduce performance but not cause failure.

more complex size relations are disallowed (e.g. multiplication); these restricted forms allow for size checking using a Presburger constraint solver such as the Omega Calculator [125].

The types for Succ, Cons and Mk express size relations: the result has one more constructor than the argument. Note that Zero and Nil have size *one* (not zero) because the size is the number of *constructors* of the value.

Sized data types can be seen as infinite families of approximations indexed by the number of constructors e.g. $\mathsf{Nat}_0 \subseteq \mathsf{Nat}_1 \subseteq \mathsf{Nat}_2 \subseteq \ldots$. A special annotation ω is used to denote the "limit" of these approximations, e.g. Nat_ω is the type of all naturals.[13] As in the system of Reistad and Gifford, the size ordering induces a structural subtyping relation on sized types. Subtyping is necessary e.g. to be able to assign a sized type to a conditional expression returning values of different sizes.

The novelty of the type system of Hughes et al. is a typing rule for recursive functions that embodies a principle of induction on sizes and that guarantees termination for recursive function (and dually, productivity for co-recursive functions). Omitting type variable generalisations for simplicity, the rule is:

$$\frac{\begin{array}{c} all(\tau[0]) \\ \Gamma \vdash \lambda x.\, M : \forall i.\, \tau[i] \to \tau[i+1] \\ \Gamma \cup \{x : \forall i.\, \tau[i]\} \vdash N : \tau' \end{array}}{\Gamma \vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ N : \tau'} \quad i \notin FV(\Gamma) \tag{2.6.1}$$

The first two hypotheses express the induction on a size variable i that occurs in a type $\tau[i]$ (we use square brackets for a context):

a) $\tau[0]$ must be a *universal type*, i.e. one that includes the totally undefined value $\bot$;[14]

b) progress must be made at each recursive call i.e. we must be able to derive $\tau[i+1]$ assuming $\tau[i]$.

To see how rule (2.6.1) rejects non-terminating functions, consider the following (erroneous) list length function:

$$\mathit{wronglen}\ xs = \mathsf{case}\ xs\ \mathsf{of}\ \mathsf{Nil} \to \mathsf{Zero} \mid \mathsf{Cons}\ x\ xs' \ \to\ \mathsf{Succ}\ (\mathit{wronglen}\ xs)$$

This function diverges for non-empty lists because the recursive call is on xs rather than xs'. Type checking against the type

$$\mathit{wronglen} : \forall i.\, \forall a.\, \mathsf{List}_i\ a \to \mathsf{Nat}_i$$

gives rise to the proof obligation

$$\{\, \mathit{wronglen} : \mathsf{List}_i\ a \to \mathsf{Nat}_i,\ xs : \mathsf{List}_{i+1}\ a,\ \ldots \} \vdash \mathsf{Succ}\ (\mathit{wronglen}\ xs) : \mathsf{List}_{i+1}\ a$$

But typing the application $(\mathit{wronglen}\ xs)$ requires solving the subtyping constraint $\mathsf{List}_{i+1}\ a \leqslant \mathsf{List}_i\ a$ which is impossible because $i + 1 \not\leq i$; the erroneous function is therefore rejected.

The correctness of rule (2.6.1) was proved using a non-standard type semantics that allows types to exclude $\bot$ (i.e. non-termination/non-productivity). A sketch of the proof was presented in [74]; the complete proof together with type checking algorithm was presented by Pareto [120]. The algorithm requires let-bound identifiers (in particular, recursive functions) to be annotated with sizes but infers those of intermediate expressions. The type checker rejects programs whose termination/productivity is not ensured by the sized type annotations provided by the user. By the undecidability of the halting problem, such a decision procedure must reject some terminating/productive programs as well.

[13] Note that ω is *not* a size but rather a separate annotation, unlike `long` in the system of Reistad and Gifford. The distinction is important e.g. when instantiating a quantified size variable with ω, which is not always sound in Hughes and Pareto's system.

[14] The semantics for sized types of Hughes et al. is based on upwards-closed sets rather than the standard semantics based on ideals [101], so that sized types can exclude $\bot$.

The sized type system allows primitive recursive definitions over naturals and lists (e.g. *append*, *map* and *filter*). Functions with an accumulating parameter (e.g. *reverse*) can also be accepted by extending the type rule for recursion with size polymorphism (but not type polymorphism, thus retaining decidability of type checking). More complex recursions can sometimes be re-written so that the type checker will accept them (e.g. the system rejects the usual first-order definition of the Ackermann function, but accepts the higher-order primitive recursive one).

However, the system has limitations with irregular recursion patterns e.g. divide-and-conquer algorithms where the data size does not reduce uniformly in recursive calls. Consider the quicksort algorithm for lists, using an auxiliary function *split_by* that breaks up a list into two sub-lists of the smaller and greater elements with respect to a pivot[15]:

$$
\begin{array}{l}
qsort : \mathsf{List}\ t \to \mathsf{List}\ t \\
qsort\ [\,] = [\,] \\
qsort\ (x : xs) = \mathsf{case}\ split\_by\ x\ xs\ \mathsf{of} \\
\qquad\qquad\qquad (l, r) \to qsort\ l +\!\!+ [x] +\!\!+ qsort\ r \\
\\
split\_by : t \to \mathsf{List}\ t \to \mathsf{List}\ t \times \mathsf{List}\ t \\
split\_by\ pivot\ [\,] = ([\,], [\,]) \\
split\_by\ pivot\ (x : xs) = \mathsf{case}\ split\_by\ pivot\ xs\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\qquad (l, r) \to \mathsf{if}\quad \mathsf{then}\ x\ \mathsf{else}\ \leq pivot\ \mathsf{then}\ (x : l,\ h)\ \mathsf{else}\ (l,\ x : h)
\end{array}
$$

To typecheck *split_by* in the system of Hughes et al., we must choose some size i as induction variable; the natural choice is the size of the list argument (since *split_by* is defined by primitive recursion in that argument). The result depends on a dynamic test, so we can only derive a sized type with upper bounds (which are admissible by subtyping):

$$split\_by : \forall i.\, t \to \mathsf{List}_i\ t \to \mathsf{List}_i\ t \times \mathsf{List}_i\ t \tag{2.6.2}$$

Note that sizes of the result are overestimated. We would like to express a more precise relation, namely that the *sum* of the sizes of the two result lists equals the size of the argument. However, a type such as

$$\forall i\, j.\, t \to \mathsf{List}_{i+j}\ t \to \mathsf{List}_i\ t \times \mathsf{List}_j\ t$$

is *not* admissible by rule (2.6.1), since we cannot do induction on $i + j$. Using (2.6.2) as assumption for *split_by*, the type system still accepts quicksort with the type

$$qsort : \forall i.\, \mathsf{List}_i\ t \to \mathsf{List}_\omega\ t$$

which does not give an upper-bound for the size of the sorted list. Note, however, that due to the non-standard type semantics, the above sized type still ensures the termination of *qsort*.

Similarly, the sized type system is not well suited for algorithms over non-linear data structures such as trees (even though the theory of sized types is developed for generic algebraic data types). This is because the notion of size is always the depth of constructors and size relations must be linear; for example: a tree traversal algorithm exhibits complexity that is linear on the number of nodes but exponential on the tree depth and therefore would not be expressible. These limitations suggest that the sized type system, while guaranteeing very strong properties (termination/productivity), is also very restrictive in practice.

The original sized type system of Hughes et al. deals with a purely denotational notion of size, but not space or time costs. In a later work, Hughes and Pareto [73] extended the size type system with effects approximating stack and heap costs for a prototype language called Embedded ML.

[15]For simplicity, we use in this example a Haskell-style syntax for list operations. We also avoid issues of *ad-hoc* polymorphism by assuming some monomorphic type t with a total order $\leq$.

Embedded ML is first-order and with a strict semantics. The model of stack and heap costs is given by an abstract machine based on the SECD [87]. Dynamic heap allocation and deallocation is done using *regions*. The standard region system of Tofte and Talpin [147] introduces an allocation primitive

$$\mathsf{letregion}\ \rho\ \mathsf{in}\ e$$

where ρ is region variable that can used for allocations in e. After evaluation of e, region ρ is deallocated. The type and effect system of Tofte and Talpin guarantees that well-typed programs dot not access regions after deallocation.

The combination of sized types and regions allows sized of regions to be specified at the point of allocation; overflow is prevented at compile-time by the type system. Thus, the region allocation becomes

$$\mathsf{letregion}\ \rho\#e'\ \mathsf{in}\ e$$

where e' is an expression that specifies the size of region ρ. The type judgements

$$\Gamma \vdash e : \tau\ !\ \delta; p; \phi$$

are extended with effects δ, p and ϕ: δ is the *stack effect*, p is the *put effect* and and ϕ is the *store effect*. The stack and store effect are natural numbers and approximate the maximum stack depth and heap allocations during evaluation of e. The put effect tracks allocations done in regions in the current scope.

Type checking can now ensure at compile-time the absence of space overflow. Consider a function that constructs a list of naturals[16]:

$$\begin{aligned} \mathit{nats}\ n\ r = \mathsf{Cons}\ n\ (\mathsf{case}\ n\ \mathsf{of} & \\ & 0 \rightarrow \mathsf{Nil}\ r \\ & \mid\ m+1 \rightarrow \mathit{nats}\ m\ r)\ r \end{aligned}$$

The type checker accepts *nats* with type

$$\mathit{nats} : \forall k\, r.\, \mathsf{Nat}_k \times r \rightarrow \mathsf{List}_{k+1}\ (\mathsf{Nat}_k)\ r\ \mathsf{with}\ \delta = 5k; r{+}{=} 3k+1$$

which specifies both stack and heap allocation as functions of the size k. The stack effect accounts for 5 stack words at each recursive invocation: one word for each bound variable n, r, m, the intermediate result and the return address. The put effect specifies that region r can grow by (at most) $3k+1$ heap cells (the constants are derived from the particular operational semantics). We now see that the application

$$\mathsf{letregion}\ r\#13\ \mathsf{in}\ \mathit{length}\ (\mathit{nats}\ 4\ r)$$

is rejected by the type system because the local region r is too small for the computation of *nats*: the size k of the list is 5 (not 4) so at least $3 \times 5 + 1 = 16$ heap cells are required. To fix the program, it suffices to specify a larger size for region r. The correctness of the type system with respect to an abstract machine is presented in detail in Pareto [121].

One first limitation of this work is that the let-region allocation is *not* sufficient to obtain bounded space behaviour in reactive systems because region lifetimes have to be nested. To deallocate values but re-use regions, Hughes and Pareto propose extending their system with *region resetting* [11]. Furthermore, Embedded ML has no language mechanism for specifying reactive or infinite computations. Streams cannot be implemented as ordinary data types because the language has a strict semantics and is first-order.

[16] Like the system of Tofte and Talpin, constructors such as Nil and Cons take an extra region argument to specify where values are to be allocated.

Limitations regarding the expressive power of the recursion rule that applied to the size system alone also hold here.

Another drawback of Hughes and Pareto's approach is that it requires user annotations of both sizes *and* costs. While sizes are denotational properties that programmers can reason about in a high-level language, stack and heap costs are dependent on implementation details. Requiring the user to specify costs in type annotations (even if these are checkable by the compiler) lowers the level of software development. Even if fully automatic inference is not feasible, it would be preferable to have the user write size annotations and have the system infer costs automatically; this was left as future work in [73] and not addressed in [121].

Chin and Khoo [23] addressed the problem of *inferring* rather than just *checking* sized types. This system extends the prior work in two regards: first, they allow sizes to be expressed as general *Presburger constraints* (first-order logic formulae with linear arithmetic over the integers)[17] and second, by presenting an algorithm that computes a size formulae for a recursive function using an operation of "transitive closure" on constraints [85].

For example, the analysis of Chin and Khoo infers the following size information for the standard list append function:

$$\begin{aligned} \mathit{append} : \mathsf{List}^m\ t \to \mathsf{List}^n\ t \to \mathsf{List}^l\ t \\ \text{s.t. } \mathbf{size}\ m \geq 0 \wedge n \geq 0 \wedge l = m + n \\ \mathbf{inv}\ 0 \leq m^+ < m \wedge n^+ = n \end{aligned}$$

The size constraint expresses the dependency between input and output list sizes, while the invariance constraint expresses properties that hold for all recursive calls (where n^+ and m^+ are the sizes of arguments in recursive calls). Invariants such as these are useful in termination analysis or in programming transformations such as partial evaluation. We will focus here on inference of size relations, since similar techniques are used for inference of invariants.

Note that the size information on the list length is more precise than what could be expressed in the system of Hughes et al.: rather than just an upper bound, the equality constraint expresses the exact result size. In general, it is possible to express lower bounds, upper bounds or equalities (simultaneous lower and upper bound).

The term language is a strict, higher-order functional notation with integers, booleans and lists. Data types are annotated with size variables and all size information is expressed by separate size constraints. Thus typing judgements take the form

$$\Gamma \vdash e :: (\tau,\, \phi)$$

where e is an expression, τ an annotated type and ϕ a constraint on the annotations of τ expressing the size of e.

The notion of size is specific to each data type: the size of a list is its length; the size of an integer is its value (negative sizes for negative integers); boolean values False and True have sizes 0 and 1, respectively. Assigning sizes to booleans (and other enumerated types) allows expressing control flow information in size constraints. For example, consider the function testing a list for emptiness:

$$\mathit{null}\ xs = \mathsf{case}\ xs\ \mathsf{of}\ [\,] \to \mathsf{True}\ \mid\ x : xs' \to \mathsf{False}$$

The sized type inferred for *null* is

$$\mathit{null} :: (\mathsf{List}^n a \to \mathsf{Bool} c,\ (n = 0 \wedge c = 1) \vee (n > 0 \wedge c = 0))$$

[17] By contrast, type annotations in the system of Hughes and Pareto are restricted to linear size expressions. The type checking algorithm, however, generates a set of Presburger constraints for verifying the admissibility of typing.

where the size c of the boolean result encodes which branch of the conditional was taken.

Unlike the system of Hughes and Pareto, the typing rule for recursive functions in Chin and Khoo's system does not impose a well-founded order on sizes. Again omitting type generalisation for simplicity, the type rule is:

$$\frac{\Gamma \cup \{x :: (\tau_1,\, \phi_1)\} \vdash e_1 :: (\tau_1,\, \phi_1) \qquad \Gamma \cup \{x :: (\tau_1, \phi_1)\} \vdash e_2 :: (\tau_2,\, \phi_2)}{\Gamma \vdash \mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2 :: (\tau_2,\, \phi_2)} \tag{2.6.3}$$

Re-visiting our erroneous length function example, rule (2.6.3) allows us to type it as

$$\mathit{wronglen} :: (\mathsf{List}^i a \to \mathsf{Int}^j,\, i \geq 0 \wedge j \geq 0)$$

which expresses approximate information about the function semantics (namely, that both the argument and result must have non-negative sizes). A more precise information about *wronglen* is expressed by the sized type

$$\mathit{wronglen} :: (\mathsf{List}^i a \to \mathsf{Int}^j,\, i = 0 \wedge j = 0)$$

that is, *wronglen* is only defined for the empty list. Both sized types are admissible by rule (2.6.3).

The treatment of recursion in the system of Chin and Khoo corresponds to a distinct objective to that of Hughes and Pareto: rule (2.6.1) guarantees a *liveness* property (termination/productivity) whereas rule (2.6.3) guarantees a *safety* property (approximation of the dynamic sizes of values).

The type rules for expressions and non-recursive functions in Chin and Khoo's system are syntax directed, so that size type inference can be done by synthesising constraints from sub-expressions. However, rule (2.6.3) for letrec is not syntax directed since the size constraint for the result appears in the hypothesis. To compute a size constraint for

$$\mathsf{letrec}\ f = \lambda x.\, M\ \mathsf{in}\ N$$

Chin and Khoo first compute a constraint for the non-recursive term

$$\lambda f.\, \lambda x.\, M$$

i.e. a constraint expressing the size-change relation between two successive recursive iterations; then they employ an algorithm to approximate the *transitive closure* of a Presburger constraint [85]. One difficulty is that the transitive closure might not be expressible as a Presburger formula and the algorithm sometimes yields lower-bound approximations. This is inadequate for the type rule (2.6.3), so some post-processing steps are employed to obtain a safe upper bound. These computations are implemented using the Omega Calculator [125].

Chin and Khoo formulate the soundness of their size analysis with respect a standard higher-order denotational semantics. The proof, however, has one important technical flaw: it relies the existence of a constraint $\mathcal{S}(v :: \tau)$ describing the exact size of a value v of annotated type τ. While this is valid for zero-order values, it fails to hold for functional values because of the lattice of constraints is an incomplete partial order.

List constructors have specialised typing rules instead of being treated as constants as in the sized type systems [74, 127]; this is required to derive a type for lists where the head has different size than the tail.

$$\frac{\Gamma \vdash e_1 :: (\tau_1,\, \phi_1) \qquad \Gamma \vdash e_2 :: (\mathsf{List}^m \tau_2,\, \phi_2)}{\Gamma \vdash (e_1 : e_2) :: (\mathsf{List}^n \tau,\, n = m + 1 \wedge \phi_1 \wedge \phi_2 \wedge (\tau = \tau_1 \vee \tau = \tau_2))} \tag{2.6.4}$$

In rule (2.6.4) the type τ is constrained to be identical to τ_1 and τ_2 except for "fresh" size annotations; the equations $\tau = \tau_1$ and $\tau = \tau_2$ specify the equality constraints between size annotations in two types; the size constraint for the application specifies the length of the result list and size elements.

One limitation of the type system is that while rule (2.6.4) can be used to infer size relations on the list lengths, it often fails to infer sizes of values *inside* lists. Consider for example, the *tail* function on a list of integers:

$$tail\ \overbrace{(\underbrace{x}_{\mathsf{Int}^k} : \underbrace{xs}_{\mathsf{List}^m\mathsf{Int}^j})}^{\mathsf{List}^n\mathsf{Int}^i} = \overbrace{xs}^{\mathsf{List}^m\mathsf{Int}^j}$$

Rule (2.6.4) generates the size constraint

$$n = m + 1 \wedge (\mathsf{Int}^i = \mathsf{Int}^k \vee \mathsf{Int}^i = \mathsf{Int}^j) \iff n = m + 1 \wedge (i = k \vee i = j)$$

where m, k and j are "fresh" size variables; to obtain the type for the function, the constraint is simplified by existentially quantifying variable k that does not occur in the argument or result type; this yields the sized type

$$tail :: (\mathsf{List}^n\mathsf{Int}^i \to \mathsf{List}^m\mathsf{Int}^j,\ n = 1 + m)$$

where no size information is obtained for elements inside the list.

In a subsequent work Chin et al. [24] propose an extension to the sized type system with collection constraints to address this problem. However, the extended constraints fall outside the capabilities of a Presburger solver; the cited paper does not address the issue of solving these combined constraints.

Another limitation is that the type system is not type polymorphic since no size information is captured for type variables. The system is still able to obtain good size information for monomorphic instances. For example, the first tuple projection can be typed $fst : (\mathsf{Int}^i \times \mathsf{Int}^j \to \mathsf{Int}^k,\ k = i)$ but no size information is obtained for the polymorphic version $fst : (\forall a \forall b.\, a \times b \to b,\ True)$.

The size type system of Chin and Khoo allows higher-order functions, but no size relations are inferred from uses of functional arguments. For example, no sizes are inferred for the usual *compose* function $\lambda f.\, \lambda g.\, \lambda x.\, @f(@gx)$. Moreover, since the type system does not capture sizes for polymorphic functions, there is no analogue of a "principal size type" to be inferred in such cases.

2.7 Dependent types

The defining characteristic of dependent type systems is the possibility of parameterising types over values. Dependent type systems generalise the function type $A \to B$ to the *dependent product* $\Pi x : A.\, B$ where the type B of the co-domain is allowed to vary with x; the simple function type is obtained as an instance where x does not occur in B.

Restricted forms of type dependency have long been used informally in programming languages. For example, the Pascal array type depends on its size; and the types of arguments of the C-language `printf` depend on its first argument (the format string). Dependent type systems are formal basis for reasoning about such notions.

By the Curry-Howard correspondence dependent types allow expressing both *propositions* and *computational* (data) types in a single framework; therefore dependent type theories form the basis of proof assistants e.g. *Coq* [76] and program verifiers e.g. *Lego* [99].

More recently, there has been an increase of research in functional programming languages incorporating dependent types e.g. *Dependent ML* [163], *Cayenne* [5], *Agda* [25] and *Epigram* [103]. This is motivated by the desire to express more refined program properties using types than is possible with the standard polymorphic type systems. In fact, some extensions of the Haskell type system implemented in GHC e.g. type classes with functional dependencies [77] and generalised algebraic data types [78] allow simulating some of the expressive power of dependent types [3, 102].

Dependent ML (DML) is a conservative extension of the ML language with dependent types [163, 164]. The motivation for DML was to extend a realistic programming language with dependent types

whist retaining both decidability of type checking and a low overhead of type annotations. This is achieved by separating arbitrary ML terms (where general recursion is allowed and whose equivalence is therefore undecidable) from the *indices* allowed in types (taken from some decidable constraint domain).

Computation on DML type indices is restricted to constraint normalisation; this allows reducing the type checking of DML programs to constraint solving in the underlying domain. The constraint domain of natural indices with addition allows capturing size invariants of data structures; deciding the equivalence of the DML types with such indices can then be reduced to checking equivalence of Presburger constraints e.g. using the Omega calculator [125]. Xi [163] presented applications of DML types with natural indices to program error detection and optimisations e.g. array bounds check and dead-code elimination.

Dependent types in DML are introduced by *refining* a standard data type declaration. For example, a canonical declaration for a list data type

```
datatype 'a = nil | cons of 'a * a' list
```

can be refined with a natural length measure by the declaration:

```
typeref 'a list of nat with
   nil <| 'a list(0)
 | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

This refinement assigns a type with length zero for `nil` and a type for `cons` that increases the length by one; the notation `{n:nat}` is the concrete syntax for introducing a dependent product $\Pi n : \texttt{nat}$. Size properties regarding lists can then be expressed by dependent type annotations; for example, the size relation for the list append function is expressed by the type

```
append <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

and the DML type checker can verify that this size relation holds for the canonical recursive definition of append.

For size relations that cannot be expressed exactly, DML allows the use of *dependent sum types*. For example, the higher-order *filter* function computes a sub-list of elements verifying some predicate; since the length of result depends on the predicate it cannot be specified exactly; however, an upper-bound can be specified by the type

```
filter <| ('a -> bool) * {n:nat} 'a list(n)
                                    -> [m:nat | m<=n] 'a list(m)
```

where `[m:nat | m<=n]` is a *dependent sum* that constraints the result list length m to be at most the length n of the original.

DML with integer indices allows expressing properties similar to the sized type systems [23, 74, 127]. The main distinctions between the two approaches are: DML indices are user-definable for each data type whereas the notion of "size" in the sized type systems above is rigid; there is no implicit subtyping relation for size coercion in DML (instead, relevant functions must be annotated with dependent sum types); and finally, the DML type checker can *verify* user-annotated size relations but not *infer* them as in [23, 127].

Grobauer [46] presented a method for automatically deriving cost recurrences from first-order DML programs. The main contribution is the use of indices in DML types as data sizes for expressing the recurrences. This allows the user to specify more precise size measures for data e.g. nested lists or trees. The cost model is asymptotic (e.g. the number of function calls or some other primitive operation).

This work focuses on extracting cost recurrences but not on obtaining *solutions* to the cost equations. Except in very simple cases, obtaining closed form solutions requires human intervention. For example, a function merging two lists in order (part of a merge sort example)

```
fun merge l = case l of
   (nil, l2) => l2
   (l1, nil) => l1
   (cons(h1,t1),cons(h2,t2)) =>
       if h1<h2 then cons(h1, merge(t1,l2))
                else cons(h2, merge(l1,t2))
with merge <| {n1:nat}{n2:nat} list(n1)*list(n2) -> list(n1+n2)
```

yields the following cost recurrence (braces represent possibly-guarded maximum between alternatives):

$$merge^c\ n_1\ n_2 = \begin{cases} n_1 = 0 \mapsto 0 \\ n_2 = 0 \mapsto 0 \\ n_1 > 0 \wedge n_2 > 0 \mapsto 1 + \begin{cases} merge^c\ (n_1 - 1)\ n_2 \\ merge^c\ n_1\ (n_2 - 1) \end{cases} \end{cases} \tag{2.7.1}$$

It is immediate that the cost recurrence mimics the recursive structure of the original function. Even using computer algebra systems such as *Maple* or *Mathematica*, some human intervention is required to convert a recurrence such as (2.7.1) into the closed form expression $merge^c\ n_1\ n_2 = \min(n_1, n_2)$.

Crary and Weirich [33] used a system based on *proof-carrying code* [111] to perform verification of resources bounds. This system is based on a intermediate compiler language called *LXres* that allows expressing resource properties in types by exposing a "virtual clock" representing some available resource (e.g. time). Resource properties can then verified by the type checker. To deal with variable-time procedures, they employ a technique of encoding static type-level representations of data using using sum and inductive kinds; this simulates type dependency while allowing a simpler theory and type checker.

Costs can be expressed as primitive-recursive functions over the static data representations (so that type checking remains decidable). These must be provided by the user: the system allows *verifying* resource bounds, but makes no attempt to *infer* them.

Brady and Hammond [17] employed a dependently-typed language similar to *Epigram* to encode and verify size properties of functional programs. Their approach generalises the previous examples of sized lists in DML by introducing a dependent type Size that pairs a type indexed by a natural size and a predicate (itself represented as an dependent type). A term $\mathsf{size}\ v\ p : \mathsf{Size}\ A\ P$ pairs a value v of indexed type $A\ n$ and a proof p that v respects a size property P.

Brady and Hammond applied this framework to express size relations of functions on lists, including an example similar to the *split_by* function of Section 2.6. They also extend the technique to capture size relations for higher-order functions by associating size predicates and functions with higher-order arguments. The authors illustrate the technique with the higher-order functions such *twice*, *map* and *fold*.

A first limitation of this work is that it considers only verifying sizes expressed as dependent types. The elaboration of a simply typed program in Haskell or ML into a dependently typed version with size annotations is left to the user (particularly guessing size relations of functions). The extent to which this step can be automated is not addressed.

Secondly, this work shows the use of dependent types for expressing size information but not time or space costs. Although the authors mention that the technique is extendible to other metrics such as heap, stack or time usage, we remark that there is a fundamental distinction the *denotational* property

such as size and an *intentional* property such as cost; the former is a property of *values* while the latter is a property of *computations*.

Danielsson [36] has also used a dependently-typed language for expressing complexity analysis of functional programs. This work focuses on expressing costs rather than sizes by encapsulating values in a cost monad [157] parameterised by the number of computation steps: $\mathsf{Thunk}\ n\ a$ is the type of a computation that evaluates to an a in n steps. The unit and bind operations for the thunk monad are:

$$\begin{aligned} &\mathit{return} : a \to \mathsf{Thunk}\ 0\ a \\ &(\gg\!=) : \mathsf{Thunk}\ m\ a \to (a \to \mathsf{Thunk}\ n\ b) \to \mathsf{Thunk}\ (m+n)\ b \end{aligned}$$

The monadic unit injects a value into the cost monad with zero cost while the bind combines costs from two computations. Any atomic costs must be explicitly introduced using "tick" annotations in the program; each *tick* adds one unit of cost:

$$\mathit{tick} : \mathsf{Thunk}\ n\ a \to \mathsf{Thunk}\ (1+n)\ a$$

Note that the Thunk type is dependent on the natural n and that both the monadic operations and *tick* have dependent types.

These basic combinators form library implemented in the dependently typed language *Agda* and allow a programmer to specify machine-checkable complexity proofs; for example, assuming a dependent type for lists annotated with the length, and assigning a unit cost to each lambda-abstraction, we can typecheck a list concatenation function annotated with a linear cost on the first argument:

$$\begin{aligned} (+\!\!+) &: \mathsf{List}\ m\ a \to \mathsf{List}\ n\ a \to \mathsf{Thunk}\ (1+2*m)\ (\mathsf{List}\ (m+n)\ a) \\ [\,] +\!\!+ ys &= \mathit{tick}\ (\mathit{return}\ ys) \\ (x : xs) +\!\!+ ys &= \mathit{tick}\ (xs +\!\!+ ys \gg\!= \lambda t \to \mathit{tick}\ (\mathit{return}\ (x : t))) \end{aligned}$$

The cost monad is quite expressive e.g. it can be used to reason about the complexity of *lazy* evaluation by explicitly embedding Thunk types into data structures.

One limitation of this work is that it requires insightful annotations by the user and a considerable knowledge of dependent type systems. For example, to type check the concatenation example above requires providing a lemma for the arithmetic equality $1+((1+2*m)+(1+0)) = 1+2*(m+1)$. Non-trivial programs also require the introduction of auxiliary operators e.g. to "waste" costs and ensure that the two branches of a conditional admit the same type[18].

The cost model used is quite abstract: it counts number of "steps" specified by the number of ticks annotated in the code. Presumably the technique could be extended to a model of cost based on an abstract machine e.g. as in [73].

Finally, the system allows only *checking* cost bounds but does not aid in obtaining the cost bounds in the first place.

2.8 Amortised cost analysis

Amortised complexity analysis aims at obtaining bounds for the cost of a sequence of operations [118, 143]; it is sometimes possible to obtain better worst-case bounds by amortisation than by reasoning about the costs of individual operations. For example, it might be possible to obtain a worst-case bound of $O(n)$ for a sequence of n operations even if some of the individual operations cost more than $O(1)$.

[18] This is analogous to the subeffecting allowed in effects systems for time [127] expect that the latter is implicit.

The "physicist method" for deriving amortised bounds starts by assigning a non-negative *potential* function to data. The *amortised cost* of an operation is then defined as the sum of the actual cost (e.g. time cost or heap cells allocated) plus the difference in potential incurred by the operation. The key idea is to choose the potential functions so as to facilitate computing the amortised cost e.g. in such a way as to make the amortised costs constant. Provided the potential is always non-negative and initially zero, the accumulated amortised costs will be an upper-bound on the accumulated actual costs [118].

Hofmann and Jost [67] proposed a type-based analysis for heap space usage using amortisation. Instead of extending type judgements with effects as in [38, 73, 127], the analysis of Hoffman and Jost is based on annotating data types with weights representing the relative contribution of parts of a data structure to the overall heap usage (the *potential* associated with the data structure).

The language under analysis is a first order functional notation with a strict semantics and algebraic data types including sums, products, booleans and lists. There are two kinds of pattern-matching deconstructors for heap-allocated values: a deallocating match and non-deallocating match'. The heap cost is defined by a big-step operational semantics instrumented with the size of a free list of heap cells; the free list reduces at each constructor application and grows at each match (but not at match').

The augmented typing judgements take the form $\Gamma,\ k \vdash e : A,\ k'$ where Γ are the type assumptions, e is an expression, A is an annotated type and k, k' are non-negative rational numbers representing the available potential before and after the evaluation of e. The annotations in A together with k and k' give both an upper bound on the initial heap space for evaluation of e and a lower bound on the available heap space after evaluation. For example, the judgement

$$x : \mathsf{L}(\mathsf{L}(\mathsf{B},1),2),\ 3 \vdash e : \mathsf{L}(\mathsf{B},4),\ 5$$

informally says that if x is a list of lists of booleans then e is a list of booleans; furthermore, if $x = [l_1, \ldots, l_n]$ then a free list of size $3 + 2n + 1\sum_i |l_i|$ is sufficient to evaluate e; and if e evaluates to a list $[b_1, \ldots, b_m]$ of length m, the resulting free list will have size at least $5 + 4m$.

From this example we can see that type annotations play a very different role here than in the sized type systems: in the system of Hoffman and Jost an annotation represents not a *size*, but the *coefficient* of the heap cost incurred by a part of a data structure. The upper bound on the initial free list is a function of the (unknown) sizes of the input. Note also that the lower bound on the final free list size is a function of the (unknown) size of the output and that no input/output size relation is obtained.

The type system of Hofmann and Jost performs an amortised analysis of the size of the free list: the coefficients in types represent the *potential* associated with the data structures; the typing rules constrain the annotations so that the amortised costs for each expression are properly accounted. For example, the typing rules for constructing and deconstructing a list node are:[19]

$$\frac{n \geq \mathsf{SIZE}(A \otimes \mathsf{L}(A,k)) + k + n'}{\Gamma, x_h : A, x_t : \mathsf{L}(A,k), n \vdash \mathsf{cons}(x_h, x_t) : \mathsf{L}(A,k), n'} \tag{2.8.1}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e_1 : C, n' \\ \Gamma, x_h : A, x_t : \mathsf{L}(A,k), n + \mathsf{SIZE}(A \otimes \mathsf{L}(A,k)) + k \vdash e_2 : C, n'\end{array}}{\begin{array}{rl}\Gamma, x : \mathsf{L}(A,k), n \vdash \mathsf{match}\ x\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow e_1 \\ & \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2\end{array} : C, n'} \tag{2.8.2}$$

Rule (2.8.1) specifies that the available potential n must be at least the amortised cost of cons, that is, the actual heap cells used (given by the SIZE function) plus the potential k associated with the list

[19] Following Hofmann and Jost [67] and without loss of generality, we present the type rules for expressions in let normal form.

elements (because the list length is increased by one). Dually, rule (2.8.2) specifies that the available potential at the cons alternative increases by the amortised cost (because match does deallocation).

Hofmann and Jost presented an algorithm that automatically infers the type annotations. Their technique associates each program P with a system of linear inequalities $\mathcal{L}(P)$ such that the valid annotated type derivations for P correspond to the admissible solutions of $\mathcal{L}(P)$; these solutions can be obtained by a standard linear programming solvers.

The worst-case theoretical complexity for solving linear programs is polynomial; the variants of the simplex algorithm used in solver implementations, although exponential in the worst-case, is quite efficient in practice. This compares favourably with the sized type systems [23, 73, 74] where type checking alone requires checking validity of Presburger constraints with doubly-exponential worst-case time.

Since annotations represent coefficients of the potential function, the system can only derive heap bounds that are linear on the sizes of data structures. However, since the language implements deallocation using destructive matching, it is still expressive enough to obtain heap costs for many list processing functions, including insertion algorithms such as insertion sort and quicksort.[20] Unlike the sized type analysis of Hughes and Pareto [73], the amortised analysis deals with the irregular divide-and-conquer recursions by "splitting" the potentials between the two recursive calls. Hofmann and Jost also present good results for a binary tree traversal and report successfully analysis of other textbook examples.

One limitation of the analysis of Hofmann and Jost is that the inferred type annotations are sometimes not sufficiently polymorphic because every use of a function shares the same potentials. Consider the identity function $f : \mathsf{L}(\mathsf{B}) \to \mathsf{L}(\mathsf{B})$ on a list of booleans; if a particular use requires the annotation $f : \mathsf{L}(\mathsf{B}, 5), 3 \to \mathsf{L}(\mathsf{B}, 5), 3$ then it not possible to apply f to an argument of type $\mathsf{L}(\mathsf{B}, 0)$. The authors suggest that this can be relaxed by conducting separate analysis for each use of f. However, this implies that is not possible to analyse functions separately from their use, i.e. the analysis is not fully modular.

Hofmann and Jost have considered heap usage but not time or stack usage. Time could, in principle, be treated similarly to heap, by simply recording the number of execution steps instead of the size of a free list. The only difference is the absence of a deallocation mechanism for time costs.

Extending the amortised analysis for stack usage is less straightforward. One technical problem is that a realistic model for stack must employ a small-step rather than a big-step semantics as used in [67]. Another concern is that the bounds expressible by the amortised analysis are linear on the size of data structures (the total number of elements). While this is generally a good match for obtaining heap bounds, for example, it will yield coarse stack bounds for a tree search algorithm whose worst-case complexity is linear on the depth of the tree. A recently submitted PhD thesis investigates the extension of amortised analysis to stack costs; the definition of potential is modified to account the depth of data structures [19].

2.9 Other related work

Hofmann [65] has proposed the use of a linear typing discipline for ensuring that data structures are used in a single-threaded way and can therefore be update in-place. Hofmann further shows that first-order functional programs that admit a linear type in this system can be translated into C-language programs with bounded space behaviour by construction: there are no uses of `malloc()` because all data structures are updated in-place; thus dynamic memory requirements are bounded by the usage of initial data. Of course, this guarantee applies to heap but not to stack.

[20] The sorting algorithms exhibit linear space or even constant bounds by reusing the heap associated with the input list for constructing the sorted list—i.e. they destroy the original list.

Some early works on complexity theory have studied complexity bounds of Turing-incomplete languages. Meyer and Ritchie [104] studied the complexity of bounded loop programs; such programs cannot implement all computable functions but can implement to the first-order primitive-recursive functions on naturals [91, chapter 5]. The complexity bounds are expressed using a family of primitive-recursive functions indexed by the depth of loop nesting and the number of instructions in the program. However, the bounds are rather coarse e.g. a program with a single loop is bounded by a linear function but a program with two nested loops is bounded by an exponential.

Turner [148, 149] proposed a discipline for *strong* functional programming, that is, where program termination is guaranteed by construction. The principal objective is the simpler equational theory resulting from the absence of a "bottom" value associated with partiality. Unlike approaches based e.g. on constructive type theory, Turner proposes an *elementary* discipline that could be used at an introductory programming level; he restricts a pure functional language such Miranda or Haskell by:

a) requiring all case-analysis definitions to be exhaustive;

b) extending all built-in operations to be total (e.g. arithmetic);

c) requiring arguments of recursive calls to be structural sub-components of the formal parameters;

d) requiring recursive data types to be *covariant* (that is, recursion on the left of the arrow type constructor is disallowed).

The resulting programming language is not Turing-complete, but is expressive enough to encode higher-order primitive recursive functions over naturals and other inductive types. To express non-terminating interactions (e.g. an operating system), Turner proposes separating the recursive data types which must be finite (e.g. naturals and lists) from co-recursive ones which are infinite (e.g. streams). Co-recursive definitions must be guarded by co-constructors; this is sufficient to ensure that co-values are productive.[21]

Turner argues that the restriction to primitive recursive definitions captures most useful computable functions. However, algorithms must sometimes be re-written with worse time or space complexity than an equivalent general-recursive formulation; this is undesirable for resource-constrained systems. In any case, we remark that the restriction to a total programming language does *not*, by itself, guarantee resource bounds, except in a naïve extensional sense.

A prerequisite for all cost analysis is to choose a *model of costs*. Most of the previous works [12, 38, 90, 127, 129, 154, 156] chose to count the number of function calls (or the corresponding formal notion of β-reductions in the lambda-calculus). This metric has the advantage of being easily understood by relation with a naive equation rewrite semantics for an applicative language. However, it bears little relation with the time or space costs of a real implementation.

On the theoretical side, Dal Lago and Martini [34] have argued against using the number of β-reductions as a cost model for the lambda-calculus. They proposed a model where the cost of a reduction $M \to N$ is proportional to the difference $|N| - |M|$ between the sizes of *redex* and *reduct* and prove that it satisfies a polynomial-invariance result, i.e. that it can be simulated by a Turing machine within a polynomial-time bound overhead and vice-versa (unlike the cost model based on β-reductions).

Hope and Hutton [72] proposed counting the reduction steps of an abstract machine. Such a model stands half-way between the very abstract measure (number of β-reduction) and measuring real-time

[21] This is similar to the size type system of Hughes et al. [74]; the latter, however, ensures termination and productivity by a *semantic* properties of sizes rather than *syntactical* restrictions.

of a concrete implementation. Hope and Hutton follow the methodology of Ager et al. [1] and Danvy [37]: starting from a denotational evaluator for the language, they apply a sequence of meaning-preserving program transformations to obtain an abstract machine interpreter; this interpreter is then straightforwardly extended with a step-counter; finally, the program transformations are reversed to get back a cost-instrumented denotational evaluator. The principal strength of this approach is that the program transformations are calculated, thus giving a constructive methodology for reasoning about costs of an implementation at the source level.

Bonenfant et al. [14] conducted worst-case execution time (WCET) analysis to obtain bounds on real-time costs for a subset of the abstract machine instructions of *Hume*, a functionally-inspired research language for resource-sensitive systems. Their approach is to translate the abstract machine instructions into C and use a C compiler to obtain machine code; they then employ *aiT*, a commercial tool for static WCET analysis of machine code blocks [41, 42]. Unlike approaches based on experimental tests, the *aiT* tool uses abstract interpretation to model cache and pipeline states of specific microprocessors and is capable of obtaining *guaranteed* worst-case time bounds. Bonenfant et al. applied this tool to derive WCET costs of compiled code for a Renesas M32C/85 micro-controller, compared the results with experimental timings and report a close match with the analysis bounds.

2.10 Worst-case Execution Time Analysis

Worst-case execution time (WCET) analysis is required for a variety of embedded systems applications, especially those with safety- or mission-critical aspects. Common examples include avionics software and autonomous vehicle control systems [137]. Our work aims to construct *fully automatic* source-level static WCET analyses, that are correlated to actual execution costs. Since we must provide formal, automatically-produced *guarantees* on WCET bounds, we base our work on a high-quality abstract interpretation approach (AbsInt GmbH's **aiT** tool [45]), to give low-level timing information for bytecode instructions. We combine this with an equally formal, *type-based* approach that lifts this information to higher-level language constructs so that it can be applied to source programs. The problem is to maintain the strong WCET guarantees we need, while giving good quality information. Our approach to constructing WCET analyses is based on the idea of *amortisation* [142]. This represents the first attempt of which we are aware to provide an automatic amortised WCET analysis. We have produced a prototype implementation using our approach, and we report here on some preliminary results obtained using this analysis tool.

Amortised cost approaches [26] allow costs to be averaged according to use. The basic intuition is that by amortising over the time costs incurred by common usage patterns (e.g. that for a stack, every *pop* is balanced by a *push*), we can construct timings that reflect more accurately real worst-case times. Typically, amortised analysis is performed by hand to determine the complexity of programs that involve complex data structures [117]. We have previously, however, applied the approach to give automatically derived, and provably correct, upper bounds on space costs for heap allocations [66]. In both cases, since alternative program execution paths may have very different costs, by amortising over common patterns we can avoid the needless over-estimation that would otherwise occur.

2.10.1 WCET Analyses for Conventional Programming Languages

Typical WCET analyses, such as **aiT** or **bound-T** work at the low level, operating on relatively simple C or assembler code fragments rather than the high level sources we have shown here. In finding WCET solutions for concrete programs, it is usually necessary for the programmer to provide additional detailed information in the form of specific program annotations, and this make require significant effort in some

cases. For example, it may be necessary to indicate the range of values that a loop variable may take if the associated iteration is not bounded by a literal value.

As described above, our work is based on an automatic analysis that exploits amortisation over data structures [116] to provide linear cost bounds [68] for programs involving boxes, recursion, higher-order functions and the other high-level language features that we exploit in the domain-specific Hume design. When using general-purpose programming notations for embedded systems, however, it is not yet possible to use such technically advanced analyses, and manual resource analysis based on profiling and/or manual code inspection has therefore long formed the state-of-the-art in embedded systems.

Recently, static program analyses for certain resource properties have matured to encompass some industrial applications. For example, stack consumption of *non-recursive* programs is now well understood (e.g. [126, 151]), and has been industrially applied in the form of AbsInt's **StackAnalyzer** tool[22]. A variety of both commercial and academic tools also exist for calculating guaranteed bounds on worst-case execution time [161], including **aiT** [45], **bound-T** [71], **SWEET** [93, 133], **Chronos** [109], **Heptane** and **OTAWA**. Such tools typically work on machine-code fragments, yielding analyses for specific input cases. Of these tools, AbsInt's **aiT** achieves both the best coverage and the best quality, being able to produce WCET bounds that are within 7-8% of the measured WCET for the standard WCET benchmark suite [141] (somewhat tighter than the 22% we have measured as an average discrepancy on our [perhaps more complicated] test cases). In comparison, Synchron gives results that are 80%-90% of the measured WCET for the same benchmark suite. Qualitative results are not available for the other WCET systems.

[22] `http://www.absint.com/stackanalyzer`

Chapter 3

The Hume Abstract Machine

Kevin Hammond and Hans-Wolfgang Loidl

Abstract

This chapter provides a formal specification of the Hume Abstract Machine (HAM) semantics. The specification contains instructions to support higher-order functions and exceptions, but no explicit support for timeouts.

3.1 Introduction

For the execution of Hume [58] programs we define an abstract machine, the Hume Abstract Machine (HAM). This definition is an extension of the initial design, described in [53], by constructs for higher-order functions and exceptions. Additionally, we formally specify the components of the machine and its behaviour in the form of a 2-level, small-step, operational semantics and give a reference implementation of the instructions of the HAM. The operational semantics uses the approach of resource algebras, which we have developed in a previous project [4], to collect information on the resource consumption during execution. The resource algebras are designed in a modular way and can be instantiated without modifying the rules of the operational semantics. We define the cost model for the HAM by giving resource algebras for stack space, heap space and time consumption. The values for stack and heap space are independent of the underlying processor. For obtaining tight bounds on execution time we have used the aiT tool [45] of the AbsInt project partner.

Together with the Hume formal semantics (Chapter 6), and the formal description of the translation of Hume to HAM (Chapter 4), it gives a description about the costs for executing Hume and HAM programs. This is in turn a prerequisite for developing static analyses of resource consumption. This formal specification is also the basis for ongoing work on the certification of the resource consumption of Hume code.

3.2 Hume Abstract Machine Design and Reference Implementation

name	interpretation
S	stack
H	heap
sp	stack pointer
hp	heap pointer
fp	frame pointer
slp	function frame pointer
mp	match pointer
inp	input pointer
rs	current ruleset
$base$	base ruleset

name	interpretation
pc	program counter
pcr	restart program counter
$blocked$	box blocked
$blockedon$	output on which blocked
$INITPC$	initial program counter
ins	input buffers
$outs$	output buffers
$nIns$	number of inputs
$nOuts$	number of outputs

Figure 3.1: Box-specific registers, constants and memory areas — the *box state record*

name	interpretation
$rules$	array of rule entry points
$nRules$	number of rules
rp	current rule pointer

Figure 3.2: Ruleset-specific registers and constants

The goal of the Hume Abstract Machine (HAM) design is to provide a credible basis for research into bounded time and space computation, allowing formal cost models to be verified against a realistic implementation. Absolute space- and time-performance (while an important long-term objective for

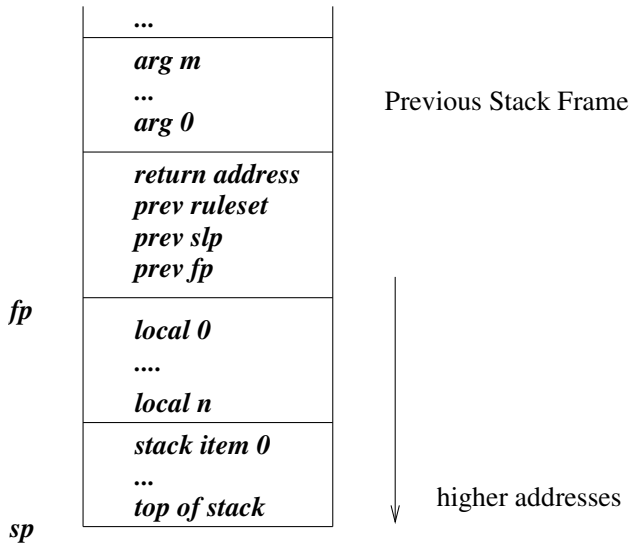


Figure 3.3: Stack frame layout in the Hume Abstract Machine

Hume) is thus less important in this initial design than predictability, simplicity and ease of implementation.

The Hume Abstract Machine is loosely based on the design of the classical G-Machine [6] or SECD-Machine [88]. with extensions to manage concurrency and asynchronicity. Each Hume box is implemented as a thread with its own dynamic stack (S) and heap (H) and associated stack and heap pointers (*sp* and *hp*). These and the other items that form part of the individual *state record* are shown in Figure 3.1. Each function and box has an associated *ruleset* (Figure 3.2). The ruleset is used for two purposes: it gives the address of the next rule to try if matching fails; and it is used to reorder rules if fair matching is specified. The box ruleset is specified as the *base* field of the state record. Function rulesets are set as part of a function call.

Separate stacks are needed to maintain independent state records. Separate heaps allow a simple model of garbage collection where the entire heap becomes garbage each time a box completes. Small pointer ranges can be used in both cases (8-bit stack and heap pointers are possible for a number of applications). This helps conserve space. The corresponding disadvantage of this design is the need to communicate arguments and results between boxes rather than using a physically or virtually shared heap. This is achieved in the HAM reference implementation by copying such values between heaps at the beginning and end of the box execution. There is an analogy with the *working copies* of global variables that may be obtained by implementations of the JVM [92]. However, variable accesses in the JVM may occur at any point during thread execution, not only at the beginning/end as in the HAM. Moreover, unlike the HAM, which is stateless, the JVM maintains a virtually shared heap containing *master copies* of each variable. Our design is thus closer to that of Eden [18]: a reactive functional language based on Haskell.

The layout of a typical stack frame is shown in Figure 3.3. The abstract machine design uses a pure stack calling convention. Function arguments are followed by a four-item frame-header containing the return address, a pointer to the previous ruleset, the static link pointer and the previous frame pointer. In this description, the size of this subframe is given by the constant $\mathcal{S}_{frame}$. The local frame pointer fp points immediately after the frame-header, to the address of the first local variable. The function frame pointer *slp* points to the beginning of the nearest frame representing a function (note that let etc instructions in Hume also allocate frames). For consistency, the same layout is used at the outer

```
for i = 1 to nBoxes do
   runnable := false;
   for j = 1 to box[i].base.nRules do
      if ¬ runnable then
         runnable := true;
         for k = 1 to box[i].nIns do
            runnable &= box[i].required[j,k] ⇒ box[i].ins[k].available
         endfor
      endif
   endfor
   if runnable then schedule (box[i]) endif
endfor
```

Figure 3.4: Scheduling Algorithm

box level. In this case, the box inputs are stored in the argument position, and the return address item is redundant. Note that all values on the stack other than the saved return address, ruleset and frame pointer are pointers to the local heap (i.e. they are *boxed* [123]): in the current design there is no separate *basic value stack* to handle scalar values as in some versions of the G-Machine [123], STG-Machine [122] etc. nor are scalars and heap objects mixed on the stack as in the JVM [92] or recent versions of the STG-Machine.

3.2.1 Box Scheduling

The implementation maintains a vector of boxes, each with its own state record. Boxes are connected through *wires*, which are shared communication buffers each connecting an *in* of one box to an *out* of another (or the same) box. Each wire comprises a pair of a value (*value*) and a flag indicating that the value is valid (*available*), used to ensure correct locking.

Boxes are scheduled under the control of a built-in scheduler. The exact scheduling order is not fixed by the HAM semantics. Implementations are free to realise any order that satisfies the conditions stated in Section 6.1.4. A box is deemed to be *runnable* if all the required inputs are available for any

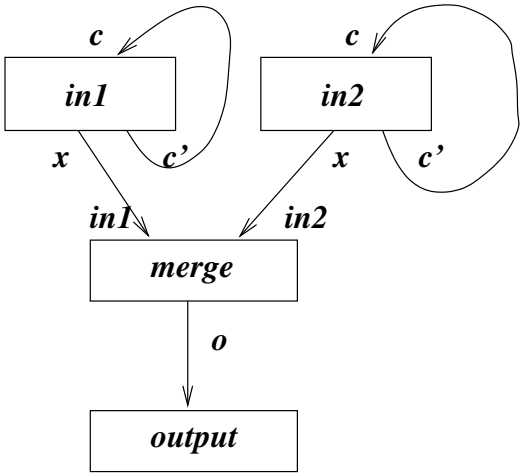


Figure 3.5: Hume example program as a network of boxes

constant	value (words)	constant	value (words)
$\mathcal{H}_{\mathsf{Int}}$	2	$\mathcal{H}_{\mathsf{Constr}}$	2
$\mathcal{H}_{\mathsf{Float}}$	2	$\mathcal{H}_{\mathsf{Tuple}}$	2
$\mathcal{H}_{\mathsf{Bool}}$	2	$\mathcal{H}_{\mathrm{f}}$	4
$\mathcal{H}_{\mathsf{Char}}$	2	$\mathcal{H}_{\mathsf{Chain}}$	4
$\mathcal{H}_{\mathsf{Str}}$	2	$\mathcal{H}_{\mathsf{Vec}}$	2
$\mathcal{S}_{saved}$	4	$\mathcal{H}_{\mathsf{Exn}}$	2

Table 3.1: Sizes of headers for heap objects in the prototype Hume Abstract Machine

of its rules to be executed (Figure 3.4). A compiler-specified matrix is used to determine whether an input is needed: for some box t, $box[t].required[r, i]$ is true if input i is required to run rule r of that box. A single execution cycle comprises the following (all realised via explicit HAM instructions):

a) initialise stack- and heap-pointers and the base ruleset;

b) check input availability against possible matches;

c) copy data from input wires into the local heap for matching;

d) match available inputs against rules;

e) consume those inputs that have been matched and which are not ignored in the selected rule;

f) create a stack frame to hold local variables;

g) bind variables to input values;

h) evaluate the RHS of the selected rule;

i) check that outputs can be written to all output wires;

j) write non-ignored outputs to the corresponding wires;

k) reorder match rules according to the fairness criteria.

Note that all wires are single-buffered. A box will therefore block when writing to a wire which contains an output that has not yet been consumed. In order to ensure a consistent semantics, a single check is performed on all output wires just before any output is written. The check ignores * output positions. The box suspends if any of the needed output wires is occupied.

3.2.2 Heap Representations

In the prototype design, all heap cells are *boxed* with tags distinguishing different kinds of objects (see Figure 3.6. Furthermore, tuple (Tuple), constructor (Constr) and exception structures (Exn) require *size* fields. In the case of constructors the *constructor tag* field (c) is encoded together with Constr tag itself. Function closures (f) contain a pointer to an instruction sequence, the number of needed (m) and of already provided (p) arguments, and a list of pointers to those provided arguments. All data objects in a structure are referenced by pointer. There is one special representation: strings are represented as a tagged sequence of bytes. Clearly, heap usage could be reduced using a more compact representation such as that used by modern high-performance implementations such as SML-NJ [100] or the STG-Machine [122]. A variant of the HAM, using an unboxed representation, has been produced since

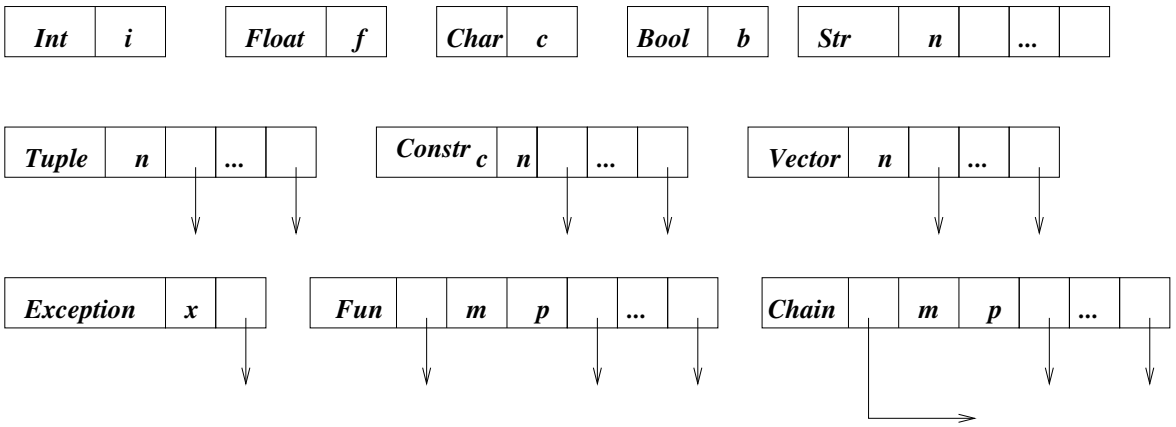


Figure 3.6: Heap representations in the Hume Abstract Machine

project start. For now, we are, however, primarily concerned with bounding and predicting memory usage and therefore use the boxed data representation.

Heap and Stack Sizes.

The sizes of the stack and heap spaces associated with each box are fixed at compile-time using a static size analysis to compute the upper bound of stack and heap usage [57]. This size analysis is defined for flat Hume programs containing no recursion (FSM-Hume), and is currently being extended to more general cases [153]. This size analysis has been integrated into our prototype compiler (`phamc`), and results are being used in the back-end. As described in Chapter 8, we have also produced analyses for space and time consumption of Hume programs, based on the principle of amortised costs [83], and developed prototype implementations of these analyses.

Exceptions.

Two forms of exceptions can occur during the execution of HAM code: synchronous and asynchronous exception. While synchronous exceptions are related to the execution of a particular piece of HAM code (e.g. division-by-zero), asynchronous exceptions can occur at any point and need to be checked by an external system.

The most important synchronous exceptions are stack- and heap-overflow. Before each (block of) instruction, that involves an increase of the stack- or heap-pointer, a stack- or heap-check is necessary, which makes sure that enough stack- or heap space is available, and if not raises an exception. For the pseudo-code in the reference implementation we assume that such stack- and heap-checks are added for any HAM instruction.

The most important asynchronous exception is a timeout. We assume that the HAM is embedded into a system that provides facilities for setting a timer and for checking when such a timer expires. In this case, the handler code for timeouts in the current box must be executed.

3.2.3 The Abstract Machine Instructions

The abstract machine instructions implement the abstract machine design described above. These instructions are shown in Figures 3.7–3.11. They are classified into stack, heap and control-flow operations, which are fairly standard, and matching and scheduling operations, which reflect the specific nature of the Hume design. The description of the instructions uses two auxiliary functions: *maxVars* calculates the maximum number of variables in a list of patterns; and *labels* generates new labels for a set of function/box rules. Where labels *lt*, *ln* etc. are used, these are assumed to be unique.

MkBool b	$H[hp] = \mathsf{Bool}\ b;\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Bool}};\ \ ++pc$
MkChar c	$H[hp] = \mathsf{Char}\ c;\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Char}};\ \ ++pc$
MkInt i	$H[hp] = \mathsf{Int}\ i;\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Int}};\ \ ++pc$
MkFloat f	$H[hp] = \mathsf{Float}\ f;\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Float}};\ \ ++pc$
MkString s	$H[hp] = \mathsf{Str}\ s;\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Str}} + ssize(s);\ \ ++pc$
MkNone	$H[hp] = \mathsf{None};\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{None}};\ \ ++pc$
MkCon $c\ n$	$H[hp] = \mathsf{Constr}\ c\ n\ (S[sp-1])\ \ldots\ (S[sp-n-1]);\ sp := sp - n;$ $S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Constr}} + n;\ \ ++pc$
MkTuple n	$H[hp] = \mathsf{Tuple}\ n\ (S[sp-1])\ \ldots\ (S[sp-n-1]);\ sp := sp - n;$ $S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Tuple}} + n;\ \ ++pc$
MkFun $l\ m\ p$	$H[hp] = \mathrm{f}\ l\ m\ p\ (S[sp-1])\ \ldots\ (S[sp-p-1]);$ $sp := sp - p;\ S[sp] := hp;\ \ ++sp;\ hp := hp + \mathcal{H}_{\mathrm{f}} + p;\ \ ++pc$

Figure 3.7: Abstract Machine Instructions: Heap operations

Push n	$sp := sp + n;\ \ ++pc$
Pop n	$sp := sp - n;\ \ ++pc$
Slide n SlideVar v SlideVarF $d\ v$	$m := n;$ $m := H[S[fp+v]+1];$ $fp' := fp;$ **while** $d \neq 0$ **do** $fp' := S[fp'-2];--d$ **endwhile**; $m := H[S[fp'+v]+1]$ $S[sp-m-1] := S[sp-1];\ sp := sp - m;\ \ ++pc$
Copy n	$S[sp] := S[sp-n-1];\ \ ++sp;\ \ ++pc$
CopyArg n	$S[sp] := S[fp - \mathcal{S}_{saved} - n - 1];\ \ ++sp;\ \ ++pc$
CreateFrame n	$S[sp] := fp;\ fp := sp + \mathcal{S}_{saved};\ sp := sp + \mathcal{S}_{saved} + n;\ lp := sp;\ \ ++pc$
PushVar v	$S[sp] := S[fp+v];\ \ ++sp;\ \ ++pc$
PushVarF $d\ v$	$fp' := fp;$ **while** $d \neq 0$ **do** $fp' := S[fp'-2];\ --d$ **endwhile**; $S[sp] := S[fp'+v];\ \ ++sp;\ \ ++pc$
MakeVar v	$S[fp+v] := S[sp-1];\ \ --sp;\ \ ++pc$

Figure 3.8: Abstract Machine Instructions: Stack operations

Heap Object Creation (Fig 3.7).

Tagged objects are created in the heap, and pointers to the new object stored on the top of the stack. For scalar values (booleans, characters, integers, floats and strings) the actual value is taken directly from the instruction stream (in the case of a string, this is a pointer into the global string table). The corresponding constructor instructions are **MkBool**, **MkChar**, **MkInt***, **MkFloat*** and **MkString**. The instruction **MkNone** creates a special **None** value in the heap. This is used to prevent the writing of a particular output under dynamic programmer control.

Finally, two instructions build structured values. **MkCon** takes two arguments from the instruction stream, a constructor tag and a number of arguments, and builds the corresponding constructor in the heap using the relevant number of arguments from the stack. **MkTuple** is similar except that it has no constructor tag parameter, and builds a tuple rather than a constructor. **MkFun** takes the top p elements from the stack and builds a function closure in the heap. The construction of an exception closure is encoded in the **Raise** instruction, together with the transfer of control to the exception handler of the current box.

Stack Operations (Fig 3.8).

The abstract machine uses a number of simple stack manipulation operations. **Push** increments the stack pointer by a constant. This is used to create fixed space on the stack. **Pop** decrements the stack pointer. **Slide** pops the stack by a fixed amount, and ensures that the top of stack after the stack is the same as before the pop. **SlideVar** and **SlideVarF** perform the same operation but read the amount for the slide from a local or non-local variable, respectively. This is used, for example, to remove the arguments to a function when it returns. **Copy** duplicates the contents of a stack location relative to the current top of stack. Finally, **CopyArg** copies the specified box or function argument to the top of stack.

Three operations are provided on variables. **PushVar** copies the specified variable to the top of stack. **PushVarF** does the same, but from a non-local target stack frame (specified as the depth d in the frame list and the relative offset v, both given in the instruction). **MakeVar** assigns the value on the top of the stack to the corresponding local variable v.

Control-flow Operations (Fig 3.9).

The abstract machine control instructions are conventional. **Goto** sets the pc to the appropriate instruction. **If** does the same conditionally on the value on the top of the stack. **Call** calls the specified function, saving the current ruleset on the stack for future use. The new ruleset and program counter are derived from the label for the function that is called. **CallVar** and **CallVarF** call the function which is pointed to by the a local or non-local variable, respectively. This variable must point to a f closure in the heap, which contains a pointer to the code to be executed, as well as a list of already provided arguments. **Ap** calls the function pointed to by the top-of-stack element. **CallPrim1** and **CallPrim2** call primitive functions with one or two arguments, respectively.

Matching Operations (Fig 3.10).

Matching is initiated by the **StartMatches** instruction, which sets the program counter to the first rule in the base rule set.

The same matching operations are used both for box inputs and for function arguments. The operations are divided into three sets: the **MatchRule** operation, which initialises the matching for a rule, and matches with a closing **MatchedRule** indicating the end of a matching block; the **MatchAvailable** and **MatchNone** operations which check box input availability; and the value matching operations such as **MatchBool** etc.

Goto l	$pc := l$
If l	**if** $S[sp-1] := true$ **then** $pc := l$ **else** $++pc$ **endif**; $--sp$
Call f	$S[sp++] := pc+1$; $S[sp++] := fn$; $S[sp++] := slp$; $slp := sp+1$; $fn := f.ruleset$; $fn.rp := 0$; $pc := fn.rules[0]$
TailCall $f\ n\ d\ sz$	**while** $d \neq 0$ **do** $fp := S[fp-2]$; $--d$ **endwhile**; **for** i **in** $1..n$ **do** $S[fp - \mathcal{S}_{saved} - i] := S[sp-i]$ **endfor** $sp := fp + sz$; $lp := sp$; $fn := f.ruleset$; $fn.rp := 0$; $pc := fn.rules[0]$
CallVarF $d\ v\ n$ CallVar $v\ n$ Ap n	$\left\{ \begin{array}{ll} \textbf{while}\ d \neq 0\ \textbf{do}\ fp := S[fp-2];\ --d\ \textbf{endwhile};\ c := S[fp+v] & \textbf{if CallVarF} \\ c := S[fp+v] & \textbf{if CallVar} \\ c := S[sp] & \textbf{if Ap} \end{array} \right.$ $f := H[c+1]$; $m := H[c+2]$; $p := H[c+3]$ **if** $(n+p \geq m)$ **then** **for** i **in** $p-1..0$ **do** $S[sp++] := H[c+4+i]$ **endfor**; $S[sp++] := pc$; $S[sp++] := fn$; $S[sp++] := slp$; $fn.rp := 0$; $pc := H[c+1]$; **else** $H[hp] :=$ f $f\ m\ (p+n)\ H[c+4]\ \ldots\ H[c+4+(p-1)]\ S[sp-1]\ \ldots\ S[sp-n-1]$; $hp := hp+p+n+4$; **endif**;
Return	$fp := S[fp-1]$; $slp := S[fp-2]$; $fn := S[fp-3]$; $pc := S[fp-4]$; $sp' := fp - \mathcal{S}_{saved}$; $S[sp'] := S[sp-1]$; $sp := sp'+1$
CallPrim1 p	$S[sp-1] := p\ (S[sp])\ (S[sp-1])$; $++pc$
CallPrim2 p	$S[sp-2] := p\ (S[sp])\ (S[sp-1])\ (S[sp-2])$; $--sp$; $++pc$
Raise x	$H[hp] :=_{\text{Exn}} x\ S[sp]$; $--sp$; $sp := fp := slp := 0$; $S[sp++] := hp$; $hp := hp + \mathcal{H}_{\mathsf{Exn}} + 1$; $sp := sp + \mathcal{S}_{saved}$; $pc := fn.rules[fn.handler]$

Figure 3.9: Abstract Machine Instructions: Control-flow

MatchRule	$mp := fp - \mathbb{S}_{saved} + 1;\ inp := 0;\ pc := fn.rules[fn.rp];\ ++fn.rp;\ sp := lp$
MatchedRule	$sp := lp$
MatchNone	$--mp;\ ++inp;\ ++pc$
MatchAvailable	**if** $ins[inp].available$ **then** $++pc$ **else** $pc := fn.rules[fn.rp]$ **endif**; $inp := inp + 1$
MatchBool b	$--mp;$ **if** $H[S[mp]] \neq$ Bool b **then** $pc := fn.rules[fn.rp]$ **endif**
MatchChar c	$--mp;$ **if** $H[S[mp]] \neq$ Char c **then** $pc := fn.rules[fn.rp]$ **endif**
MatchString s	$--mp;$ **if** $H[S[mp]] \neq$ Str s **then** $pc := fn.rules[fn.rp]$ **endif**
MatchInt i	$--mp;$ **if** $H[S[mp]] \neq$ Int i **then** $pc := fn.rules[fn.rp]$ **endif**
MatchFloat f	$--mp;$ **if** $H[S[mp]] \neq$ Float f **then** $pc := fn.rules[fn.rp]$ **endif**
MatchCon $c\ n$	$--mp;$ **if** $H[S[mp]] \neq$ Constr $c\ n\ xs$ **then** $pc := fn.rules[fn.rp]$ **endif**
MatchTuple n	$--mp;$ **if** $H[S[mp]] \neq$ Tuple $n\ xs$ **then** $pc := fn.rules[fn.rp]$ **endif**
MatchExn x	$--mp;$ **if** $H[S[mp]] \neq$ Exn $x\ xs$ **then** $pc := fn.rules[fn.rp]$ **endif**
Unpack	$--sp;$ **if** $H[S[sp]] =$ Tuple $n\ xs$ **then** $offset := 2$ **else if** $H[S[sp]] =$ Constr $c\ n\ xs$ **then** $offset := 3$; **else if** $H[S[sp]] =$ f $f\ m\ n$ **then** $offset := 4$; **endif**; **for** $i := 0$ **to** $n - 1$ **do** $S[sp] := H[S[sp] + offset + i];\ ++sp$; **endfor**; $++pc$
StartMatches	$pc := base.rules[0]$
Reorder	$n := fn.nrules - 1;\ r := fn.rules[fn.rp];$ **for** $i := fn.rp$ **to** n **do** $fn.rules[i] := fn.rules[i + 1]$ **endfor**; $fn.rules[n] := r;\ ++pc$

Figure 3.10: Abstract Machine Instructions: Rule matching

CopyInput n	$S[sp] := copy\ (ins[n]);\ \ ++sp;\ \ ++pc$
Consume n	$ins[n].available := false; ++pc$
MaybeConsume n	**if** $ins[n].available$ **then** $ins[n].available := false$ **endif**; $++pc$

CheckOutputs	**for** $i\ :=\ 0$ **to** $nouts$ **do**
	if $H[S[sp-i-1]]\ \neq$ None **and** $\neg\ outs[i].available$ **then**
	$blocked := true;\ bon := i;\ pcr := pc;\ reschedule;$
	endif;
	endfor; $++pc$

Write n	$--sp;$ **if** $H[S[sp]]\ \neq$ None **then** $outs[n] := copy(H[S[sp]]);$ **endif**; $++pc$
Input	$H[hp] :=$ Char $getchar;\ S[sp] := hp;\ \ ++sp;$ $hp := hp + \mathcal{H}_{char};\ \ ++pc$
Output	$putvalue(S[sp-1]);\ \ --sp;\ \ ++pc$
Within $h\ t$	$fn.rules[fn.within\_handler] := h; S[sp] := lo(t);; S[sp+1] := hi(t); sp := sp+2$
WithinStackSpace $h\ p$	$fn.rules[fn.withinspace\ \_handler] := h; S[sp+1] := splim; splim := sp+p; sp := sp+1$
WithinHeapSpace $h\ p$	$fn.rules[fn.withinspace\ \_handler] := h; S[sp+1] := hplim; hplim := hp+p; sp := sp+1$
DoneWithin	$unsetTimer()$
RaiseWithin $h\ t$	$setTimer(S[sp], S[sp-1])$
DoneWithinStackSpace h	$splim := S[sp-1]; S[sp-1] := S[sp]; sp := sp-1$
DoneWithinHeapSpace h	$hplim := S[sp-1]; S[sp-1] := S[sp]; sp := sp-1$
Schedule	$reschedule$

Figure 3.11: Abstract Machine Instructions: Scheduling and Wire I/O

Matching takes place against a special stack pointer, the match pointer *mp*, which records the current match position. This is initialised in **MatchRule** to be just above the first argument to the box or function. The input pointer *inp* is also initialised to support input availability checking. Finally, the program counter is initialised to the start of the next rule.

The availability checking operations are used only for box inputs. **MatchAvailable** checks whether the next box input is available. If not, then the entire rule match fails, and the next rule is tried. Otherwise the input pointer *inp* is incremented. **MatchNone** simply increments *inp* without checking input availability. This is used to implement ***** and **_*** in patterns.

The match operations match the heap value pointed to by *mp* against the specified value. Failure means the next rule is tried. Otherwise the next match position is checked. Constructors and tuple matches are applied only to the outer level. Nested matching is achieved by unpacking the arguments to the constructor or tuple onto the stack using the **Unpack** instruction.

Finally, after successful matching, rules may be reordered by **Reorder** if fair matching is required. This is ensured by moving the successful rule to the end of the ruleset. As a consequence, a least recently used policy is implemented.

Scheduling, input and output operations (Fig 3.11).

Box input and output is handled by two sets of operations. The **CopyInput** copies the specified input from the input wire into the heap and places it on the top of the stack prior to matching. If matching

Label l	l labels the next instruction
Function $f\ l_1\ \ldots\ l_n$	Function f has rules at labels $l_1\ \ldots\ l_n$
Box $b\ h\ s\ i\ o\ r$	Box b has heap h , stack s, i inputs, o outputs and r rules
Rule $b\ l_1\ \ldots\ l_n$	Box b has rules at labels $l_1\ \ldots\ l_n$
Require $b\ x_1\ \ldots\ x_n$	Box b requires inputs $x_1\ \ldots\ x_n$
Stream s In/Out $h\ s$	Stream h has heap h and stack s
Wire $wi\ i\ wo\ o\ h$	Wire input $wi.i$ to output $wo.o$ with heap h

Figure 3.12: Abstract Machine Pseudo-Instructions

is successful, input is *consumed* using the **Consume** operation, which resets the availability semaphore for the appropriate input wire, thereby permitting new write operations on that wire.

Output is handled by two similar operations. The **Write** operation writes the value on the top of the stack to the specified output wire. Before this can be done, the **CheckOutputs** operation is used to ensure that all required Write operations will succeed. This is achieved by checking that all output wire buffers are empty (as indicated by the wire's *availability* semaphore). If not, then the box blocks until the value on the wire has been consumed, and the *availability* semaphore has been cleared. If the heap value is **None** (corresponding to ***** on the output), then the **Write** will not actually write anything to the output wire, and the *availability* semaphore is ignored by **CheckOutputs**.

Two operations are provided to manage stream input/output. A special box is attached to each stream input and output device. Executing the **Input** operation blocks the box if no input is available. Otherwise the input is read into the box's heap, and can then be written to its output wire using normal abstract machine operations. The **Output** operation simply writes the value on the top of stack to the appropriate device.

Control is returned to the scheduler either when a box blocks as a consequence of being unable to write some output during the **CheckOutputs** operation, or explicitly when a box terminates as a consequence of exiting the **Schedule** operation. In either case, the scheduler will select a new runnable box to execute. If there is no runnable box, then in the concurrent implementation the system will terminate. In a distributed system, it would be necessary to check for global termination, including outstanding communications that could awaken some box.

Box initialisation (Fig 3.13).

When a box is scheduled, its registers are initialised as shown in Figure 3.13. The initialisation during the wire input initialisation phase is similar except that different base and `INITPC` values are used, corresponding to the *_init* code for the box. Registers other than *blocked* and *pc* are not initialised if the box is restarted after it is blocked. In this case, *blocked* is set to *false* and *pc* is set to *pcr*. The registers *splim* and *hplim* hold the upper bounds for the stack and the heap areas, initialised with the constants `SPLIM` and `HPLIM`. These registers can be temporarily modified by the family of **Within** instructions.

`sp := 0`	stack pointer
`hp := 0`	heap pointer
`fp := 0`	frame pointer
`slp := 0`	function frame pointer
`inp := 0`	in pointer
`fn := base`	code base
`pc := INITPC`	program counter
`blocked := false`	blocker flag
`splim := SPLIM`	end of stack area
`hplim := HPLIM`	end of heap area

Figure 3.13: Box initialisation

Chapter 4

Translation from Hume to HAM

Kevin Hammond, Hans-Wolfgang Loidl and Steffen Jost

Abstract

This chapter gives a formal description of the compilation of Hume programs to Hume Abstract Machine (HAM) instructions. Input to the translation is Hume with higher-order functions and exceptions, but without timeouts. Output are sequences of HAM instructions plus special directives to the HAM for structuring the execution. The translation described here exactly reflects the code-generation part of the Hume-to-HAM compiler (`phamc`) and is therefore a suitable basis for mapping computation costs of Hume to HAM instructions.

4.1 Introduction

One essential feature of Hume is predictability of computation costs for arbitrary Hume expressions [57]. This is a prerequisite for developing static analyses for heap, stack and time consumption of Hume programs, which will be one major task in the project. As a basis for such an analysis this document formally describes the translation process from high-level Hume code down to instructions on the Hume Abstract Machine (HAM). This translation models those optimisations performed by the `phamc` prototype Hume compiler, which intentionally refrains from more aggressive optimisations in order to maintain predictability.

In the remainder of the chapter we first present the abstract syntax of Hume; give a series of translation rules for the components of the Hume language, producing sequences of HAM instructions as output; and present examples of compiling Hume to HAM, taken from the `phamc` compiler.

4.2 Hume Language Structure and Syntax

Hume [58] is a functionally-based research language aimed at applications requiring bounded time and space behaviour, such as real-time embedded systems. The challenge to be met by the Hume design is to preserve the essential properties of costability and low-level interfacing that are required by real-time embedded systems whilst providing as high-level a programming environment as possible.

Figure 4.1 shows the Hume abstract syntax. The language uses a rule-based approach, with a purely functional expression notation embedded in an asynchronous process model. This simplifies both correctness proofs and the construction of cost models at the expression level. Process abstractions ("boxes") specify an asynchronous and stateless mapping of inputs to outputs, which are scheduled whenever required inputs become available. Boxes can be seen as stateless objects with a rigid communication structure, which both assists process/communication costing and simplifies the construction of deadlock/termination proofs. They are wired explicitly into a static process network (again simplifying both correctness and costing) using a single-buffer approach. Single-buffering allows tight controls over buffer sizes based on the types of values that are communicated, gives a simple and easily implemented semantics, and can be extended to multiple buffering stages by simply introducing additional intermediary boxes. Boxes are activated whenever required inputs become available, write the outputs they produce to the corresponding buffers, and then suspend. In this way, we achieve time- or event-triggered process activation, as well as repetition at the box level.

The expression layer of Hume provides several levels of increasing expressive power to program the behaviour of a box. In this document we use higher-order Hume, including exceptions. Therefore, functions can be passed to other functions, be stored in variables and partial application is permitted. Exceptions are permitted to transfer control to the exception handler attached to the current box. However, at present we do not model timeouts, which are still subject to minor changes in the design.

For example, we can define a Hume program to continuously poll standard input, recording the input status:

```
-- check for input within given time

stream input from "std_in";
stream output to "std_out";

data STATUS = WAITING | RECEIVED;

box getinput
in  (i::char)
out (o::char,handshake::STATUS)
```

program ::=	$decl_1$ `;` ... `;` $decl_n$	$n \geq 1$
decl ::=	*box* | id `=` *expr* | id ⟨ $match_1$ `|` ⋯ `|` $match_n$ ⟩	$n \geq 1$
box ::=	`box` id *ins* *outs* `fair`/`unfair` *bmatches* [`handle` *cmatches*]	
ins/outs ::=	⟨ $\mathrm{id}_1, \ldots, \mathrm{id}_n$ ⟩	$n \geq 0$
bmatches ::=	*expr* | ⟨ $bmatch_1$ `|` ⋯ `|` $bmatch_n$ ⟩	$n \geq 1$
cmatches ::=	*exnexpr* | ⟨ $cmatch_1$ `|` ⋯ `|` $cmatch_n$ ⟩	$n \geq 1$
bmatch ::=	⟨ $bpat_1$ `,` ... `,` $bpat_n$ ⟩ `->` *expr*	$n \geq 1$
cmatch ::=	*cpat* `->` *exnexpr*	
match ::=	⟨ pat_1 `,` ... `,` pat_n ⟩ `->` *expr*	$n \geq 1$
exnmatch ::=	⟨ pat_1 `,` ... `,` pat_n ⟩ `->` *exnexpr*	$n \geq 1$
expr ::=	int | float | char | bool | string | `*`	
	| var $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| id $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| con $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| `(` $expr_1$ `,` ... `,` $expr_n$ `)`	$n \geq 2$
	| `if` $expr_1$ `then` $expr_2$ `else` $expr_3$	
	| `case` *expr* `of` ⟨ $match_1$ `|` ⋯ `|` $match_n$ ⟩	$n \geq 1$
	| `let` ⟨ $vdecl_1$ `,` ... `,` $vdecl_n$ ⟩ `in` *expr*	$n \geq 1$
	| *expr* `within` int `time raise` exn`()`	
	| *expr* `within` int `stack raise` exn`()`	
	| *expr* `within` int `heap raise` exn`()`	
	| *raiseexpr*	
raiseexpr ::=	`raise` exn`(`*exnexpr*`)`	
exnexpr ::=	int | float | char | bool | string | `*` | var	
	| con $exnexpr_1$ ⋯ $exnexpr_n$	$n \geq 0$
	| `(` $exnexpr_1$ `,` ... `,` $exnexpr_n$ `)`	$n \geq 2$
	| `if` $exnexpr_1$ `then` $exnexpr_2$ `else` $exnexpr_3$	
	| `case` *exnexpr* `of` ⟨ $exnmatch_1$ `|` ⋯ `|` $exnmatch_n$ ⟩	$n \geq 1$
	| `let` ⟨ $exnvdecl_1$ `,` ... `,` $exnvdecl_n$ ⟩ `in` *exnexpr*	$n \geq 1$
vdecl ::=	var `=` *expr*	
exnvdecl ::=	var `=` *exnexpr*	
bpat ::=	*pat* | `*` | `_*`	
cpat ::=	exn *pat*	
pat ::=	int | float | char | bool | string | `_` | var	
	| con pat_1 ⋯ pat_n	$n \geq 0$
	| `(` pat_1 `,` ... `,` pat_n `)`	$n \geq 2$

Figure 4.1: Hume abstract syntax

```
match
  v -> (v,RECEIVED)
| * -> (*,WAITING);

wire getinput (input) (output,timer.monitor);
```

4.3 Compilation Rules

Figures 4.2–4.7 give rules for compiling Hume abstract syntax forms into the Hume Abstract Machine (HAM) instructions, as a formal compilation scheme similar to that for the G-machine [6]. These rules have been used to construct a compiler (`phamc`) from Hume source code to the HAM.

The compilation scheme uses a simple sequence notation: $\langle\ i_1,\ \ldots,\ i_n\ \rangle$ denotes a sequence of n items. The @ operation concatenates two such sequences. Many rules also use an environment ρ which maps identifiers to $\langle$ *depth*, *offset* $\rangle$ pairs.

Four auxiliary functions are used, but not defined here: *maxVars* calculates the maximum number of variables in a list of patterns; *bindDefs* augments the environment with bindings for the variable definitions taken from a declaration sequence — the *depth* of these new bindings is 0, whilst the depth of existing variable bindings in the environment is incremented by 1; *bindVars* does the same for a sequence of patterns; and *labels* generates new labels for a set of function/box/exception rules. Note that where labels *lt*, *ln*, *lx* etc. are used, these are assumed to be unique in the obvious way: there is at most one **Label** pseudo-instruction for each label in the translated program. Labels for boxes, functions and exception blocks are derived in a standard way from the (unique) name of the box or function.

The rules are structured by abstract syntax class. The rules for translating expressions ($\mathcal{C}_E$ etc. — Figure 4.2) are generally straightforward, but note that function frames are created to deal with *let*-, *case*- and *raise*-expressions, which then exploit the function calling mechanism. In the first two cases, this allows the creation of local stack frames. For *case*-expressions and *raise*-expressions, it allows the exploitation of the standard pattern matching instructions. It would obviously be possible to eliminate the function call for *let*-expressions provided the stack frame was properly set up in order to allow access to non-local definitions. For exceptions, since the transfer of control is permanent, it would be possible to replace the entire stack by the exception value and to use a **Goto** rather than a **Call**. In this case, each translated exception rule would finish with a **Schedule** rather than a **Return**. This has been avoided here purely for reasons of complexity.

The rules for translating box and function declarations are shown in Figure 4.3. These rules create new stack frames for the evaluation of the box or function, label the entry points and introduce appropriate pseudo-instructions. In the case of box declarations, it is also necessary to copy inputs to the stack using **CopyInput** instructions and to deal with fair matching and the exception handlers.

Box bodies are compiled using $\mathcal{C}_R/\mathcal{C}_{R'}$ (Figure 4.5). These rules compile matches for the outer level patterns using $\mathcal{C}_P$, then compile inner pattern matches using $\mathcal{C}_A$, before introducing **Consume** instructions for non-***** input positions, including _*****. The right hand side can now be compiled. If more than one result is to be produced, the tuple of outputs is unpacked onto the stack. A **CheckOutputs** is inserted to verify that the outputs can be written using appropriate **Write** instructions. Finally, a **Reorder** is inserted if needed to deal with fair matching, and a **Schedule** returns control to the

expr

$$
\begin{array}{lcl}
\mathcal{C}_E\ \rho\ (b) & = & \langle\ \text{MkBool}\ b\ \rangle \\
\mathcal{C}_E\ \rho\ (c) & = & \langle\ \text{MkChar}\ c\ \rangle \\
\mathcal{C}_E\ \rho\ (s) & = & \langle\ \text{MkString}\ s\ \rangle \\
\mathcal{C}_E\ \rho\ (i) & = & \langle\ \text{MkInt}\ i\ \rangle \\
\mathcal{C}_E\ \rho\ (f) & = & \langle\ \text{MkFloat}\ f\ \rangle \\
\mathcal{C}_E\ \rho\ (*) & = & \langle\ \text{MkNone}\ \rangle \\
\mathcal{C}_E\ \rho\ (\ e_1,\ \ldots,\ e_n\ ) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \langle\ \text{MkTuple}\ n\ \rangle \\
\mathcal{C}_E\ \rho\ \text{con}\ e_1,\ \ldots,\ e_n & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \langle\ \text{MkCon con}\ n\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (v) & = & \textbf{let}\ \langle\ d, m\ \rangle\ =\ \rho\ v\ \textbf{in} \\
 & & \textbf{if}\ d\ =\ 0\ \textbf{then}\ \langle\ \text{PushVar}\ m\ \rangle \\
 & & \textbf{else}\ \langle\ \text{PushVarF}\ d\ m\ \rangle\ \textbf{endif} \\
\\
\mathcal{C}_E\ \rho\ (p\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \langle\ \text{CallPrim}^n\ p\ \rangle \\
\mathcal{C}_E\ \rho\ (\text{f}\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @ \\
 & & \textbf{let}\ m =\ \textit{arity}\ \text{f}\ \textbf{in} \\
 & & \textbf{if}\ n\ =\ m\ \textbf{then}\ \langle\ \text{Call f,Slide}\ n\ \rangle \\
 & & \textbf{else if}\ n\ <m\ \textbf{then}\ \langle\ \text{MkFun f}\ m\ n\ \rangle \\
 & & \textbf{else}\ \langle\ \text{Call f,Slide}\ n\ \rangle\ \underbrace{@\ \langle\ \text{Ap 1,\ Slide 2}\ \rangle}_{m-n}\ \textbf{endif} \\
\\
\mathcal{C}_E\ \rho\ (v\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \mathcal{C}_E\ \rho\ v\ @ \\
 & & \textbf{let}\ \langle\ d, m\ \rangle\ =\ \rho\ v\ \textbf{in} \\
 & & \textbf{if}\ d\ =\ 0\ \textbf{then}\ \langle\text{CallVar}\ m\ n,\ \text{SlideVar}\ m\ \rangle \\
 & & \textbf{else}\ \langle\text{CallVarF}\ d\ m\ n,\ \text{SlideVarF}\ d\ m\ \rangle\ \textbf{endif} \\
\\
\mathcal{C}_E\ \rho\ (\textbf{if}\ c\ \textbf{then}\ t\ \textbf{else}\ f) & = & \mathcal{C}_E\ \rho\ c\ @\ \langle\ \text{If}\ lt\ \rangle\ @\ \mathcal{C}_E\ \rho\ f\ @ \\
 & & \langle\ \text{Goto}\ ln,\ \text{Label}\ lt\ \rangle\ @\ \mathcal{C}_E\ \rho\ t\ @ \\
 & & \langle\ \text{Label}\ ln\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (\textbf{case}\ e\ \textbf{of}\ ms) & = & \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{Call}\ lc,\text{Slide}\ 1,\text{Goto}\ ln,\text{Label}\ lc\ \rangle\ @ \\
 & & \mathcal{C}_{Case}\ \rho\ ms\ @ \\
 & & \langle\ \text{Label}\ ln,\text{Function}\ lc\ (\textit{labels}\ lc)\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (\textbf{let}\ d_1\ \ldots\ d_n\ \textbf{in}\ e) & = & \textbf{let}\ \rho'\ =\ \textit{bindDefs}\ \langle\ d_1,\ \ldots,\ d_n\ \rangle\ \rho\ \textbf{in} \\
 & & \langle\ \text{Call}\ ll,\text{Goto}\ ln,\text{Label}\ ll,\text{CreateFrame}\ n\ \rangle\ @ \\
 & & \mathcal{C}_{Let}\ \rho\ 0\ d_1\ @\ \ldots\ @\ \mathcal{C}_{Let}\ \rho\ (n-1)\ d_n\ @ \\
 & & \mathcal{C}_E\ \rho'\ e\ @\ \langle\ \text{Return,Label}\ ln\ \rangle\ @ \\
 & & \langle\ \text{Function}\ ll\ \langle\rangle\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (\textbf{raise}\ x\ e) & = & \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{Raise}\ x\rangle \\
\\
\mathcal{C}_E\ \rho\ (e\ \textbf{within}\ t\ \texttt{time}\ \ \textbf{raise}\ x) & = & \langle\ \text{Within}\ lx\ t\ \rangle\ @\ \mathcal{C}_E\ \rho\ e\ @ \\
 & & \langle\ \text{Goto}\ ln,\text{Label}\ lx,\text{RaiseWithin,\ MkTuple}\ 0,\ \text{Raise Timeout}\ x,\ \text{Label}\ ln,\ \text{Do} \\
\mathcal{C}_E\ \rho\ (e\ \textbf{within}\ p\ \texttt{stack}\ \ \textbf{raise}\ x) & = & \langle\ \text{WithinStackSpace}\ x\ p\ \rangle\ @\ \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{DoneWithinStackSpace}\ x\ \rangle \\
\mathcal{C}_E\ \rho\ (e\ \textbf{within}\ p\ \texttt{heap}\ \ \textbf{raise}\ x) & = & \langle\ \text{WithinHeapSpace}\ x\ p\ \rangle\ @\ \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{DoneWithinHeapSpace}\ x\ \rangle \\
\\
\mathcal{C}_{Case}\ \rho\ \langle\ r_1,\ldots,r_m\ \rangle & = & \textbf{let}\ n\ =\ \textit{maxVars}\ \langle\ r_1,\ \ldots,\ r_m\ \rangle\ \textbf{in} \\
 & & \langle\ \text{CreateFrame}\ n\ \rangle\ @ \\
 & & \mathcal{C}_F\ \rho\ \langle\ r_1,\ldots,r_m\ \rangle \\
\\
\mathcal{C}_{Let}\ \rho\ n\ (\ id = e\ ) & = & \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{MakeVar}\ n\ \rangle
\end{array}
$$

Figure 4.2: Compilation Rules for Expressions

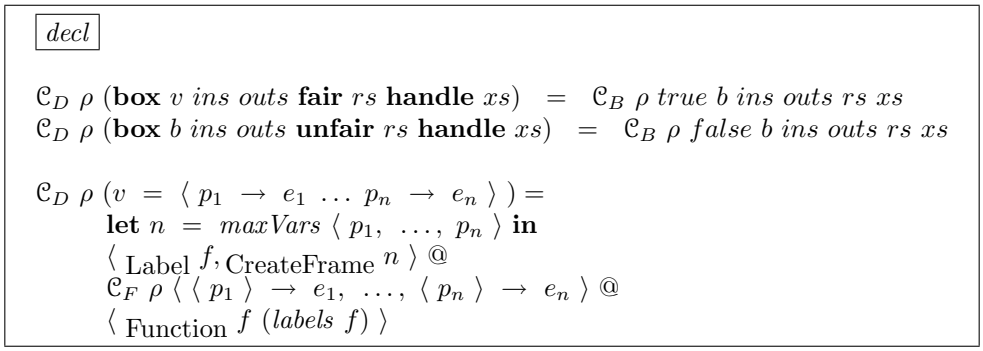


Figure 4.3: Compilation Rules for Declarations

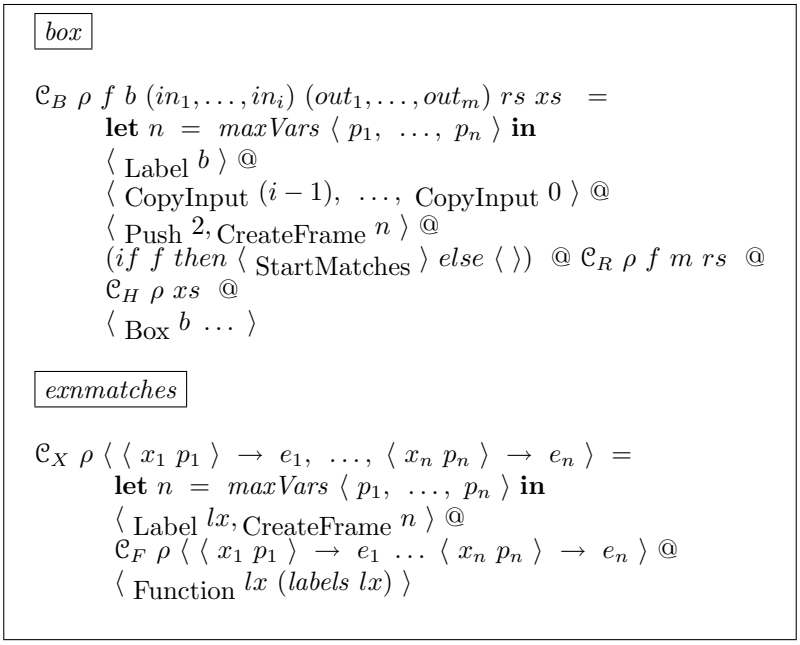


Figure 4.4: Compilation Rules for Declarations, Box Bodies and Exception Handlers

bmatches

$$
\begin{array}{lcl}
\mathcal{C}_R\ \rho\ f\ m\ \langle\ r_1,\ \ldots,\ r_n\ \rangle & = & \mathcal{C}_{R'}\ \rho\ f\ m\ r_1\ @\ \ldots\ @\ \mathcal{C}_{R'}\ \rho\ f\ m\ r_n \\
\\
\mathcal{C}_{R'}\ \rho\ f\ m\ (\ \langle\ p_1,\ \ldots,\ p_n \rangle\ \rightarrow\ e\ ) & = & \langle\ \text{Label}\ lr, \text{MatchRule}\ \rangle\ @ \\
 & & \mathcal{C}_P\ p_1\ @\ \ldots\ @\ \mathcal{C}_P\ p_n\ @ \\
 & & \mathcal{C}_A\ p_1\ @\ \ldots\ @\ \mathcal{C}_A\ p_n\ @ \\
 & & \mathcal{C}_C\ 0\ p_1\ @\ \ldots\ @\ \mathcal{C}_C\ (n-1)\ p_n\ @ \\
 & & \mathcal{C}_E\ \rho\ e\ @ \\
 & & (if\ m\ >\ 1\ then\ \langle\ \text{Unpack}\ \rangle\ else\ \langle\ \rangle)\ @ \\
 & & \langle\ \text{CheckOutputs}\ \rangle\ @ \\
 & & \langle\ \text{Write}\ (n-1)\ \cdots\ \text{Write}\ 0\ \rangle\ @ \\
 & & (if\ f\ then\ \langle\ \text{Reorder}\ \rangle\ else\ \langle\ \rangle)\ @ \\
 & & \langle\ \text{Schedule}\ \rangle
\end{array}
$$

matches

$$
\begin{array}{lcl}
\mathcal{C}_F\ \rho\ \langle\ r_1,\ \ldots,\ r_n\ \rangle & = & \mathcal{C}_{F'}\ \rho\ r_1\ @\ \ldots\ @\ \mathcal{C}_{F'}\ \rho\ r_n \\
\mathcal{C}_{F'}\ \rho\ (\ \langle\ p_1,\ \ldots,\ p_n\ \rangle\ \rightarrow\ e\ ) & = & \textbf{let}\ \rho' = bindVars\ \langle\ p_1\ ,\ \ldots,\ p_n\ \rangle\rho\ \textbf{in} \\
 & & \langle\ \text{Label}\ lf, \text{MatchRule}\ \rangle\ @ \\
 & & \mathcal{C}_{P'}\ p_1\ @\ \ldots\ @\ \mathcal{C}_{P'}\ p_n\ @ \\
 & & \mathcal{C}_A\ p_1\ @\ \ldots\ @\ \mathcal{C}_A\ p_n\ @ \\
 & & \mathcal{C}_E\ \rho'\ e\ @ \\
 & & \langle\ \text{Return}\ \rangle \\
\\
\mathcal{C}_C\ n\ (\textbf{*}) & = & \langle\ \rangle \\
\mathcal{C}_C\ n\ (\_\textbf{*}) & = & \langle\ \text{MaybeConsume}\ n\ \rangle \\
\mathcal{C}_C\ n\ (p) & = & \langle\ \text{Consume}\ n\ \rangle
\end{array}
$$

Figure 4.5: Compilation Rules for Rule Matches, Functions and Exception Handlers

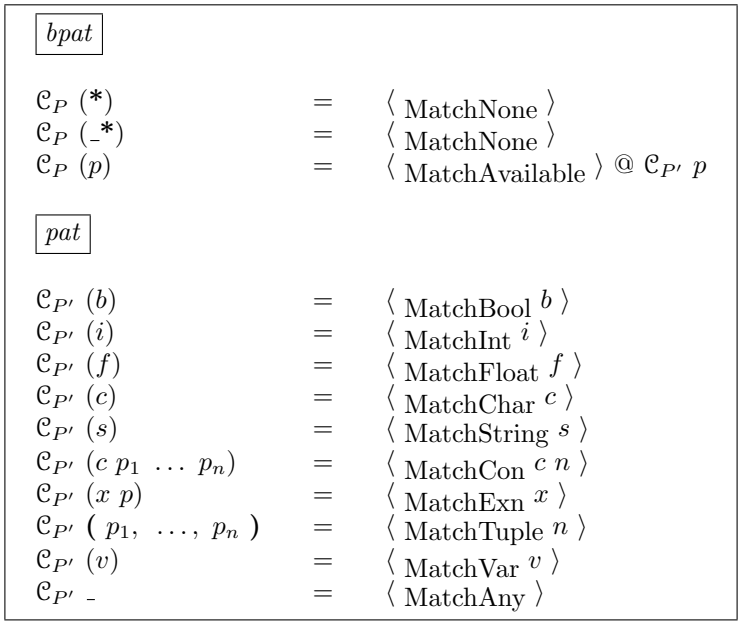


Figure 4.6: Compilation Rules for Patterns

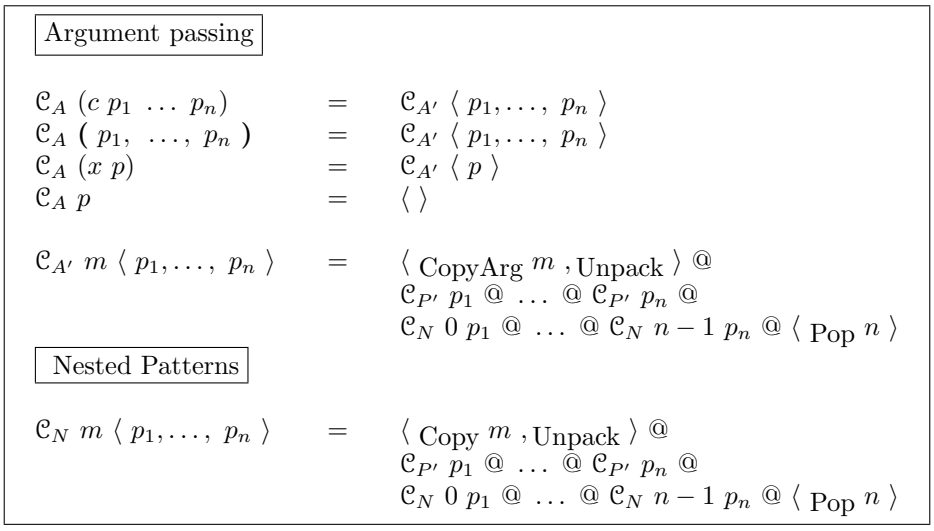


Figure 4.7: Compilation Rules for Argument Passing

scheduler. The compilation of function/handler bodies using $\mathcal{C}_F/\mathcal{C}_{F'}$ is similar, except that $\mathcal{C}_{P'}$ is used rather than $\mathcal{C}_P$, there is no need to deal with box inputs/outputs or fair matching, and a **Return** rather than **Schedule** is inserted at the end of each compiled rule.

Finally patterns are compiled using $\mathcal{C}_P/\mathcal{C}_{P'}$, where $\mathcal{C}_P$ inserts the **MatchNone**/ **MatchAvailable** instructions that are needed at the box level, and $\mathcal{C}_{P'}$ compiles simple patterns. Constructed values are matched in two stages: firstly the outer part (the constructor, tuple or exception) is matched, and then if the match is successful, the matched object is deconstructed on the stack to allow its inner components to be matched against the inner patterns. These nested patterns are compiled using $\mathcal{C}_A$ and $\mathcal{C}_N$. $\mathcal{C}_A$ inserts **CopyArg** and **Unpack** instructions to decompose function/box arguments, where $\mathcal{C}_N$ deals with the general nested case using **Copy** instructions to replicate items that are in the local stack frame.

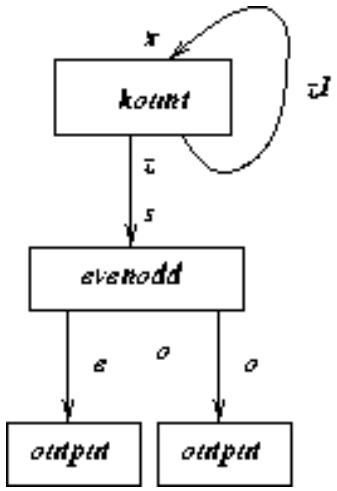


Figure 4.8: Even/odd example

4.4 Example code

This section presents example Hume programs and the HAM code generated by the `phamc` compiler.

4.4.1 Even/Odd

The following simple example program implements a counter, which feeds into an `evenodd` box, that outputs even numbers to the left and odd numbers to the right output wire (see Figure 4.8).

Hume code

```
program

stream outputE to "std_out";
stream outputO to "std_out";

type integer = int 32;

even s = s mod 2 == 0;

box kount
in (x::integer)
out (z::integer, z1::integer)
match
 (x) -> (x,x+1);

box evenodd
in  (s :: integer)
out (e :: integer, o :: integer)
match
 (x) -> if (even x)
          then (x,*)
          else (*,x) ;

box outE
```

```
in  (x :: integer)
out (z :: string)
match
 (x) -> ("EVEN " ++ x as string ++ "\n") ;

box outO
in  (x :: integer)
out (z :: string)
match
 (x) -> ("ODD " ++ x as string ++ "\n") ;

wire kount (kount.z1 initially 1) (evenodd.s, kount.x);
wire evenodd (kount.z) (outE.x, outO.x);
wire outE (evenodd.e) (outputE);
wire outO (evenodd.o) (outputO);
```

HAM code

```
Label "kount"
CopyInput 0
Push 3
CreateFrame 1

Label "kount_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
MkInt 1
PushVar 0
CallPrim "+"
PushVar 0
MkTuple 2
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "evenodd"
CopyInput 0
Push 3
CreateFrame 1

Label "evenodd_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
PushVar 0
Call "f_even"
Slide 1
```

```
If "t_evenodd_0_0"
PushVar 0
MkNone
MkTuple 2
Goto "n_evenodd_0_0"

Label "t_evenodd_0_0"
MkNone
PushVar 0
MkTuple 2

Label "n_evenodd_0_0"
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "outE"
CopyInput 0
Push 3
CreateFrame 1

Label "outE_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
MkString "\n"
PushVar 0
CallPrim "show"
CallPrim "++"
MkString "EVEN "
CallPrim "++"
CheckOutputs
Write 0
Schedule

Label "outO"
CopyInput 0
Push 3
CreateFrame 1

Label "outO_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
MkString "\n"
PushVar 0
CallPrim "show"
CallPrim "++"
MkString "ODD "
```

```
CallPrim "++"
CheckOutputs
Write 0
Schedule

Label "f_even"
CreateFrame 1

Label "f_even_0"
MatchRule
MatchVar 0
MatchedRule
MkInt 0
MkInt 2
PushVar 0
CallPrim "mod"
CallPrim "=="
Return
Function "f_even" "f_even_0"

Box "kount" "kount" 10 8 1 2 1 "kount_init" "kount_handler" NullT
Rule "kount" "kount_0"
Require "kount" True

Box "evenodd" "evenodd" 15 17 1 2 1 "evenodd_init" "evenodd_handler" NullT
Rule "evenodd" "evenodd_0"
Require "evenodd" True

Box "outE" "outE" 26 8 1 1 1 "outE_init" "outE_handler" NullT
Rule "outE" "outE_0"
Require "outE" True

Box "outO" "outO" 26 8 1 1 1 "outO_init" "outO_handler" NullT
Rule "outO" "outO_0"
Require "outO" True

Label "evenodd_init"
Schedule

Label "kount_init"
MkInt 1
Write 1
Schedule

Label "outE_init"
Schedule

Label "outO_init"
Schedule
Stream "outputE" Out "s_write" "std_out" 50 1 0 NullT
Stream "outputO" Out "s_write" "std_out" 50 1 0 NullT
Wire "outputE" 0 "outputE" 0 50 0 NullT
Wire "outputO" 0 "outputO" 0 50 0 NullT

Label "s_read"
```

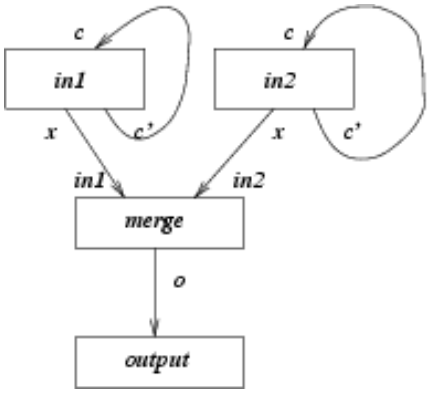


Figure 4.9: Wiring threads

```
Input
Write 0
Schedule

Label "s_write"
CopyInput 0
Consume 0
Output
Schedule

Label "s_timeout"
MkTuple 0
Raise "Timeout"

Label "s_soverflow"
MkTuple 0
Raise "StackOverflow"

Label "s_hoverflow"
MkTuple 0
Raise "HeapOverflow"
Wire "kount" 0 "kount" 1 2 0 NullT
Wire "evenodd" 0 "kount" 0 2 0 NullT
Wire "outE" 0 "evenodd" 0 2 0 NullT
Wire "outputE" 0 "outE" 0 2 0 NullT
Wire "outO" 0 "evenodd" 1 2 0 NullT
Wire "outputO" 0 "outO" 0 2 0 NullT
```

4.4.2 Fair Merge

This example realises a fair merge operation on two input streams, as depicted in Figure 4.9.

Hume code

```
program

stream output to "std_out";
```

```
type integer = int 32;

box k1
in  (x::integer)
out (z::integer, z1::integer)
match
 (x) -> (x,x);

box k2
in  (x::integer)
out (z::integer, z1::integer)
match
 (x) -> (x,x);

box merge
in  (i1, i2 :: integer)
out (o :: integer)
fair
  (x,*) -> x
| (*,y) -> y;

wire k1 (k1.z1 initially 1) (merge.i1, k1.x);
wire k2 (k2.z1 initially 2) (merge.i2, k2.x);
wire merge (k1.z, k2.z) (output);
```

HAM code

```
Label "k1"
CopyInput 0
Push 3
CreateFrame 1

Label "k1_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
PushVar 0
PushVar 0
MkTuple 2
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "k2"
CopyInput 0
Push 3
CreateFrame 1

Label "k2_0"
MatchRule
MatchAvailable
```

```
MatchVar 0
Consume 0
MatchedRule
PushVar 0
PushVar 0
MkTuple 2
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "merge"
CopyInput 1
CopyInput 0
Push 3
CreateFrame 1
StartMatches

Label "merge_0"
MatchRule
MatchAvailable
MatchVar 0
MatchNone
Consume 0
MatchedRule
PushVar 0
CheckOutputs
Write 0
Reorder
Schedule

Label "merge_1"
MatchRule
MatchNone
MatchAvailable
MatchVar 0
Consume 1
MatchedRule
PushVar 0
CheckOutputs
Write 0
Reorder
Schedule

Box "k1" "k1" 6 8 1 2 1 "k1_init" "k1_handler" NullT
Rule "k1" "k1_0"
Require "k1" True

Box "k2" "k2" 6 8 1 2 1 "k2_init" "k2_handler" NullT
Rule "k2" "k2_0"
Require "k2" True

Box "merge" "merge" 5 8 2 1 2 "merge_init" "merge_handler" NullT
Rule "merge" "merge_0" "merge_1"
```

```
Require "merge" True False
Require "merge" False True

Label "k1_init"
MkInt 1
Write 1
Schedule

Label "k2_init"
MkInt 2
Write 1
Schedule

Label "merge_init"
Schedule
Stream "output" Out "s_write" "std_out" 2 1 0 NullT
Wire "output" 0 "output" 0 2 0 NullT

Label "s_read"
Input
Write 0
Schedule

Label "s_write"
CopyInput 0
Consume 0
Output
Schedule

Label "s_timeout"
MkTuple 0
Raise "Timeout"

Label "s_soverflow"
MkTuple 0
Raise "StackOverflow"

Label "s_hoverflow"
MkTuple 0
Raise "HeapOverflow"
Wire "k1" 0 "k1" 1 2 0 NullT
Wire "k2" 0 "k2" 1 2 0 NullT
Wire "merge" 0 "k1" 0 2 0 NullT
Wire "merge" 1 "k2" 0 2 0 NullT
Wire "output" 0 "merge" 0 2 0 NullT
```

Chapter 5

High-Level Cost Model for Hume Programs

Kevin Hammond and Hans-Wolfgang Loidl

Abstract

This chapter gives a cost model for execution time, stack and heap space consumption for Hume. The cost models are described as resource algebras that can be freely added to the HAM specification. The latter is encoded as a 2-level, small step operational semantics.

5.1 Introduction

In Chapter 3, we gave a formal defininition of the Hume Abstract Machine (HAM) for executing Hume programs. This definition is an extension of the initial design, described in [53], by constructs for higher-order functions and exceptions. We define the cost model for the HAM by giving resource algebras for stack space, heap space and time consumption. The values for stack and heap space are independent of the underlying processor. For obtaining tight bounds on execution time we have used the aiT tool [45] of the AbsInt project partner.

The structure of this document is as follows. Section 3.2 describes the HAM design in general, the data structures used and the behaviour of the machine by presenting a reference implementation of the HAM instructions in pseudo-C. Section 5.2 describes the concept of a resource algebra and instantiates it for stack space, heap space and time consumption. Section 6.1 provides a formal specification of the HAM as a 2-level, small-step operational semantics. Finally, Section 6.2 summarises.

5.2 Cost Modelling via Resource Algebras

As the basis for reasoning about resource consumption in Hume programs, we now present cost models of the HAM for heap space, stack space and time consumption. One of our main design goals is to separate the tracking of these resources as far as possible from the description of the functional behaviour of the abstract machine. We therefore use the concept of *resource algebras* [4] as introduced in the MRG project.

A *resource algebra* $\mathcal{R}$ is a partially ordered monoid $(R, 0, +, \leq)$, i.e. $(R, 0, +)$ is a monoid and $(R, \leq)$ is a partially ordered set, where 0 is the minimum element, and + is order preserving on both sides. Moreover, $\mathcal{R}$ has constants in R for each expression former of the language. These constants denote the costs of the corresponding expression. In general, these constants are parameterised with the current state of the computation, in particular the current stack and heap. These costs are combined by the rules of the operational semantics in Section 6.1 with the + operator.

5.2.1 A Resource Algebra for Heap Space

In instantiating the concept of a resource algebra with heap costs, we use the natural numbers $\mathbb{N}$ as domain R, with addition as +, the natural number zero as 0 and the less-or-equal relation as $\leq$. Table 5.1 defines the changes in heap size for all instructions in the HAM language. The tables $\mathcal{P}rim1_{s,\mathcal{H},f}$ and $\mathcal{P}rim2_{s,\mathcal{H},f}$ define the heap costs for unary and binary primitive operations, with the current stack s, heap $\mathcal{H}$, and the primitive function f as arguments.

The heap consumption of the constructors is defined by the constants in Table 3.1. The notation $|\cdot|$ represents the number of arguments in a tuple, vector, string etc. The total heap consumption for tuples, constructors, exceptions and function closures, is this number of arguments added with its header size. In the case of an exception this is always 1. Note that the native code generator of the Hume compiler pre-allocates constants, and therefore **Mk...** operations are compiled to the new **GETCONST** instruction, which just loads a pointer. Naturally, the stack operations do not consume any heap space. Matching operations only compare values, and therefore consume neither heap nor stack space. An input operation allocates a new character in the heap and therefore consumes one character cell. Raising an exception involves first the construction of an exception closure. The 1 reserves space for the runtime value passed via the exception.

Special attention has to be paid for the function call instructions. A standard **Call** and a **TailCall** do not consume any heap. When calling a function variable via **CallVar** or **CallVarF** we have to distinguish between a saturated application, where enough arguments are supplied to execute the function, and an un-saturated application, where fewer arguments are supplied. In the former case,

$\mathcal{R}^{\texttt{MkBool b}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Bool}}$		$\mathcal{R}^{\texttt{StartMatches}}_{s,\mathcal{H}} = 0$	
$\mathcal{R}^{\texttt{MkChar x}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Char}}$		$\mathcal{R}^{\texttt{MatchRule}}_{s,\mathcal{H}} = 0$	
$\mathcal{R}^{\texttt{MkString x}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Str}} + \mid x \mid$		$\mathcal{R}^{\texttt{MatchedRule}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Reorder}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkInt i}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Int}}$	$\mathcal{R}^{\texttt{Copy i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchNone}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{CheckOutputs}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkFloat f}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Float}}$	$\mathcal{R}^{\texttt{CopyArg i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchAvailable}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{CopyInput i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkTuple n}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Tuple}} + n$	$\mathcal{R}^{\texttt{CreateFrame i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{AVAILSET}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{COPYALLINPUTS}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkCon c n}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Constr}} + n$	$\mathcal{R}^{\texttt{PushVar i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchAny}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Consume i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkVector n}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Vec}} + n$	$\mathcal{R}^{\texttt{PushVarF d i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchBool b}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MaybeConsume i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkFun f m p}}_{s,\mathcal{H}} = \mathcal{H}_{\mathrm{f}} + p$	$\mathcal{R}^{\texttt{MakeVar i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchChar x}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{CONSUMESET}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MkNone}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Goto lbl}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchString x}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Write i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{Push i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{If lbl}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchInt i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Input}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Char}}$
$\mathcal{R}^{\texttt{Pop i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Call f}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{MatchFloat i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Output}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{Slide i}}_{s,\mathcal{H}} = 0$		$\mathcal{R}^{\texttt{MatchTuple i}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Schedule}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{SlideVar i}}_{s,\mathcal{H}} = 0$		$\mathcal{R}^{\texttt{MatchCon i j}}_{s,\mathcal{H}} = 0$	$\mathcal{R}^{\texttt{Raise i}}_{s,\mathcal{H}} = \mathcal{H}_{\mathsf{Exn}} + 1$
$\mathcal{R}^{\texttt{SlideVarF i}}_{s,\mathcal{H}} = 0$		$\mathcal{R}^{\texttt{MatchExn x}}_{s,\mathcal{H}} = 0$	

$$\mathcal{R}^{\texttt{TailCall f n d sz}}_{s,\mathcal{H}} = 0$$

$$\mathcal{R}^{\texttt{CallVar f n}}_{s,\mathcal{H}} = \begin{cases} 0 & \text{if } \mathit{arity}\ s_f^{\mathit{locals}} \leq n + \mathit{providedArgs}\ s_f^{\mathit{locals}} \\ \mathcal{H}_{\mathsf{Chain}} + n & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{CallVarF d n}}_{s,\mathcal{H}} = \begin{cases} 0 & \text{if } \mathit{arity}\ (s - \square^i)_j^{\mathit{locals}} \leq n + \mathit{providedArgs}\ (s - \square^i)_j^{\mathit{locals}} \\ \mathcal{H}_{\mathsf{Chain}} + n & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{Unpack}}_{s,\mathcal{H}} = 0 \qquad \mathcal{R}^{\texttt{CallPrim1 f}}_{s,\mathcal{H}} = \mathcal{P}\mathit{rim1}_{s,\mathcal{H},f}\ s_0$$

$$\mathcal{R}^{\texttt{Return}}_{s,\mathcal{H}} = 0 \qquad \mathcal{R}^{\texttt{CallPrim2 f}}_{s,\mathcal{H}} = \mathcal{P}\mathit{rim2}_{s,\mathcal{H},f}\ s_0(s - \top_0)$$

Table 5.1: Heap consumption of HAM instructions

the function is executed without any additional heap consumption. In the latter case, a new so-called "chain" closure is allocated (representing a partial application) and the n arguments supplied to this call are stored in this closure. The function $\mathit{providedArgs}\ s_f^{\mathit{locals}}$ represents the number of all arguments to the partially applied function f in such chain closures in the heap. The notation for manipulating stacks is explained in Figure 6.1.

For primitive operations we use a table mapping the operation, and its runtime arguments, to the corresponding heap consumption: $\mathcal{P}\mathit{rim1}_{s,\mathcal{H},f}, \mathcal{P}\mathit{rim2}_{s,\mathcal{H},f}$. Usually, this is the same as the size of the return type, as indicated in the corresponding **Mk...** operation.

5.2.2 A Resource Algebra for Stack Space

For modelling stack space we again use $(\mathbb{N}, +, \leq)$ as resource algebra. In this case the constants are defined in Table 5.2.

For all heap operations the stack size increases by 1, since a pointer to the newly allocated heap object is left on the stack. Pushing adds and popping subtracts 1 from the stack size. The slide operations reduce the stack size. In the case of **Slide** the amount is part of the instruction, in the case of **SlideVar** and **SlideVarF** the amount is the value of a local or non-local variable, respectively. For a non-local variable this means that i stack frames are popped before a variable lookup is made to the j-th variable in the *locals* component of the frame. The notation $\mid s^{\mathit{vals}} \mid$ is used to denote the length of

$\mathcal{R}^{\texttt{MkBool b}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{MkChar x}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{MkString x}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{MkInt i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{MkFloat f}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{GETCONST}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{MkTuple n}}_{s,\mathcal{H}} = -n + 1$
$\mathcal{R}^{\texttt{MkCon c n}}_{s,\mathcal{H}} = -n + 1$
$\mathcal{R}^{\texttt{MkVector n}}_{s,\mathcal{H}} = -n + 1$
$\mathcal{R}^{\texttt{MkFun f m p}}_{s,\mathcal{H}} = -p + 1$
$\mathcal{R}^{\texttt{MkNone}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{Push i}}_{s,\mathcal{H}} = i$
$\mathcal{R}^{\texttt{PushVar i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{Pop i}}_{s,\mathcal{H}} = -i$
$\mathcal{R}^{\texttt{Slide i}}_{s,\mathcal{H}} = -i$
$\mathcal{R}^{\texttt{SlideVar i}}_{s,\mathcal{H}} = -(\mathcal{H}\ s_i^{locals})$
$\mathcal{R}^{\texttt{SlideVarF i j}}_{s,\mathcal{H}} = -((s - \square^i)_j^{locals})$

$\mathcal{R}^{\texttt{Copy i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{CopyArg i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{CreateFrame i}}_{s,\mathcal{H}} = i + 1$
$\mathcal{R}^{\texttt{PushVar i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{PushVarF d i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{MakeVar i}}_{s,\mathcal{H}} = -1$
$\mathcal{R}^{\texttt{Goto lbl}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{If lbl}}_{s,\mathcal{H}} = -1$
$\mathcal{R}^{\texttt{Call f}}_{s,\mathcal{H}} = 3$

$\mathcal{R}^{\texttt{StartMatches}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchRule}}_{s,\mathcal{H}} = - \mid s^{vals} \mid$
$\mathcal{R}^{\texttt{MatchedRule}}_{s,\mathcal{H}} = - \mid s^{vals} \mid$
$\mathcal{R}^{\texttt{MatchNone}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchAvailable}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{AVAILSET}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchNone b}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchBool b}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchChar x}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchString x}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchInt i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchFloat i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchTuple i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchCon i j}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchVector i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MatchExn x}}_{s,\mathcal{H}} = 0$

$\mathcal{R}^{\texttt{Reorder}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{CheckOutputs}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{CopyInput i}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{COPYALLINPUTS i}}_{s,\mathcal{H}} = i$
$\mathcal{R}^{\texttt{Consume i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{MaybeConsume i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{CONSUMESET i}}_{s,\mathcal{H}} = 0$
$\mathcal{R}^{\texttt{Write i}}_{s,\mathcal{H}} = -1$
$\mathcal{R}^{\texttt{Input}}_{s,\mathcal{H}} = 1$
$\mathcal{R}^{\texttt{Output}}_{s,\mathcal{H}} = -1$
$\mathcal{R}^{\texttt{Schedule}}_{s,\mathcal{H}} = 0$

$$\mathcal{R}^{\texttt{TailCall f n d sz}}_{s,\mathcal{H}} = \mathcal{S}_{saved} + sz - \Sigma_{0 \le i \le d}(\mathcal{S}_{saved} + \mid (s - \square^i)^{locals} \mid + \mid (s - \square^i)^{vals} \mid)$$

$$\mathcal{R}^{\texttt{CallVar f n}}_{s,\mathcal{H}} = \begin{cases} providedArgs\ s_f^{locals} + 3 & \text{if } arity\ s_f^{locals} \le n\ + providedArgs\ s_f^{locals} \\ -n + 1 & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{CallVarF f d n}}_{s,\mathcal{H}} = \begin{cases} providedArgs\ s_f^{locals} + 3 & \text{if } arity(s - \square^i)_j^{locals} \le n + providedArgs\ (s - \square^i)_j^{locals} \\ -n + 1 & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{Unpack}}_{s,\mathcal{H}} = \begin{cases} n - 1 & \text{if } \mathcal{H}\ s_0 = \mathsf{Tuple}\ n\ xs \\ n - 1 & \text{if } \mathcal{H}\ s_0 = \mathsf{Constr}\ c\ n\ xs \\ -1 & \text{otherwise} \end{cases}$$

$\mathcal{R}^{\texttt{CallPrim1 f}}_{s,\mathcal{H}} = 0$

$\mathcal{R}^{\texttt{Return}}_{s,\mathcal{H}} = -(\mathcal{S}_{saved} + \mid s^{locals} \mid + \mid s^{vals} \mid) + 1$

$\mathcal{R}^{\texttt{CallPrim2 f}}_{s,\mathcal{H}} = -1$

$\mathcal{R}^{\texttt{Raise x}}_{s,\mathcal{H}} = \mathcal{S}_{saved} + 1 - \Sigma_{0 \le i < \infty}(\mathcal{S}_{saved} + \mid (s - \square^i)^{locals} \mid + \mid (s - \square^i)^{vals} \mid)$

Table 5.2: Stack consumption of HAM instructions

the *vals* component in the topmost stack frame. In **MatchRule** and **MatchedRule** this component can be discarded, hence the negative value.

Copy operations add a pointer onto the top of the stack. A **CreateFrame** instruction pushes as many (dummy) elements onto the stack as prescribed by its argument i. Matching operations only compare values, and therefore consume neither heap nor stack space. A **Write** operation writes the heap elemented, pointed to by the top-of-stack element, to an output stream and then pops the pointer from the stack.

A **TailCall** operation removes all frames up to the d-th frame from the stack and then allocates a new frame of size sz. In the cases of applying a function variable we again have to distinguish between a saturated and an un-saturated application. In the former case, all arguments found in a chain of partial application closures are added onto the stack, plus 3 cells for the function frame itself. In the latter case, n cells are removed from the stack, a new "chain" closure is constructed, and a pointer to this closure is left on the stack.

An **Unpack** operation simply consumes the top-of-stack pointer unless it is a tuple or constructor. In the latter cases, it thereafter adds all the elements of the tuple or constructor onto the stack. In a **Raise** instruction the entire stack is discarded, a new frame is built on the empty stack and a pointer to an exception closure is pushed onto the stack. Therefore, we have to subtract the entire stack size, computed as a sum over the sizes of all stack frames, and then add the frame header size plus 1. A **Return** operation removes the topmost frame from the stack. The size of the stack frame is the frame header size plus the number of local variables plus the size of the expression stack. The stack consumption of a unary primitive operation is 0, because it takes its argument from the stack and leaves a pointer to the result on the stack. For a binary primitive operation the stack consumption is -1 because two arguments are taken from the stack and the result value is left on the stack.

5.2.3 A Resource Algebra for Time

$\mathcal{R}^{\texttt{MkBool b}}_{s,\mathcal{H}} = 85$ $\mathcal{R}^{\texttt{MkChar x}}_{s,\mathcal{H}} = 84$ $\mathcal{R}^{\texttt{MkInt i}}_{s,\mathcal{H}} = 83$ $\mathcal{R}^{\texttt{MkFloat f}}_{s,\mathcal{H}} = 91$ $\mathcal{R}^{\texttt{GETCONST i}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{MkTuple n}}_{s,\mathcal{H}} = 52n + 78$ $\mathcal{R}^{\texttt{MkCon c n}}_{s,\mathcal{H}} = 54n + 107$ $\mathcal{R}^{\texttt{MkFun f m p}}_{s,\mathcal{H}} = 60p + 166$ $\mathcal{R}^{\texttt{MkVector n}}_{s,\mathcal{H}} = 52n + 76$ $\mathcal{R}^{\texttt{MkNone}}_{s,\mathcal{H}} = 25$ $\mathcal{R}^{\texttt{Push i}}_{s,\mathcal{H}} = 9$ $\mathcal{R}^{\texttt{Pop i}}_{s,\mathcal{H}} = 9$ $\mathcal{R}^{\texttt{Slide i}}_{s,\mathcal{H}} = 53$ $\mathcal{R}^{\texttt{SlideVar i}}_{s,\mathcal{H}} = 76$ $\mathcal{R}^{\texttt{SlideVarF i d}}_{s,\mathcal{H}} = 11d + 79$	$\mathcal{R}^{\texttt{Copy i}}_{s,\mathcal{H}} = 31$ $\mathcal{R}^{\texttt{CopyArg i}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{CreateFrame i}}_{s,\mathcal{H}} = 72$ $\mathcal{R}^{\texttt{PushVar i}}_{s,\mathcal{H}} = 39$ $\mathcal{R}^{\texttt{PushVarF d i}}_{s,\mathcal{H}} = 11d + 35$ $\mathcal{R}^{\texttt{MakeVar i}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{Goto lbl}}_{s,\mathcal{H}} = 3$ $\mathcal{R}^{\texttt{Call f}}_{s,\mathcal{H}} = 70$ $\mathcal{R}^{\texttt{Return}}_{s,\mathcal{H}} = 51 + 15f$	$\mathcal{R}^{\texttt{StartMatches}}_{s,\mathcal{H}} = 111$ $\mathcal{R}^{\texttt{MatchRule}}_{s,\mathcal{H}} = 20$ $\mathcal{R}^{\texttt{MatchedRule}}_{s,\mathcal{H}} = 10$ $\mathcal{R}^{\texttt{MatchNone}}_{s,\mathcal{H}} = 11$ $\mathcal{R}^{\texttt{MatchAny}}_{s,\mathcal{H}} = 11$ $\mathcal{R}^{\texttt{MatchAvailable}}_{s,\mathcal{H}} = 13$ $\mathcal{R}^{\texttt{AVAILSET n}}_{s,\mathcal{H}} = 6 + 7n$ $\mathcal{R}^{\texttt{MatchBool b}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{MatchChar x}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{MatchString x}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{MatchInt i}}_{s,\mathcal{H}} = 32$ $\mathcal{R}^{\texttt{MatchFloat i}}_{s,\mathcal{H}} = 35$ $\mathcal{R}^{\texttt{MatchTuple i}}_{s,\mathcal{H}} = 11$ $\mathcal{R}^{\texttt{MatchCon i j}}_{s,\mathcal{H}} = 30$ $\mathcal{R}^{\texttt{MatchExn x}}_{s,\mathcal{H}} = 30$ $\mathcal{R}^{\texttt{MatchVector i}}_{s,\mathcal{H}} = 11$	$\mathcal{R}^{\texttt{Reorder}}_{s,\mathcal{H}} =$ $\mathcal{R}^{\texttt{CheckOutputs}}_{s,\mathcal{H}} = 602$ $\mathcal{R}^{\texttt{CopyInput i}}_{s,\mathcal{H}} = 78$ $\mathcal{R}^{\texttt{COPYALLINPUTS n}}_{s,\mathcal{H}} = 4 + 74n$ $\mathcal{R}^{\texttt{Consume i}}_{s,\mathcal{H}} = 31$ $\mathcal{R}^{\texttt{MaybeConsume i}}_{s,\mathcal{H}} = 28$ $\mathcal{R}^{\texttt{CONSUMESET n}}_{s,\mathcal{H}} = 32n$ $\mathcal{R}^{\texttt{Write i}}_{s,\mathcal{H}} =$ $\mathcal{R}^{\texttt{Input}}_{s,\mathcal{H}} =$ $\mathcal{R}^{\texttt{Output}}_{s,\mathcal{H}} =$ $\mathcal{R}^{\texttt{Schedule}}_{s,\mathcal{H}} = 602$ $\mathcal{R}^{\texttt{Raise i}}_{s,\mathcal{H}} = 377$
$\mathcal{R}^{\texttt{TailCall f n d sz}}_{s,\mathcal{H}} = 22 + 36d + 27n$ $\mathcal{R}^{\texttt{CallVar n}}_{s,\mathcal{H}} = 13$ $\mathcal{R}^{\texttt{CallVarF d n}}_{s,\mathcal{H}} = 13d$ $\mathcal{R}^{\texttt{MkString x}}_{s,\mathcal{H}} = 13 \mid x \mid +140$ $\mathcal{R}^{\texttt{Unpack x}}_{s,\mathcal{H}} = 44 \mid x \mid +51$ $\mathcal{R}^{\texttt{CopyInput i}}_{s,\mathcal{H}} = \begin{cases} 78 \text{ if } \mathsf{Int} \\ 74 \mid s \mid +4 \text{ if } \mathsf{Str} \end{cases}$		$\mathcal{R}^{\texttt{If lbl}}_{s,\mathcal{H}} = 30$ $\mathcal{R}^{\texttt{CallPrim1 f}}_{s,\mathcal{H}} = \mathcal{P}rim1_{s,\mathcal{H},f}\ s_0$ $\mathcal{R}^{\texttt{CallPrim2 f}}_{s,\mathcal{H}} = \mathcal{P}rim2_{s,\mathcal{H},f}\ s_0(s - \top_0)$	

Table 5.3: WCET of HAM instructions on a M32C (in cycles)

For modelling time consumption we use rational numbers as the base domain with the usual addition and less-or-equal relation $(\mathbb{R}, +, \leq)$ as resource algebra.

In Table 5.3 we summarise upper bounds for execution time on our target processor, a Renesas M32C. We have derived these bounds by using AbsInt's aiT tool [45], which performs machine-code-level analysis of worst case execution time. Details on these measurements are reported in Chapter 8.

This tool accounts for low-level architecture issues such as the states of the pipeline and, if present, of the cache. However, comparisons of the sum of HAM instruction costs with analysing entire basic blocks have shown that these low-level architecture issues have little influence on the overall costs on the M32C processor. We therefore don't model these low-level issues in this resource algebra for time and compute the costs of a basic block as the sum of the costs of its HAM instructions. In the future, we might consider a system that uses the aiT tool on basic blocks of the program under consideration.

The current version of the Hume native code-generator pre-allocates constants in the text area of the compiled code. Therefore, all **Mk...** instructions in the HAM code are replaced by the new **GETCONST** HAM instruction, which only fetches a pointer to the pre-allocated constant. The costs for the **Mk...** instructions are reported only for completeness. As expected, the costs of instructions involving a variable number of arguments are parametric in this number (usually n). All d arguments represent frame-depth, p represents the number of provided arguments in a **MkFun**. In the costs for the **Return** instruction, f indicates that the current function, from which to return, is the f-th function in the program. This is due to the fact that the **Return** on the M32C uses a switch statement over all possible return points. The HAM instructions **AVAILSET**, **COPYALLINPUTS**, and **CONSUMESET n** are new HAM instructions for checking the availability of wire values, for copying wire values and for consuming values (after a successful match), respectively. Each instruction is parameterised by the number of arguments.

==.c = 125					
==.n = 125	>.c = 124		/.n = 206		@ = 89
==.w = 125	>.n = 124		/.w = 206		vecdef =
==.i = 125	>.w = 124		/.i = 206		vecmap =
==.f = 125	>.i = 124	+.n = 116	/.f = 959	toInt.i = 2973	update =
==.s = 125	>.f = 209	+.w = 116	/ = 959	toInt.n = 2973	vecmake =
==.e = 125	>.s = 209	+.i = 116	mod = 1294	toInt.w = 2973	sqrt =
== = 125	> = 209	+.f = 1106	mod.n = 1294	toInt.f = 2973	ln =
!=.c = 243	<=.c = 122	+ = 1106	mod.w = 1294	toInt = 2973	log10 =
!=.n = 243	<=.n = 122	-.n = 116	mod.i = 1294	toFloat.n = 699	sin =
!=.w = 243	<=.w = 122	-.w = 116	% = 1294	toFloat.w = 699	cos =
!=.i = 243	<=.i = 122	-.i = 116	div.n = 206	toFloat.i = 699	tan =
!=.f = 243	<=.f = 211	-.f = 1112	div.w = 206	toFloat.f = 699	asin =
!=.s = 243	<=.s = 211	- = 1112	div.i = 206	toFloat = 699	acos =
!=.e = 243	<= = 211	*.n = 124	div = 206	toChar =	atan =
!= = 243	<.c = 124	*.w = 124	&& = 153	show =	sinh =
>=.c = 122	<.n = 124	*.i = 124	|| = 156	length =	cosh =
>=.n = 122	<.w = 124	*.f = 356	^& = 151	++ =	tanh =
>=.w = 122	<.i = 124	* = 356	^| = 151	++.s =	atan2 = 14305
>=.i = 122	<.f = 209		^ = 151		** =
>=.f = 211	<.s = 209		~ = 151		exp =
>=.s = 211	< = 209		not = 118		
>= = 211					

Table 5.4: WCET of primitive operations on a M32C (in cycles)

Table 5.4 summarises the bounds on execution time for primitive functions used in the Hume compiler. The type is indicated by the suffix in the function name with `i` for integer, `n` for natural, `w` for word, `f` for float, and `c` for character. The polymorphic versions should never be used by the Hume native code generator and are listed only for completeness. The time bounds in Table 5.4 have been

derived by using the aiT tool. Work on analysing such primitive operations is ongoing.

Chapter 6

Resource-Aware Operational Semantics of the Hume Abstract Machine

Kevin Hammond and Hans-Wolfgang Loidl

Abstract

This chapter describes as a 2-level, small step operational semantics for the Hume Abstract Machine. We formally specify the components of the machine and its behaviour in the form of a 2-level, small-step, operational semantics and give a reference implementation of the instructions of the HAM. The operational semantics uses the approach of resource algebras, which we have developed in a previous project [4], to collect information on the resource consumption during execution. The resource algebras are designed in a modular way and can be instantiated without modifying the rules of the operational semantics.

6.1 Formalisation of the HAM

The goal of this section is to give a formal specification of the HAM semantics, corresponding to the behaviour of the references implementation in the previous section. This formalisation is the basis of an HAM operational semantics that is encoded in Isabelle. This encoding is developed alongside this document and will be the basis for automated certification of Hume code as needed in WP4.

6.1.1 Basic Definitions

As basic types we use *Locn* as locations into the heap, *Ref* as references (either a proper location or a null-pointer *Nullref*). Heap values (*HVal*) are a tagged union of basic types, compound types of tuples or constructors, exceptions or function closures. We use disjoint name spaces for *Label*, *Function* and box names (*BName*).

$$
\begin{array}{lcl}
\mathit{Locn} & \equiv & \mathbb{N} \\
\mathit{Ref} & \equiv & \mathit{Nullref} \mid \mathit{Ref}\ \mathit{Locn} \\
\mathit{HVal} & \equiv & \mathsf{Int}\ \mathbb{I} \mid \mathsf{Tuple}\ \mathbb{N}\ (\mathit{Ref}\ \mathit{list}) \\
 & & \mid \mathsf{Bool}\ \mathbb{B} \mid \mid \mathsf{Constr}\ \mathbb{N}\ \mathbb{N}\ (\mathit{Ref}\ \mathit{list}) \\
 & & \mid \mathsf{Char}\ \mathbb{R} \mid \mathsf{f}\ \mathit{Function}\ \mathbb{N}\ \mathbb{N}\ (\mathit{Ref}\ \mathit{list}) \\
 & & \mid \mathsf{Str}\ \mathbb{R} \mid \mathsf{Exn}\ \mathbb{N}\ \mathit{Ref} \\
 & & \mid R\ \mathit{Ref} \mid \mathsf{None}
\end{array}
$$

Stack values are references into the heap. A stack is a list of frame records. A frame record (*Frame*) contains the return address (*ret*), the arguments (*args*), the local variabls (*locals*) and the expression stack (*vals*). By default, stack operations such as push and pop work on the value stack. See Figure 6.1 for the notation used to access and modify stack entries.

The heap is a finite map of locations to heap values, i.e. a function from locations to either the value *None* or *Some* x, where x is an *HVal*. An IO record collects all state info, not recorded in stack or heap. These are mainly related to wire input/output and rule matching. In particular, name of the box (b), a flag whether the box is blocked (*blocked*), a pointer to the next rule in the rule set of the box (*rp*), and pointers to the next wire (*inp*) and the next stack value (*mp*) to be checked in a rule match.

$$
\begin{array}{lcl}
\mathit{SVal} & \equiv & \mathit{Ref} \\
\mathit{Frame} & \equiv & (\!|\ \mathit{ret} :: \mathit{Label},\ \mathit{args} :: \mathit{SVal}\ \mathit{list},\ \mathit{locals} :: \mathit{SVal}\ \mathit{list},\ \mathit{vals} :: \mathit{SVal}\ \mathit{list}\ |\!) \\
\mathit{Stack} & \equiv & \mathit{Frame}\ \mathit{list} \\
\mathit{Heap} & \equiv & \mathit{Locn} \rightsquigarrow_f \mathit{HVal} \\
\mathit{Wire} & \equiv & \mathbb{N} \\
\mathit{IO} & \equiv & (\!|\ b :: \mathit{BName},\ \mathit{blocked} :: \mathit{bool},\ \mathit{crp} :: \mathbb{N},\ \mathit{rp} :: \mathbb{N},\ \mathit{inp} :: \mathit{Wire}, \mathit{mp} :: \mathbb{N}\ |\!)
\end{array}
$$

6.1.2 A Roadmap through the Operational Semantics

The initial set of rules deals with the construction of heap values. The next set encodes basic operations on the stack components. Note that we abstract over several pointers present in the reference implementation, by using a structured stack representation as a list of frames.

The main control-flow operations are (conditional) jumps and calls covered in the rules stepGoto, stepIfTrue, stepIfFalse, stepTailCall, stepCall, stepReturn, stepCallPrim1, stepCallPrim2 at the end of the list of rules. To determine the target of a (conditional) jump the table Σ^{Lab}, mapping labels to instructions sequences, is used. For function calls a similar table Σ^{Fun} and for exceptions Σ^{Exn} is used.

The rules relevant for higher-order functions are stepCallVarSaturated, stepCallVarUnsaturated, stepCallVarFSaturated, stepCallVarFUnsaturated, stepApSaturated, stepApUnsaturated, stepSlideVar, stepSlideVarF. A higher-order function can be called from a local variable `CallVar` a non-local variable `CallVarF` or from the top of the stack `Ap`. In each case we distinguish between the case where enough arguments are provided to perform the call, i.e. the call is saturated, or whether another function closure needs to be allocated. The only rule needed for exceptions is stepRaise.

The interface between expression level and system level is defined by the rule stepCheckOutputsFalse. Note that no rules for a `Schedule` as first instruction or an empty instruction list exists. Thus, in this case, no reduction can be done on expression level, which operationally amounts to yielding control to the scheduler and performing a rule on the system level.

6.1.3 Rules of the Operational Semantics

The following judgement of the small-step semantics on Hume expression level

$$s, \mathcal{H}, io, \theta \vdash \texttt{cs} \Downarrow_n^m (s', \eta', io', \texttt{cs}', p)\ \theta'$$

is read as: with an initial stack s, initial heap $\mathcal{H}$, IO record io and with the wire environment θ, the HAM instruction sequence (a list) cs evaluates in $n - m$ steps to a final stack s', final heap η', a final IO record io', using p resources and leaving the code sequence $\texttt{cs}'$ as continuation. The new wire environment is θ'. The semantics of resources is intentionally left open in this semantics. It can be instantiated to every structure corresponding to a resource algebra as discussed in Section 5.2. Being a small-step semantics, the result of one step will be a state with a continuation code sequence.

While the usage of a stack and heap are standard, the IO record collects additional information on the state of a box execution, that is primarily needed during box input/output and matching operations. The pointers mp and inp are indices into the arguments component of the top stack frame (s^{args}) and into the set of in-wires for the current box. The `MatchBool` etc. instructions check whether the current argument is a boolean value, and if not continue with the next rule. The `CopyInput` instruction is the only one that uses the inp pointer in order to copy the value from the current wire into the argument portion of the stack frame.

The notation used in the rules of the operational semantics are summarised in Figure 6.1. We use # for list cons, @ for list append, _++ for incrementing an integer variable. The construct $(\!| \cdot |\!)$ constructs a record, $r(\!| \ x := \cdot \ |\!)$ modifies the record r by updating the field x. We use $f\ x$ for applying a function f to argument x, and $f(x \mapsto \cdot)$ for updating the function f with a mapping of x to $\cdot$.

Heap operations

The heap operations are fairly standard. They allocate a fresh location, by calling *fresh* $(dom\ \mathcal{H})$. It is guaranteed by definition that *fresh* $S \notin S$. A pointer to the newly allocated heap object is left on the stack. In the case of tuples and constructors, the top n elements are taken from the stack and put into the heap closure.

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkBool x})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + l, \mathcal{H}(l \mapsto (\mathsf{Bool}\ x)), io, \texttt{cs}, \mathcal{R}_{s,\mathcal{H}}^{\texttt{MkBool x}})\ \theta}\ (\textsc{stepMkBool})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkChar x})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + l, \mathcal{H}(l \mapsto (\mathsf{Char}\ x)), io, \texttt{cs}, \mathcal{R}_{s,\mathcal{H}}^{\texttt{MkChar x}})\ \theta}\ (\textsc{stepMkChar})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkString x})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + l, \mathcal{H}(l \mapsto (\mathsf{Str}\ x)), io, \texttt{cs}, \mathcal{R}_{s,\mathcal{H}}^{\texttt{MkString x}})\ \theta}\ (\textsc{stepMkString})$$

l	a reference to location l in the heap
$fresh\ S$	returns an element x, s.t. $x \notin S$
$dom\ f$	the domain of a partial function f, i.e. the values x for which $f\ x$ is defined
s_0	get the top-of-stack element from the stack s
$s + v$	push value v on top of the stack s
$s + xs$	push all elements in the list xs on top of the stack s
$s - \top$	pop the top element from the stack s
$s - \top^n$	pop the n topmost elements from the stack s
$s - \square^i$	pop the top i frames from the stack s
$s + \bot^n$	push n dummy values onto the stack s
$s - \top^n + s_0$	pop elements 1 to n from the stack but keep the top-of-stack
$s_{0..n-1}$	get the elements 0 to n-1 from the stack s
$s_n^{locals} := s_0$	assign top-of-stack to n-th local variable
$s + \boxed{\text{c,f}}$	allocate a new frame (for function f) on the stack, with return address c
$s - \square$	remove the topmost frame from the stack s
$\boxed{\text{f=l}} = s_{\boxed{\text{i}}}$	binds l to the value of field f in the i-th frame on stack s
io_f	access the value of field f in record io
$io(\!\vert\ f := x\ \vert\!)$	set the value of field f in record io to x
$f(x \mapsto y)$	update the function f to map x to y
$wire^{b,m}$	the name of the m-th in-wire of box b
$wire_{b,m}$	the name of the m-th out-wire of box b

Figure 6.1: Notation used in the operational semantics

$$\frac{l = fresh\ (dom\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkInt x})\#\texttt{cs} \Downarrow^n_{n+1} ((s + l, \mathcal{H}(l \mapsto (\mathsf{Int}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkInt x}}_{s,\mathcal{H}})\ \theta} \text{ (stepMkInt)}$$

$$\frac{l = fresh\ (dom\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkFloat x})\#\texttt{cs} \Downarrow^n_{n+1} ((s + l, \mathcal{H}(l \mapsto (\mathsf{Float}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkFloat x}}_{s,\mathcal{H}})\ \theta} \text{ (stepMkFloat)}$$

$$\frac{l = fresh\ (dom\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkTuple j})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^j + l, \mathcal{H}(l \mapsto (\mathsf{Tuple}\ x\ s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{MkTuple j}}_{s,\mathcal{H}})\ \theta} \text{ (stepMkTuple)}$$

$$\frac{l = fresh\ (dom\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkCon i j})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^j + l, \mathcal{H}(l \mapsto (\mathsf{Constr}_i\ j\ s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{MkCon i j}}_{s,\mathcal{H}})\ \theta} \text{ (stepMkCon)}$$

$$\frac{l = fresh\ (dom\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkFun f m p})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^p + l, \mathcal{H}(l \mapsto (\text{f}\ f\ m\ p\ (s_{0..p-1}))), io, \texttt{cs}, \mathcal{R}^{\texttt{MkFun f m p}}_{s,\mathcal{H}})\ \theta} \text{ (stepMkFun)}$$

$$\frac{l = fresh\ (dom\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{MkNone})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \mathcal{H}(l \mapsto \mathsf{None}), io, \texttt{cs}, \mathcal{R}^{\texttt{MkNone}}_{s,\mathcal{H}})\ \theta} \text{ (stepMkNone)}$$

Stack operations

The stack operations usually modify the expression stack, i.e. the *vals* component of the top stack frame. Superscripting accesses a particular component of the top stack frame; subscripting such a component selects an element; e.g. s_x^{args} accesses the argument x of the top stack frame. The notation $s -_{\square}^{i}$ is used for popping the top i frames from stack s. For more details on notation see Figure 6.1.

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Push i})\#\texttt{cs} \Downarrow^n_{n+1} (s + \bot^i, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Push i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepPush})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Pop i})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^i, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Pop i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepPop})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Slide i})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top^{i+1} + s_0, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Slide i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepSlide})$$

$$\frac{\mathsf{Int}\ i = \mathcal{H}\ s_x^{locals}}{s, \mathcal{H}, io, \theta \vdash (\texttt{SlideVar x})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top^{i+1} + s_0, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{SlideVar x}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepSlideVar})$$

$$\frac{\mathsf{Int}\ i = \mathcal{H}\ (s -_{\square}^{x})_y^{locals}}{s, \mathcal{H}, io, \theta \vdash (\texttt{SlideVarF x y})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top^{i+1} + s_0, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{SlideVarF x y}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepSlideVarF})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Copy i})\#\texttt{cs} \Downarrow^n_{n+1} (s + s_i, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Copy i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCopy})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{CopyArg i})\#\texttt{cs} \Downarrow^n_{n+1} ((s + s_i^{args}, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{CopyArg i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCopyArg})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{CreateFrame i})\#\texttt{cs} \Downarrow^n_{n+1} (s + \boxed{\text{c,f}} + \bot^i, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{CreateFrame i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCreateFrame})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{PushVar i})\#\texttt{cs} \Downarrow^n_{n+1} (s + s_i^{locals}, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{PushVar i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepPushVar})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{PushVarF i j})\#\texttt{cs} \Downarrow^n_{n+1} (s + (s -_{\square}^{i})_j^{locals}, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{PushVarF i j}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepPushVarF})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{MakeVar i})\#\texttt{cs} \Downarrow^n_{n+1} ((s_i^{locals} := s_0) - \top, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{MakeVar i}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepMakeVar})$$

Control-flow operations

The following set of rules describes the control flow on the expression level of Hume. Here we assume that the compiler has generated for each box several tables, mapping labels, including function names, and exceptions to their corresponding instruction sequences: $\Sigma^{Lab}, \Sigma^{Exn}$. For convenience we drop the box name from the table lookup. Note, that the box name is always available as io_b.

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Goto l})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io, \Sigma^{Lab}_{\texttt{l}}, \mathcal{R}^{\texttt{Goto l}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepGoto})$$

$$\frac{l = s_0 \quad \mathcal{H}\ l = \mathsf{Bool}\ \mathit{true}}{s, \mathcal{H}, io, \theta \vdash (\texttt{If lbl})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \mathcal{H}, io, \Sigma^{Lab}_{\texttt{lbl}}, \mathcal{R}^{\texttt{If lbl}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepIfTrue})$$

$$\frac{l = s_0 \quad \mathcal{H}\ l = \mathsf{Bool}\ \mathit{false}}{s, \mathcal{H}, io, \theta \vdash (\texttt{If lbl})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{If lbl}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepIfFalse})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Call f})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} (s + \boxed{\text{c,f}}, \mathcal{H}, io, \Sigma^{\mathit{Fun}}_{\texttt{f}}, \mathcal{R}^{\texttt{Call f}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCall})$$

$$\frac{s' \quad = \quad ((s - \square^d) + \boxed{\text{c,f}})^{\mathit{args}} := s_{0\ldots j-1} + \top^z}{s, \mathcal{H}, io, \theta \vdash (\texttt{TailCall f j d z})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} (s', \mathcal{H}, io, \Sigma^{\mathit{Fun}}_{\texttt{f}}, \mathcal{R}^{\texttt{TailCall f j d z}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepTailCall})$$

$$\frac{\boxed{\text{ret=l}} = s_{\boxed{0}}}{s, \mathcal{H}, io, \theta \vdash (\texttt{Return})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \square + s_0, \mathcal{H}, io, \Sigma^{\mathit{Lab}}_{l}, \mathcal{R}^{\texttt{Return}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepReturn})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{CallPrim1 f})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^1 + (f\ \ s_0), \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{CallPrim1 f}}_{s,\mathcal{H}})\ \theta} \ (\textsc{stepCallPrim1})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{CallPrim2 f})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^2 + (f\ \ s_0\ \ s_1), \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{CallPrim2 f}}_{s,\mathcal{H}})\ \theta} \ (\textsc{stepCallPrim2})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{Raise x})\#\texttt{cs} \Downarrow^n_{n+1} (\emptyset + l, \mathcal{H}(l \mapsto (\mathsf{Exn}\ x\ s_0)), io, \Sigma^{\mathit{Exn}}_{x}, \mathcal{R}^{\texttt{Raise x}}_{s,\mathcal{H}})\ \theta} \ (\textsc{stepRaise})$$

Three HAM instructions can be used for applying a higher-order function to some arguments: `CallVar` for a local variable, `CallVarF` for a non-local variable, and `Ap` for a function closure pointed to by the top-of-stack element. Each of these 3 come in a version for unsaturated application, i.e. the number of arguments provided is smaller than the arity of the function, and in a saturated version. In the former case a new function closure is built in the heap. In the latter case the function is applied to all its arguments.

$$\frac{\text{f}\ f\ m\ p\ xs = \mathcal{H}\ s^{\mathit{locals}}_i \quad p + j \geq m}{s, \mathcal{H}, io, \theta \vdash (\texttt{CallVar i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} ((s - s_{0..j-1} + \boxed{\text{c,f}})^{\mathit{args}} := xs@s_{0..j-1}, \mathcal{H}, io, \Sigma^{\mathit{Fun}}_{\texttt{f}}, \mathcal{R}^{\texttt{CallVar i j}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCallVarSaturated})$$

$$\frac{\text{f}\ f\ m\ p\ xs = \mathcal{H}\ s^{\mathit{locals}}_i \quad p + j < m \quad l = \mathit{fresh}\ \mathcal{H}}{s, \mathcal{H}, io, \theta \vdash (\texttt{CallVar i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} (s - s_{0..j-1}, \mathcal{H}(l \mapsto \text{f}\ f\ m\ (p + j)\ (xs@s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{CallVar i j}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCallVarUnsaturated})$$

$$\frac{\text{f}\ f\ m\ p\ xs = \mathcal{H}\ (s - \square^d)^{\mathit{locals}}_i \quad p + j \geq m}{s, \mathcal{H}, io, \theta \vdash (\texttt{CallVarF d i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} ((s - s_{0..j-1} + \boxed{\text{c,f}})^{\mathit{args}} := xs@s_{0..j-1}, \mathcal{H}, io, \Sigma^{\mathit{Fun}}_{\texttt{f}}, \mathcal{R}^{\texttt{CallVarF d i j}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepCallVarFSaturated})$$

$$\frac{\text{f}\ f\ m\ p\ xs = \mathcal{H}\ (s - \square^d)^{\mathit{locals}}_i \quad p + j < m \quad l = \mathit{fresh}\ \mathcal{H}}{s, \mathcal{H}, io, \theta \vdash (\texttt{CallVarF d i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} (s - s_{0..j-1}, \mathcal{H}(l \mapsto \text{f}\ f\ m\ (p + j)\ (xs@s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{CallVarF d}}_{s,\mathcal{H}}} \quad (\textsc{stepCallVarFUnsaturated})$$

$$\frac{\text{f}\ f\ m\ p\ xs = \mathcal{H}\ s_0 \quad p + j \geq m}{s, \mathcal{H}, io, \theta \vdash (\texttt{Ap j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} ((s - s_{0..j-1} + \boxed{\text{c,f}})^{\mathit{args}} := xs@s_{0..j-1}, \mathcal{H}, io, \Sigma^{\mathit{Fun}}_{\texttt{f}}, \mathcal{R}^{\texttt{Ap j}}_{s,\mathcal{H}})\ \theta} \quad (\textsc{stepApSaturated})$$

$$\frac{\mathrm{f}\ f\ m\ p\ xs = \mathcal{H}\ s_0 \quad p + j < m \quad l = \mathit{fresh}\ \mathcal{H}}{s, \mathcal{H}, io, \theta \vdash (\texttt{Ap j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} (s - s_{0..j-1}, \mathcal{H}(l \mapsto \mathrm{f}\ f\ m\ (p+j)\ (xs@s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{Ap j}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepApUnsaturated})$$

Matching, scheduling and I/O operations

The following functions are used to retrieve wire names: $wire^{b,i}$ is the name of the i-th input wire of box b; $wire_{b,i}$ is the name of the i-th output wire of box b. These can be calculated from the static information in *Box*. Note that box name and wire are usually retrieved from the io record. The function *outputable* checks whether all needed output wires are free, corresponding to the innermost loop in Figure 3.4.

The rules for matching always compare the heap cell, pointed to by the top-of-stack element, with a value encoded in the operation itself. If the expected kind of heap cell is found, evaluation continues with the next instruction, otherwise the next rule is tried.

In these rules several fields of the IO record are used to perform matching. The field io_{rp} points to the next rule. If a match fails with the current rule, execution will continue at $\Sigma^{RuleSet}_{io_{rp}}$, where $\Sigma^{RuleSet}$ is the rule-set of the current box. The field io_{inp} points to the wire that is examined for input. Before performing matching the HAM code must check availability of a data item on this wire. The field io_{mp} points to a position on the stack representing the current value. It is copied from the wire to the stack before a match can be performed.

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{StartMatches})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{rp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{StartMatches}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepStartMatches})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchRule})\#\texttt{cs} \Downarrow^n_{n+1} (s^{vals} := [], \mathcal{H}, io(\!|\ io_{inp} := 0, io_{mp} := 0, io_{rp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchRule}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchRule})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchedRule})\#\texttt{cs} \Downarrow^n_{n+1} (s^{vals} := [], \mathcal{H}, io, \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchedRule}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchedRule})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchNone})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{inp} +\!+, io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchNone}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchNone})$$

$$\frac{\exists\ v.\ \theta\ wire^{io_b, io_{inp}} = \mathit{Some}\ v}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchAvailable})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{inp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchAvailable}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchAvailableTrue})$$

$$\frac{\neg\exists\ v.\ \theta\ wire^{io_b, io_{inp}} = \mathit{Some}\ v}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchAvailable})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{inp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchAvailable}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchAvailableFalse})$$

$$\frac{\mathcal{H}\ s^{args}_{io_{mp}} = \mathit{Some}\ (\mathsf{Bool}\ b)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchBool b})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchBool b}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchBoolTrue})$$

$$\frac{\neg(\mathcal{H}\ s^{args}_{io_{mp}} = \mathit{Some}\ (\mathsf{Bool}\ b))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchBool b})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchBool b}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMatchBoolFalse})$$

$$\frac{\mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Char}\ c)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchChar c})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchChar c}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchCharTrue)}$$

$$\frac{\neg(\mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Char}\ c))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchChar c})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchChar c}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchCharFalse)}$$

$$\frac{\mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Str}\ x)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchString x})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchString x}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchStringTrue)}$$

$$\frac{\neg(\mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Str}\ x))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchString x})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchString x}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchStringFalse)}$$

$$\frac{\mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Int}\ i)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchInt i})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchInt i}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchIntTrue)}$$

$$\frac{\neg(\mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Int}\ i))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchInt i})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchInt i}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchIntFalse)}$$

$$\frac{\exists\ xs.\ \mathcal{H}\ s^{args}_{io_{mp}} = Some(\mathsf{Tuple}\ m\ xs)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchTuple m})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchTuple m}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchTupleTrue)}$$

$$\frac{\neg(\exists\ xs.\ \mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Tuple}\ m\ xs))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchTuple m})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchTuple m}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchTupleFalse)}$$

$$\frac{\exists\ xs.\ \mathcal{H}\ s^{args}_{io_{mp}} = Some(\mathsf{Constr}\ i\ j\ xs)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchCon i j})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchCon i j}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchConTrue)}$$

$$\frac{\neg(\exists\ xs.\ \mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Constr}\ i\ j\ xs))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchCon i j})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchCon i j}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchConFalse)}$$

$$\frac{\exists\ l.\ \mathcal{H}\ s^{args}_{io_{mp}} = Some(\mathsf{Exn}\ x\ l)}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchExn x})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchExn x}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchExnTrue)}$$

$$\frac{\neg(\exists\ l.\ \mathcal{H}\ s^{args}_{io_{mp}} = Some\ (\mathsf{Exn}\ x\ l))}{s, \mathcal{H}, io, \theta \vdash (\texttt{MatchExn x})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchExn x}}_{s,\mathcal{H}})\ \theta} \text{(stepMatchExnFalse)}$$

The following rules unpack structured data, i.e. they bring the contents of a tuple, constructor or function closure onto the stack.

$$\frac{\exists\ l\ m\ xs.s_0 = l\ \wedge\ \mathcal{H}\ l = \mathit{Some}\ (\mathsf{Tuple}\ m\ xs)\ \wedge\ s' = (s - \top) + xs}{s, \mathcal{H}, io, \theta \vdash (\texttt{Unpack})\#\texttt{cs} \Downarrow^n_{n+1} (s', \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Unpack}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepUnpackTuple})$$

$$\frac{\exists\ l\ t\ m\ xs.s_0 = l\ \wedge\ \mathcal{H}\ l = \mathit{Some}\ (\mathsf{Constr}_t\ m\ xs)\ \wedge\ s' = (s - \top) + xs}{s, \mathcal{H}, io, \theta \vdash (\texttt{Unpack})\#\texttt{cs} \Downarrow^n_{n+1} (s', \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Unpack}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepUnpackCon})$$

$$\frac{\exists\ l\ f\ m\ p\ xs.s_0 = l\ \wedge\ \mathcal{H}\ l = \mathit{Some}\ (\mathrm{f}\ f\ m\ p\ xs)\ \wedge\ s' = (s - \top) + xs}{s, \mathcal{H}, io, \theta \vdash (\texttt{Unpack})\#\texttt{cs} \Downarrow^n_{n+1} (s', \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Unpack}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepUnpackFun})$$

The following operations copy box input into the heap, check whether a box can write to its output wires, and perform the actual write operation. Note that the compiler must ensure that all `Write` operations are guarded by `CheckOutput` operations.

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{CopyInput m})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \mathcal{H}(l \mapsto (\theta\ \mathit{wire}^{io_b,m})), io, \texttt{cs}, (\mathcal{R}^{\texttt{CopyInput m}}_{s,\mathcal{H}}))\ \theta}\ (\textsc{stepCopyInput})$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Consume m})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Consume m}}_{s,\mathcal{H}})\ \theta(\mathit{wire}^{io_b,m} \mapsto \mathit{None})}\ (\textsc{stepConsume})$$

$$\frac{\exists\ v.\ \theta\ \mathit{wire}^{io_b,m} = \mathit{Some}\ v}{s, \mathcal{H}, io, \theta \vdash (\texttt{MaybeConsume m})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{MaybeConsume m}}_{s,\mathcal{H}})\ \theta(\mathit{wire}^{io_b,m} \mapsto \mathit{None})}\ (\textsc{stepMaybeConsumeTrue})$$

$$\frac{\neg\ \exists\ v.\ \theta\ \mathit{wire}^{io_b,m} = \mathit{Some}\ v}{s, \mathcal{H}, io, \theta \vdash (\texttt{MaybeConsume m})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{MaybeConsume m}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepMaybeConsumeFalse})$$

$$\frac{\mathit{outputable}\ io\ \theta}{s, \mathcal{H}, io, \theta \vdash (\texttt{CheckOutputs})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{CheckOutputs}}_{s,\mathcal{H}})\ \theta}\ (\textsc{stepCheckOutputsTrue})$$

$$\frac{\neg\ (\mathit{outputable}\ io\ \theta)}{s, \mathcal{H}, io, \theta \vdash (\texttt{CheckOutputs})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathcal{H}, io(\!|\ \mathit{blocked} := \mathit{true}\ |\!), (\texttt{Schedule})\#(\texttt{CheckOutputs})\#\texttt{cs}, \mathcal{R}^{\texttt{CheckOutputs}}_{s,\mathcal{H}})}\ (\textsc{stepCheckOutputsFalse})$$

$$\frac{l = s_0 \quad \mathcal{H}\ l = \mathit{Some}\ x}{s, \mathcal{H}, io, \theta \vdash (\texttt{Write m})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top), \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Write m}}_{s,\mathcal{H}})\ \theta(\mathit{wire}_{io_b,m} \mapsto \mathit{Some}\ x)}\ (\textsc{stepWrite})$$

The operations `Input` and `Output` interact with the outside world, enabling character input and output. Since we are mainly interested in the internal behaviour of the system, in particular its resource consumption, we do not model the exact values and use a dummy character value $\perp$ instead. Costs are still accurately modelled, provided they are constant for all possible character values.

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{Input})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \mathcal{H}(l \mapsto (\mathsf{Char}\ \perp)), io, \texttt{cs}, \mathcal{R}^{\texttt{Input}}_{s,\mathcal{H}})\ \theta}\quad (\textsc{stepInput})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \mathcal{H})}{s, \mathcal{H}, io, \theta \vdash (\texttt{Output})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \mathcal{H}, io, \texttt{cs}, \mathcal{R}^{\texttt{Output}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepOutput)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{WithinStack h p})\#\texttt{cs} \Downarrow^n_{n+1} (s + io_{\mathit{splim}}, \mathcal{H}, io(\!|\ \mathit{splim} := |\ s\ | + p\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{WithinStack h p}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepWithinStackSpace)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{WithinHeap h p})\#\texttt{cs} \Downarrow^n_{n+1} (s + io_{\mathit{hplim}}, \mathcal{H}, io(\!|\ \mathit{hplim} := |\ \mathit{dom}\ \mathcal{H}\ | + p\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{WithinHeap h p}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepWithinHeapSpace)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{DoneWithinStack h})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \mathcal{H}, io(\!|\ \mathit{splim} := s_0\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{DoneWithinStack h}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepDoneWithinStack)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{DoneWithinHeap h})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \mathcal{H}, io(\!|\ \mathit{hplim} := s_0\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{DoneWithinHeap h}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepDoneWithinHeap)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{Within h t})\#\texttt{cs} \Downarrow^n_{n+1} (s + t, io, \texttt{cs}, \mathcal{R}^{\texttt{Within h t}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepWithin)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{DoneWithin})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathit{unsetTimer}\ io, \texttt{cs}, \mathcal{R}^{\texttt{DoneWithin}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepDoneWithin)}$$

$$\frac{}{s, \mathcal{H}, io, \theta \vdash (\texttt{RaiseWithin h t})\#\texttt{cs} \Downarrow^n_{n+1} (s - s_{0..2-1}, \mathit{setTimer}\ io\ s_0\ s_1, \texttt{cs}, \mathcal{R}^{\texttt{RaiseWithin h t}}_{s,\mathcal{H}})\ \theta} \quad \text{(stepRaiseWithin)}$$

Expression-level semantics

As a small-step semantics the above rules specify one step in the evaluation, depending on the next element in the instruction list `cs`. The semantics on expression level is the transitive closure over this relation in the following sense.

$$\frac{c \neq \texttt{Schedule} \quad s, \mathcal{H}, io, \theta \vdash (\texttt{c}\#\texttt{cs}) \Downarrow^m_{m+1} (s'', \eta'', io'', \texttt{cs}'', p'')\ \theta'' \quad s'', \eta'', io'', \theta'' \vdash \texttt{cs}'' \rightsquigarrow_n (s', \eta', io', \texttt{cs}', p')\ \theta'}{s, \mathcal{H}, io, \theta \vdash (\texttt{c}\#\texttt{cs}) \rightsquigarrow_{m+n+1} (s', \eta', io', \texttt{cs}', p'' + p')\ \theta'} \quad \text{(ssem)}$$

The continuation cs' is part of the result, because the computation may block in the CheckOutputs rule. The costs of the entire computation is the sum of the costs of the components, using the $+$ operation as defined by the relevant resource algebra. The step counters m, n are only needed for technical reasons (to enable induction over this step counter) and do not represent resource consumption in any way. Note that the rule is restricted to the case where the first instruction is different from `Schedule`. This assures that in the case of a `Schedule` control is yielded to the scheduler, which then picks the next runnable box, removes the initial `Schedule` and continues with evaluating the box.

Exceptions

The above rule for the expression-level semantics is slightly simplified in the sense that it doesn't check for (synchronous) exceptions. Such checks can be added to the rule, provided that the resource algebra used in the semantics models the resource that might become empty. For example, in the case of heap space a pre-condition of the form

$$\mid hplim - hp \mid < \mathcal{R}^{\mathsf{c}}_{s,\mathcal{H}}$$

can be added, if $\mathcal{R}$ models heap consumption. If this condition is false, another rule can be added for raising a heap-overflow exception.

6.1.4 System level

As static information we use for each box a mapping *Wiring* that maps a wire index to the corresponding box name and wire index it is connected to. The tables Σ^{Fun}, Σ^{Lab} and Σ^{Exn} are used to map function names, labels and exceptions to their corresponding instruction sequences. The $\Sigma^{RuleSet}$ is a partial mapping from natural numbers to instruction sequences, containing the code for the rules in the box. Here the domain needs to be a total order to allow reordering. A *Box* is then represented as a rule-set (a mapping from natural numbers to instruction sequences) and such a wiring.

$$\begin{array}{lcl}
\Sigma^{RuleSet} & \equiv & \mathbb{N} \rightsquigarrow_f \mathit{Instr\ list} \\
\Sigma^{Fun} & \equiv & \mathit{Function} \Rightarrow \mathit{Instr\ list} \\
\Sigma^{Lab} & \equiv & \mathit{Label} \Rightarrow \mathit{Instr\ list} \\
\Sigma^{Exn} & \equiv & \mathit{Exception} \Rightarrow \mathit{Instr\ list} \\
\mathit{Wiring} & \equiv & \mathit{Wire} \Rightarrow (\mathit{BName} \times \mathit{Wire}) \\
\mathit{Box} & \equiv & (\Sigma^{RuleSet} \times \Sigma^{Lab} \times \Sigma^{Exn} \times \mathit{Wiring})
\end{array}$$

We model the dynamic state of the entire system as follows. The entire system (*System*) is a triple (bs, β, θ), where bs is a collection of box names still to be processed in the current iteration, β is a (total) mapping of box-names to box states, and θ is a (total) mapping of wire-names to wire states. A *box state* (*BState*) is a quadrupel $(s, \mathcal{H}, io, cs)$, where s is the stack , $\mathcal{H}$ is the heap, io is the IO record, and cs is the continuation, i.e. the code still to be processed by the box. The *wire state* (*WState*) is simply an optional heap value, i.e. either a proper heap value (*Some* x) or an empty flag (*None*).

$$\begin{array}{lcl}
\mathit{WState} & \equiv & \mathit{HVal\ option} \\
\mathit{BState} & \equiv & \mathit{Stack} \times \mathit{Heap} \times \mathit{IO} \times \mathit{Instr\ list} \\
\mathit{System} & \equiv & \mathit{BName\ set} \times (\mathit{BName} \Rightarrow \mathit{BState}) \times (\mathit{WName} \Rightarrow \mathit{WState})
\end{array}$$

NeededIn maps for each box, each rule in the ruleset to the set of needed inputs, i.e. a set of wires on which input must be available for the rule to fire. *NeededOut* maps for each box, each rule in the ruleset to the set of needed inputs, i.e. a set of wires to which this rule will write. These tables are filled with the compiler directives shown in Figure 3.12.

$$\begin{array}{lcl}
\mathit{NeededIn} & :: & \mathit{Box} \Rightarrow \mathbb{N} \Rightarrow \mathit{Wire\ set} \\
\mathit{NeededOut} & :: & \mathit{Box} \Rightarrow \mathbb{N} \Rightarrow \mathit{Wire\ set}
\end{array}$$

The global table *RuleTab* maps a box name to a *Box* structure, containing its static information. Here we are only interested in the rule-set component of the box structure and we use the shorthand $\mathit{RuleTab}_1$ for the first projection on the corresponding rule tab entry.

$$\begin{array}{lcl}
\mathit{RuleTab} & :: & \mathit{BName} \Rightarrow \mathit{Box} \\
\mathit{RuleTab}_1 & :: & \mathit{BName} \Rightarrow \mathit{RuleSet}
\end{array}$$

We need the following additional definitions to specify the semantics on system level:

$$
\begin{array}{lcl}
\mathit{outputable} & :: & \mathit{IO} \Rightarrow (\mathit{WName} \Rightarrow \mathit{WState}) \Rightarrow \mathit{bool} \\
\mathit{outputable\ io\ \theta} & \equiv & (\forall w.\ w \in (\mathit{NeededOut}\ (\mathit{RuleTab}\ io_b)\ io_{rp}) \longrightarrow \\
 & & \exists\ v.\theta\ \mathit{wire}_{io_b,m} = \mathit{Some}\ v) \\
 & & \\
\mathit{runnable} & :: & (\mathit{WName} \Rightarrow \mathit{WState}) \Rightarrow \mathit{BState} \Rightarrow \mathit{bool} \\
\mathit{runnable}\ \theta\ (s, \mathcal{H}, io, cs) & \equiv & \neg(io_{blocked}) \wedge \\
 & & (\exists r.\ r \in \mathit{dom}\ (\mathit{RuleTab}_1\ io_b) \wedge \\
 & & \quad (\forall w.\ w \in (\mathit{NeededIn}\ (\mathit{RuleTab}\ io_b)\ r) \longrightarrow \exists\ v.\theta\ \mathit{wire}^{io_b,w} = \mathit{Some}\ v)) \\
 & & \\
\mathit{restartable} & :: & (\mathit{WName} \Rightarrow \mathit{WState}) \Rightarrow \mathit{BState} \Rightarrow \mathit{bool} \\
\mathit{restartable}\ \theta\ (s, \mathcal{H}, io, cs) & \equiv & io_{blocked} \wedge \\
 & & (\forall w.\ w \in (\mathit{NeededOut}\ (\mathit{RuleTab}\ io_b)\ io_{rp}) \longrightarrow \theta\ \mathit{wire}^{io_b,w} = \mathit{None})
\end{array}
$$

The transition relation on system level has the form $s \rightharpoonup s'$ meaning that in one step, the system state s proceeds to state s'. The transitive closure over the small-step semantics on expression level, written as $\rightsquigarrow_m$, is defined in the previous section. We use $\emptyset$ to denote the empty heap, mapping all locations to *None*, and the empty stack, an empty list of frames.

The rules defining the HAM semantics on system level are as follows. We use S as a short-hand for bs, β, θ, p. We distinguish between three cases: if the box, selected from the scheduling queue, is a runnable box, then the continuation of the box must be empty, and execution continues with the next rule in the box; note that in this case the box starts with an empty heap and stack; if the box, selected from the scheduling queue, is a restartable box, then the continuation must start with a `Schedule`, which is removed from the code and the remaining code is executed on expression level; if the scheduling queue is empty, it is re-filled using the $\mathcal{U}_S$ operation. The costs recorded in the last field of the state, combine the costs of the individual operations on scheduling level referring to constants such as $\mathcal{R}_S^{\in}$ as explained in the following sub-section.

$$
\frac{\begin{array}{c} bn \in_{bs,\beta,\theta,p} bs \quad (s, \mathcal{H}, io, []) = \beta\ bn \quad \mathit{runnable}\ \theta\ (s, \mathcal{H}, io, []) \\ cs = \mathit{RuleTab}_1\ io_b\ io_{rp} \quad \emptyset, \emptyset, io, \theta \vdash cs \rightsquigarrow_m (s', \eta', io', cs', p')\ \theta' \end{array}}{(bs, \beta, \theta, p) \rightharpoonup ((bs \ominus_{bs,\beta,\theta,p} bn) \oplus_{bs,\beta,\theta,p} bn,\ \beta(bn \mapsto (s', \eta', io', cs')),\ \theta', p + \mathcal{R}_S^{\in} + \mathcal{R}_S^{\ominus} + p' + \mathcal{R}_S^{\oplus})} \quad (\textsc{scheduleRunnable})
$$

$$
\frac{\begin{array}{c} bn \in_{bs,\beta,\theta,p} bs \quad (s, \mathcal{H}, io, cs'') = \beta\ bn \quad \mathit{restartable}\ \theta\ (s, \mathcal{H}, io, cs'') \\ (\texttt{Schedule}\#cs) = cs'' \quad s, \mathcal{H}, io, \theta \vdash cs \rightsquigarrow_m (s', \eta', io', cs', p')\ \theta' \end{array}}{(bs, \beta, \theta, p) \rightharpoonup ((bs \ominus_{bs,\beta,\theta,p} bn) \oplus_{bs,\beta,\theta,p} bn, \beta(bn \mapsto (s', \eta', io', cs')),\ \theta', p + \mathcal{R}_S^{\in} + \mathcal{R}_S^{\ominus} + p' + \mathcal{R}_S^{\oplus})} \quad (\textsc{scheduleRestartable})
$$

$$
\frac{\neg\ \exists\ bn \in_{bs,\beta,\theta,p} bs}{(bs, \beta, \theta, p) \rightharpoonup (\mathcal{U}_{bs,\beta,\theta,p}, \mathcal{C}_B\beta\ \theta, \mathcal{C}_W\beta\ \theta, p + \mathcal{R}_S^{\mathcal{U}} + \mathcal{R}_S^{\mathcal{C}_B} + \mathcal{R}_S^{\mathcal{C}_W})} \quad (\textsc{scheduleNIL})
$$

Note that this specification of the system level is parameterised by the concrete membership relation $\in_S$ and by the operations for picking the next box to be executed $\ominus_S$ and putting it back into the system $\oplus_S$. In the case where there is no next box available for scheduling, we assume an operation for generating a new collection of schedulable boxes, $\mathcal{U}_S$, an operation for modifying the box mapping, $\mathcal{C}_B$, and an operation for modifying the wire mapping, $\mathcal{C}_W$).

We expect implementations of Hume to use well-known scheduling policies such as round-robin scheduling based on some total ordering of the boxes derived from the source code (this is what's used in the reference implementation of the HAM). However, we leave the concrete realisation of the

scheduling operations ($\in_S, \ominus_S, \oplus_S$) and of the synchronisation operations ($\mathcal{U}_S, \mathcal{C}_B, \mathcal{C}_W$) to the implementor. Naturally, the overall costs of the system will be depend on the scheduling policy. Thus, all these parametric operations have to come with corresponding cost functions, describing how the costs coming out of the expression level shall be combined.

The following operations implement round-robin scheduling. We initially sort the boxes by a global ordering derived from the source code. The membership relation only examines the head of the list. Picking the next element means taking the head of the list. Putting a box back means adding it to the end of the list. Since the length of the scheduling queue never decreases no global synchronisation is needed, and the definition of $\mathcal{U}_S, \mathcal{C}_B, \mathcal{C}_W$ is arbitrary.

$$
\begin{array}{lcl}
\mathcal{D} & \equiv & \mathit{BName\ list} \\
b \in_S bs & \equiv & b = hd\ bs \\
\ominus_S & \equiv & \lambda bs\ b.tl\ bs \\
\oplus_S & \equiv & \lambda bs\ b.bs@[b] \\
\mathcal{U}_S & \equiv & [] \\
\mathcal{C}_B\ \beta\ \theta & \equiv & \beta \\
\mathcal{C}_W\ \beta\ \theta & \equiv & \theta
\end{array}
$$

Cost modelling on system level

The cost model on the system level is realised similarly to the expression level. We use a resource algebra $(R, +, \leq)$, where R must be the same data structure as on the expression level, but $+$ and $\leq$ may be different operations. For example, in a parallel implementation of the HAM the overall time consumption is not necessarily the sum over all the time consumptions of all boxes plus system time. The constants in the system-level resource algebra must correspond to the basic scheduling operations: $\mathcal{R}_S^{\ominus}$ for getting a box name from the scheduling queue, $\mathcal{R}_S^{\oplus}$ for putting a box back into the scheduling queue, $\mathcal{R}_S^{\in}$ for testing whether a box name is in the scheduling queue; $\mathcal{R}_S^{\mathcal{U}}$ for re-filling the scheduling queue; $\mathcal{R}_S^{\mathcal{C}_B}$ for synchronising all boxes, and $\mathcal{R}_S^{\mathcal{C}_W}$ for synchronising all wires. Note, that in general all these costs depend on the entire system state. In the case of modelling time consumption in a round-robin scheduler, the last three constants would be 0; $\mathcal{R}_S^{\in}$, $\mathcal{R}_S^{\ominus}$, $\mathcal{R}_S^{\ominus}$ would be the costs for testing for list membership, extracting an element from the list and adding an element to the end of the list.

6.2 Summary

As specification of the HAM behaviour we presented a 2 level, small-step operational semantics as well as a reference implementation in pseudo C. The former will act as the basis for reasoning about resource consumption. The latter is more concrete and thus more clearly exhibits the costs involved in executing HAM instructions. These costs are captured in the operational semantics via operations of a resource algebra applied to counters that are part of the system state. Since the rules of the semantics only refer to these abstract operations of the resource algebra, they can be chosen independently from the rules of the semantics, and we have shown instances for heap space, stack space and time consumption, thus giving cost models of the HAM for these resources. On top of the small-step semantics we have defined a system-level semantics, that deals with the scheduling of multiple boxes in Hume. Here we parameterise the semantics with concrete scheduling operations, accompanied again by a system-level resource algebra for composing costs on system level. Thus, we arrive at a simple, yet flexible, formalisation of HAM behaviour and resource consumption on system level.

Chapter 7

A Generic Formal Foundation for the Schopenhauer Resource Analyses

Steffen Jost and Kevin Hammond

Abstract

In this chapter, we describe a generic foundation for constructing static analyses for determining time and space costs for Schopenhauer programs.

7.1 Introduction

We describe a generic foundation for constructing static analyses for determining time and space costs for Schopenhauer programs, as formally defined in Chapter 6 Although we had originally constructed an analysis that was specific to one of our space metrics, which we then intended to adapt to the different problems arising in the other situations, we subsequently determined that all three metrics could be incorporated into the same generic formal rules, with significant advantages for consistency and future reuse. It follows that the observations made in this section of the report are completely generic.

We have chosen to use a type-based approach in order to produce a compile-time static analysis. We improve upon earlier published work (e.g. [69, 70]) by allowing higher-order functions, arbitrary recursive datatypes and resource polymorphism, as well as expanding from a fairly limited toy language to the application-oriented Schopenhauer language. Our approach is as follows:

a) We first refine the Schopenhauer language to a theoretically more manageable core version (Section 7.1.1). These changes are all non-essential, in the sense of merely removing syntactic sugar. For example, we replace the special syntax for Tuples and Vectors with that for general datatypes. In all cases, we take care to ensure that the correct cost for the specialised syntactic forms is still preserved. For example, in the case of Tuples and Vectors, we introduce general cost parameters for datatypes that can correctly capture the costs for Tuples, Vectors and user-defined datatypes.

b) We then define a standard type system for this core syntax (Section 7.1.2). This forms a general basis for our analyses, which we will then augment by the proper Heap-space analysis, allowing a clear separation between the underlying generic basis and the specifics that are required for the Heap-space analysis.

c) We then define an augmented type system (Chapter 8), which includes an *amortised* WCET analysis. The basic principle of an amortised analysis is described in more detail in Section 8.1.1. The augmented typing introduces a number of *linear* numeric constraints. It follows that the process of deriving such an annotated typing can be simplified by abstracting over all these numeric values, and then using a standard linear program solver to instantiate these values. This allows the WCET analysis to be automated. Since the linear programs that we generate do not present a challenge for current state of the art solvers for linear programs, it follows that they can be solved very quickly and efficiently, allowing our WCET analysis to be sensibly automated. We have previously shown [70] that the generated linear programs have good formal properties, such as being solvable by integers if they are solvable at all (this is known to be an NP-hard problem in general).

d) Finally, the augmented typings allow us to compute strict upper bounds on the WCET consumption of the typed terms, so completing our WCET analysis.

7.1.1 Core Schopenhauer Syntax

Our core syntax for Schopenhauer is shown in Figure 7.1. We assume four disjoint sets of identifiers: Var for names of functions, boxes and wires (which we refer to as *identifiers*), usually ranged over by *id*; Var for *variables*, usually ranged over by the letter x,y and z; Constrs for *constructors*, usually ranged over by the letter c and Exn for *exceptions*, usually ranged over by $_{\text{exn}}$. For simplicity we sometimes conveniently assume a further subdivision of Var into three distinct subsets for functions, boxes and wires, omitting preconditions such as "*id is a wire identifier*", since this is usually clear from the context anyway.

program ::=	$decl_1$ `;` ... `;` $decl_n$	$n \geq 1$
decl ::=	*box* | id `=` *expr* | id ⟨ $match_1$ `|` ⋯ `|` $match_n$ ⟩	$n \geq 1$
box ::=	`box` id *ins* *outs* `fair/unfair` *bmatches* [`handle` *cmatches*]	
ins/outs ::=	⟨ $\mathrm{id}_1, \ldots, \mathrm{id}_n$ ⟩	$n \geq 0$
bmatches ::=	*expr* | ⟨ $bmatch_1$ `|` ⋯ `|` $bmatch_n$ ⟩	$n \geq 1$
cmatches ::=	*exnexpr* | ⟨ $cmatch_1$ `|` ⋯ `|` $cmatch_n$ ⟩	$n \geq 1$
bmatch ::=	⟨ $bpat_1$ `,` ... `,` $bpat_n$ ⟩ `->` *expr*	$n \geq 1$
cmatch ::=	*cpat* `->` *exnexpr*	
match ::=	⟨ pat_1 `,` ... `,` pat_n ⟩ `->` *expr*	$n \geq 1$
exnmatch ::=	⟨ pat_1 `,` ... `,` pat_n ⟩ `->` *exnexpr*	$n \geq 1$
expr ::=	int | float | char | bool | string | `*`	
	| var $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| id $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| con $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| `(` $expr_1$ `,` ... `,` $expr_n$ `)`	$n \geq 2$
	| `if` $expr_1$ `then` $expr_2$ `else` $expr_3$	
	| `case` *expr* `of` ⟨ $match_1$ `|` ⋯ `|` $match_n$ ⟩	$n \geq 1$
	| `let` ⟨ $vdecl_1$ `,` ... `,` $vdecl_n$ ⟩ `in` *expr*	$n \geq 1$
	| *expr* `within` int `time raise` exn`()`	
	| *expr* `within` int `stack raise` exn`()`	
	| *expr* `within` int `heap raise` exn`()`	
	| *raiseexpr*	
raiseexpr ::=	`raise` exn`(`*exnexpr*`)`	
exnexpr ::=	int | float | char | bool | string | `*` | var	
	| con $exnexpr_1$ ⋯ $exnexpr_n$	$n \geq 0$
	| `(` $exnexpr_1$ `,` ... `,` $exnexpr_n$ `)`	$n \geq 2$
	| `if` $exnexpr_1$ `then` $exnexpr_2$ `else` $exnexpr_3$	
	| `case` *exnexpr* `of` ⟨ $exnmatch_1$ `|` ⋯ `|` $exnmatch_n$ ⟩	$n \geq 1$
	| `let` ⟨ $exnvdecl_1$ `,` ... `,` $exnvdecl_n$ ⟩ `in` *exnexpr*	$n \geq 1$
vdecl ::=	var `=` *expr*	
exnvdecl ::=	var `=` *exnexpr*	
bpat ::=	*pat* | `*` | `_*`	
cpat ::=	exn *pat*	
pat ::=	int | float | char | bool | string | `_` | var	
	| con pat_1 ⋯ pat_n	$n \geq 0$
	| `(` pat_1 `,` ... `,` pat_n `)`	$n \geq 2$

Figure 7.1: Schopenhauer Abstract Syntax

There are two main differences from standard Schopenhauer syntax: i) elimination of syntactic sugar for specific datatypes; and ii) conversion to let-normal form. These are described in detail in the following subsections. Both changes are largely superficial, but remove a number of redundancies, which would otherwise obfuscate the underlying mechanics of the analysis. The Prototype Schopenhauer Abstract Machine Compiler (phamc) used in the EmBounded project has already been augmented to automatically transform normal Schopenhauer code into the desired format. Work is also underway to allow automatic tracing of error messages in the phase to the original Schopenhauer source code, so that the translation to the core syntax becomes transparent to the Schopenhauer programmer. More information about the implementation can be found in EmBounded Deliverable D13 [79].

It is important to note that the translation to the core syntax is designed to be completely *cost neutral*, so that it never changes either the actual execution costs of Schopenhauer program or those reported by the analysis. The compilation process to abstract machine code (or concrete machine code) is still based on the original Schopenhauer source code, since the original Schopenhauer syntax is much more convenient to a programmer, and the compiler can take advantage of information about source level constructs.

Arbitrary recursive datatypes

In principle, the treatment of tuples, vectors, lists, trees, etc. should all be identical from the viewpoint of the analysis. Hence we have decided to eliminate the syntactic sugar provided for certain datatypes such as vectors and tuples and the special "none" value (`*`). Instead, we use the normal mechanisms for arbitrary recursive datatypes, i.e. instead of simply writing `(42,true)` for a pair of an integer and a bool, it is necessary to define a specific datatype and associated constructors and to use these instead, e.g. T2intbool`(42,true)`. The special costs associated with vectors and tuples are preserved by parameterising the cost that is assigned to *all* user-defined, possibly recursive datatypes. This parameterisation also easily allows for future optimisations of some datatypes by simply adjusting the cost parameters for those types. As a side effect of this transformation, the syntax of top-level pattern matches become much more simple, since instead of a list of patterns, only a single pattern of tuple type needs to be matched. Hence all top-level expressions become a single case-instruction. Again, the differences in cost are handled by a simple parameter, called case-kind, which signals whether we deal with a box entry-level, function entry-level or an ordinary expression-level matching. It is interesting to note that this mirrors exactly the treatment within the prototype implementation of the space-analysis.

Let-normal form

Nested expressions quickly become unwieldy in theoretical proof-work. We avoid this problem by using a program transformation into a "let-normal form", i.e. an expression such as "$f(e_1)(e_2)$" is transformed into "`let` x_1 `=` e_1 `in let` x_2 `=` e_2 `in` $f(x_1, x_2)$". While this form has the advantage of conveying the order of evaluation more explicitly, its main advantage is that the `let` construct becomes the only command that models sequential evaluation. For example, if we want to prove something about the term "`if` e_1 `then` e_2 `else` e_3" then this proof will contain similar steps to "`let` x `=` e_1 `in if` x `then` e_2 `else` e_3", which are missing if we only need to deal with conditionals of the form "`if` x `then` e_2 `else` e_3", i.e. where the guard expression is always known to be a simple variable. This transformation therefore allows us to remove many uninteresting repetitions in our theoretical work, allowing us to expose the important parts in each case that we consider. A third benefit is that the transformation also heavily reduces the treatment of exceptions, since evaluating a variable cannot throw an exception, e.g. in the above example, the transformed conditional does not need a specific rule for the case that an exception was raised during the evaluation of the guarding expression. The further reduces redundancies in the rules. However, if performed naively, the transformation above would clearly alter the consumption of time, stack and perhaps other resources, something that would be unacceptable if we are to obtain correct

upper bounds on execution costs. We circumvent this problem by introducing a pseudo-expression that is not available to Schopenhauer programmers, the *ghost-let*, written "`LET` $\cdots$ `IN` ". The idea is that the execution of a ghost-let has no operational cost, but behaves otherwise similarly to a normal `let` construct. This means that we can achieve a pseudo-transformation into the let-normal form without altering the actual resource cost of an execution. The simple program transformation $\varrho\,(\cdot)$ into ghost-let-normal form then works as follows (any syntactic constructs not mentioned here are unaltered by the transformation):

Prim Op	$op\ e_1 \cdots e_k$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho\,(e_1), \ldots, x_k$ `=` $\varrho\,(e_k)$ `IN` $op\ x_1 \cdots x_k$
Call	$fid\ e_1 \cdots e_k$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho\,(e_1), \ldots, x_k$ `=` $\varrho\,(e_k)$ `IN` $fid\ x_1, \ldots, x_k$
Constructor	$c\ e_1 \cdots e_k$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho\,(e_1), \ldots, x_k$ `=` $\varrho\,(e_k)$ `IN` $c\ x_1 \cdots x_k$
Tuple	$(e_1, \ldots, e_k)$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho\,(e_1), \ldots, x_k$ `=` $\varrho\,(e_k)$ `IN` $(x_1, \ldots, x_k)$
Conditional	`if` e_1 `then` e_2 `else` e_3	$\rightsquigarrow$	`LET` x `=` $\varrho\,(e_1)$ `IN if` x `then` $\varrho\,(e_2)$ `else` $\varrho\,(e_3)$
Case	`case` e `of` pat_1 `->` e_1 `|` $\cdots$ `|` pat_k `->` e_k	$\rightsquigarrow$	`LET` x `=` $\varrho\,(e)$ `IN` `case` x `of` pat_1 `->` $\varrho\,(e_1)$ `|` $\cdots$ `|` pat_b `->` $\varrho\,(e_b)$

where the introduced variables $x, y, x_i, \ldots$ are always assumed to be fresh. One might wonder why we did not add the following transformation for the ordinary let as well:

$$\texttt{let}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{in}\ e \rightsquigarrow \begin{array}{l}\texttt{LET}\ x_1 = \varrho\,(e_1), \ldots, x_1 = \varrho\,(e_k)\ \texttt{IN} \\ \quad \texttt{let}\ x_1 = x_1, \ldots, x_k = x_k\ \texttt{in}\ \varrho\,(e)\end{array}$$

The reason is that the expression `let` x_1 `=` $e_1, \ldots, x_k$ `=` e_k `in` e is really just an abbreviation for `let` x_1 `=` e_1 `in let` x_2 `=` e_2 `in` $\cdots$ `let` x_k `=` e_k `in` e. Hence the subsequent expressions may depend on the previously-introduced variables. This leads to an unusual resource behaviour, especially since the Schopenhauer compiler optimises the evaluation of `let` constructs by preallocating a stack frame for the defined variables similar to a call frame, which in turn allows the result of a preceding evaluation of subexpression to be popped from the stack before computing the next. This peculiarity requires us to treat `let` constructs individually, thereby excluding the `let` construct from the transformation into let-normal form.

7.1.2 Basic HUME Type System

Schopenhauer *types* are defined by the following grammar:

$$\begin{array}{lll} A & ::= & B \mid C \mid X \mid \mu X.\{c_1{:}A_{(1,1)}, \ldots, A_{(1,j_1)} \mid \ldots \mid c_k{:}A_{(1,k)}, \ldots, A_{(k,j_k)}\} \\ B & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{char} \mid \mathsf{bool} \mid \mathsf{string} \\ C & ::= & A_1, \ldots, A_k \rightarrow A \end{array}$$

where $c_i \in \mathsf{Constrs}$ are constructor labels. We have the usual set of base types, higher-order types, type variables and type abstractions that can be used to build arbitrary recursive datatypes.

A recursive datatype consists of a type-variable abstraction and a partial-mapping from the set of constructor labels Constrs to an ordered, possibly empty list of type-terms. For example, standard binary trees with integer-labelled leaves and nodes would be written

$$\mu Y.\{\mathsf{Leaf{:}int} \mid \mathsf{Node{:}int}, Y, Y\}$$

Note that as for all partial mappings, the order in which the constructors are listed is irrelevant, hence the type $\mu Y.\{\mathsf{Node{:}int}, Y, Y \mid \mathsf{Leaf{:}int}\}$ is identical to that given above. Furthermore we treat all Schopenhauer types module α-equivalence on the bound type variables, e.g. $\mu Y.\{\mathtt{C}{:}Y\} = \mu X.\{\mathtt{C}{:}X\}$. We should point out that non-recursive datatypes are already subsumed, since the abstracted variable is

not required to occur within the range of the partial map. Although we may sometimes omit the leading μ-operator and the superfluous abstracted variable for a non-recursive types for the sake of brevity, we do not wish to treat non-recursive types any different from recursive ones, thereby avoiding unnecessary redundancies in our type rules. Note moreover, that when referring to types, we only refer to closed type-terms. There is no explicit folding and unfolding of recursive datatypes. Folding and unfolding is implicitly paired with the construction and the destruction of each datatype in the usual way. Finally, note that we allow higher-order types in an uncurried style. This is to allow for possibly different costs in the case of partial applications, so allowing us to explicitly distinguish between $A_1 \rightarrow A_2 \rightarrow C$ and $A_1, A_2 \rightarrow C$, and so achieving a more accurate costing for higher-order functions. In this document, we will use the term "partial application" to refer only to applications of functions of the latter type.

Type Rules for Expressions

We first show how to type a single Schopenhauer expression. Our type rules for Schopenhauer expressions have the form

$$\Sigma; \Gamma \vdash e : A$$

meaning that the Schopenhauer expression e is of type A within context Γ and signature Σ. A context is a partial map from the set of variables Var to types. A signature Σ maps function identifiers belonging to the set Var to a triple consisting of a term defining the function's body, an ordered list of argument variables and a type. Since the signature Σ remains fixed with the program to be analysed, we refrain from mentioning it within the premises of each expression-level rule in favour of a leaner notation. We will describe the requirements on the signature in the later section 7.1.2, which describes how to treat whole Schopenhauer programs.

$$\frac{}{\varnothing \vdash \texttt{()} : \mathsf{unit}} \quad \text{(Unit)}$$

$$\frac{b \in \mathbb{B}}{\varnothing \vdash b : \mathsf{bool}} \quad \text{(Bool)}$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash n : \mathsf{int}} \quad \text{(Int)}$$

$$\frac{r \in \mathbb{R}}{\varnothing \vdash r : \mathsf{float}} \quad \text{(Float)}$$

$$\frac{c \texttt{ is a character}}{\varnothing \vdash c : \mathsf{char}} \quad \text{(Char)}$$

$$\frac{s \texttt{ is a string}}{\varnothing \vdash s : \mathsf{string}} \quad \text{(String)}$$

$$\frac{}{x{:}A \vdash x : A} \quad \text{(Var)}$$

$$\frac{\mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash x \ \mathrm{op} \ y : \mathsf{int}} \quad \text{(PrimBOp Int)}$$

$$\frac{\text{op} \in \{\texttt{-}\}}{x{:}\mathsf{int} \vdash \ \text{op}\ y : \mathsf{int}} \qquad \text{(PrimUOp Int)}$$

$$\frac{\text{op} \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{x{:}\mathsf{float}, y{:}\mathsf{float} \vdash x\ \text{op}\ y : \mathsf{float}} \qquad \text{(PrimBOp Float)}$$

$$\frac{\text{op} \in \{\texttt{-.}\}}{x{:}\mathsf{float} \vdash \ \text{op}\ y : \mathsf{float}} \qquad \text{(PrimUOp Float)}$$

$$\frac{\text{op} \in \{\texttt{==}, \texttt{>=}, \texttt{<=}\}}{x{:}A, y{:}A \vdash x\ \text{op}\ y : \mathsf{bool}} \qquad \text{(PrimBOp Eq)}$$

$$\frac{\text{op} \in \{\texttt{and}, \texttt{or}\}}{x{:}\mathsf{bool}, y{:}\mathsf{bool} \vdash x\ \text{op}\ y : \mathsf{bool}} \qquad \text{(PrimBOp Bool)}$$

$$\frac{\text{op} \in \{\texttt{not}\}}{x{:}\mathsf{bool} \vdash \ \text{op}\ y : \mathsf{bool}} \qquad \text{(PrimUOp Bool)}$$

$$\frac{\Sigma(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \rightarrow C\right) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash \mathit{fid}\ y_1 \cdots y_k : C} \qquad \text{(App)}$$

$$\frac{\Sigma(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \rightarrow C\right) \qquad k \geq 1 \qquad k < a}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash \mathit{fid}\ y_1 \cdots y_k : A_{k+1}, \ldots, A_a \rightarrow C} \qquad \text{(Under App)}$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \rightarrow C\right) \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1, \ldots, y_a{:}A_a \vdash \mathit{fid}\ y_1 \cdots y_a : C \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1}, \ldots, y_k{:}A_k \vdash x\ y_{a+1} \cdots y_k : E\end{array}}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash \mathit{fid}\ y_1 \cdots y_k : E} \qquad \text{(Over App)}$$

$$\frac{D = A_1, \ldots, A_a \rightarrow C \qquad k \geq 1 \qquad k = a}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash z\ y_1 \cdots y_k : C} \qquad \text{(App Var)}$$

$$\frac{\begin{array}{c}D = A_1, \ldots, A_a \rightarrow C \qquad k \geq 1 \qquad k < a \\ B = A_{k+1}, \ldots, A_a \rightarrow C\end{array}}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash z\ y_1 \cdots y_k : B} \qquad \text{(Under App Var)}$$

$$\frac{\begin{array}{c}D = A_1, \ldots, A_a \rightarrow C \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1, \ldots, y_a{:}A_a \vdash z\ y_1 \cdots y_a : C \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1}, \ldots, y_k{:}A_k \vdash x\ y_{a+1} \cdots y_k : E\end{array}}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash z\ y_1 \cdots y_k : E} \qquad \text{(Over App Var)}$$

The rules for function application, closure-creation (or under application) and over-application are repeated twice: once for applications to normal functions, and once for applications to *closures* arising from higher-order calls through variables representing parameters to function calls. While we have tried to minimise the number of rules that are required, we must treat the application of literal functions and called variables separately, since the enriched type systems for the resource analyses require slightly different costs in each case.

$$\frac{c \in \mathsf{Constrs} \qquad C = \mu X.\{\cdots \mid c : B_1, \ldots, B_k \mid \cdots\} \quad A_i = B_i \vee (A_i = C \wedge B_i = X) \text{ (for } i = 1, \ldots, k)}{x_1{:}A_1, \ldots, x_k{:}A_k \vdash c\ x_1 \ldots x_k : C} \tag{Constr}$$

$$\frac{\Gamma \vdash e_t : A \qquad \Gamma \vdash e_f :}{\Gamma, x{:}\mathsf{bool} \vdash \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A} \tag{Conditional}$$

$$\frac{\forall i\,.\, \begin{cases} A \vdash \mathrm{pat}_i\ ?\triangleright\ \Delta_i \\ \Gamma, \Delta_i \vdash e_i : B \end{cases}}{\Gamma, x{:}A \vdash \texttt{case}\ \mathrm{ck}\ x\ \texttt{of}\ \mathrm{pat}_1\ \texttt{->}\ e_1 \mid \cdots \mid \mathrm{pat}_k\ \texttt{->}\ e_k : B} \tag{Case}$$

$$\frac{\forall i\,.\, \begin{cases} \Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \curlyvee(A_i \mid \mathcal{A}_i) \\ \Delta_i, \Gamma_i \vdash e_i : A_i \end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash \texttt{LET}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{IN}\ e : A} \tag{Ghost let}$$

$$\frac{\forall i\,.\, \begin{cases} \Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \curlyvee(A_i \mid \mathcal{A}_i) \\ \Delta_i, \Gamma_i \vdash e_i : A_i \end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash \texttt{let}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{in}\ e_{k+1} : A_{k+1}} \tag{Let}$$

The type rules for the let-expression seem somewhat complicated due to the fact that they allow later definitions to depend on earlier ones. This requires the use of the sharing relations, since in the enriched type systems for the resource analyses need to be aware of multiple uses of each entity. However, each reference must have the same type in this simple version of the type system.

Exceptions We require one ordinary Schopenhauer type to be distinguished as the type of exceptions (*Err*). There is nothing special about this *user-defined type*, whose form will generally be similar to $\mu X.\{\texttt{PatternMatchFailure|DivisionByZero int|}\cdots\}$. The only pecularity about *Err* is that the argument type to all `raise` expressions must be equal to the argument type of all exception handlers. It is therefore possible to refer to type *Err* throughout an entire program. Consequently, this distinguished type is formally a part of the pervasive signature Σ, as for all normal definitions in a program. However, for the sake of brevity and readability, we will often omit it explicitly in the signature.

$$\frac{\Gamma \vdash e : \mathit{Err}}{\Gamma \vdash \texttt{raise}\ \mathrm{exn}\ e : A} \tag{Raise}$$

$$\frac{\Gamma \vdash e : A \quad \Delta \vdash \texttt{raise}\ \mathrm{exn}\ e_x : A}{\Gamma, \Delta \vdash e\ \texttt{within}\ q\ \texttt{time}\ \texttt{raise}\ \mathrm{exn}\ e_x : A} \tag{Within Time}$$

$$\frac{\Gamma \vdash e : A \qquad \Delta \vdash \texttt{raise}_{\text{ exn}}\ e_x : A}{\Gamma, \Delta \vdash e\ \texttt{within}\ q\ \texttt{stack raise}_{\text{ exn}}\ e_x : A} \quad \text{(Within Stack)}$$

$$\frac{\Gamma \vdash e : A \qquad \Delta \vdash \texttt{raise}_{\text{ exn}}\ e_x : A}{\Gamma, \Delta \vdash e\ \texttt{within}\ q\ \texttt{heap raise}_{\text{ exn}}\ e_x : A} \quad \text{(Within Heap)}$$

Substructural rules It is a noteworthy feature of our type system that all[1] substructural type rules have been made explicit, where substructural type rules are type rules which are not syntax-driven, i.e. have the same expression occuring in one of their premises. Removing the rules for weakening and sharing (also known as *contraction*) leaves a *linear type system*. The reason behind this design decision is that we require to control the number of references to an object in our type based resource analyses, so that we can determine when sharing occurs through the duplication of references, for example. Furthermore, explicit substructural rules allow the use of simpler rules overall, since this approach avoids merging these properties into all other rules, with consequent obfuscation. Note that additional substructural rules will be introduced in the annotated type system in section 8.3.

$$\frac{\Gamma \vdash e : C}{\Gamma, x{:}A \vdash e : C} \quad \text{(Weak)}$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash e : C \mid \phi \qquad \curlyvee(A \,|A_1, A_2)}{\Gamma, z{:}A \vdash e[z/x, z/y] : C} \quad \text{(Share)}$$

where the *sharing* relation $\curlyvee(A \,|\mathcal{A})$ is a relation between a types and multisets of types:

$$\curlyvee(A \,|\mathcal{A}) \text{ iff } \forall x \in \mathcal{A} \,.\, x = A$$

that is all elements (if any) of the multiset are equal to the single given type. While this definition may seem somewhat contrived at this point, it will become an important part of the annotated type system for the analysis, where this relation is redefined as partial map, mapping a single type and a multiset of types to set of numerical constraints.

Type Rules for Pattern Matches

The type rule for case-expressions requires another construct, dealing with pattern matches:

$$A \vdash_{\text{ pat}}\ ?\triangleright\ \Gamma$$

where *pat* is a Schopenhauer pattern to be tested against an object of type A. If the pattern matches successfully, then the resulting bindings are given by the context Γ.

$$\frac{}{\mathsf{unit} \vdash \texttt{()}\ ?\triangleright\ \varnothing} \quad \text{(Pattern Unit)}$$

$$\frac{b \in \mathbb{B}}{\mathsf{bool} \vdash b\ ?\triangleright\ \varnothing} \quad \text{(Pattern Bool)}$$

[1] That is all substructural properties except for permutation, which is built-in since we have, from the outset, defined contexts to be partial maps. Hence, we do not distinguish between the two contexts $x{:}A, y{:}B$ and $y{:}B, x{:}A$.

$$\frac{n \in \mathbb{Z}}{\mathsf{int} \vdash n \ ?\triangleright \ \varnothing} \quad \text{(Pattern Int)}$$

$$\frac{r \in \mathbb{R}}{\mathsf{float} \vdash r \ ?\triangleright \ \varnothing} \quad \text{(Pattern Float)}$$

$$\frac{c \ \texttt{is a character}}{\mathsf{char} \vdash c \ ?\triangleright \ \varnothing} \quad \text{(Pattern Char)}$$

$$\frac{s \ \texttt{is a string}}{\mathsf{string} \vdash s \ ?\triangleright \ \varnothing} \quad \text{(Pattern String)}$$

$$\frac{}{A \vdash x \ ?\triangleright \ x{:}A} \quad \text{(Pattern Var)}$$

$$\frac{}{A \vdash \_ \ ?\triangleright \ \varnothing} \quad \text{(Pattern Wild)}$$

$$\frac{\forall i \,.\, \Big(B_i \vdash \mathrm{pat}_i \ ?\triangleright \ \Gamma_i\Big)}{\mu X.\{\cdots \mid c : B_1, \ldots, B_k \mid \cdots\} \vdash c \ \mathrm{pat}_1 \cdots \mathrm{pat}_k \ ?\triangleright \ \Gamma_1, \cdots, \Gamma_k} \quad \text{(Pattern Constr)}$$

Type Rules for Boxes and Declarations

A Schopenhauer program consists of a number of box and function declarations. Formally, the program is defined by its signature Σ, which consist of two mappings: one mapping function-identifiers to their declaration (a triple consisting of: the functions body, a list of argument variables, and the functions type) and another mapping box-identifiers to box definitions.

Our translation to core Schopenhauer greatly simplifies the handling of boxes and thus the nature of box definitions, since instead of dealing with an arbitrary number of input and output wires and the possibilities of a wire being empty, a box is transformed into a single case expression which receives only a single input *bundle*. A bundle is a non-recursive datatype having only one unique constructor, which has as many arguments as the original box had input wires. These argument types are themselves non-recursive constructors, each having two unique constructors, one of which has no argument (indicating that no value is present in the wire), and one which has precisely one argument of the same type as the original wire. In this manner, all the input is encapsulated in a single type. A similar technique is used for the output. More precisely, if the original box definition was represented by a 7-tuple consisting of: two ordered lists of typed wire variables for input and output respectively, a list containing pairs of pattern-lists and expressions, the box-flag determining fair or unfair matching; the type of the possible exceptions and the exception handler; then the transformed box definition is a 6-tuple consisting of an expression, an argument variable, the fairness flag and the exception handler. Hence the box definition is much more similar to a function definition and much easier to manage within our analysis. In detail, the transformation simplifies

$$\Sigma(box) = \begin{pmatrix} [y_1{:}A_1, \ldots, y_k{:}A_k]; \\ [z_1{:}C_1, \ldots, z_h{:}C_h]; \\ \big[(\mathrm{pat}_{(1,1)}, \ldots, \mathrm{pat}_{(1,k)}; e_1), \ldots, (\mathrm{pat}_{(1,1)}, \ldots, \mathrm{pat}_{(j,k)}, e_j)\big]; \\ \mathrm{fairness}; B_x; \ e_x \end{pmatrix}$$

to the more concise form

$$\Sigma(box) = \big(e_{box}; y; A_{box} \rightarrowtail C_{box}; \text{fairness}; B_x; e_x\big)$$

where

$$\begin{aligned}
A_{box} = \;& \mu Y.\{\text{Bundle-ik} : \{\text{Wire-i1}:A_1 \,|\, \text{None-i1}\}, \ldots, \{\text{Wire-ik}:A_k \,|\, \text{None-ik}\}\} \\
C_{box} = \;& \mu Z.\{\text{Bundle-oh} : \{\text{Wire-o1}:C_1 \,|\, \text{None-o1}\}, \ldots, \{\text{Wire-oh}:C_h \,|\, \text{None-oh}\}\} \\
e_b = \;& \texttt{case boxcase } y \texttt{ of} \\
& \quad \text{Bundle-ik (Wire-i1 } \text{pat}_{(1,1)}) \cdots (\text{Wire-ik } \text{pat}_{(1,k)}) \texttt{ -> } e_1 \\
& \quad | \text{Bundle-ik (Wire-i1 } \text{pat}_{(2,1)}) \cdots (\text{Wire-ik } \text{pat}_{(2,k)}) \texttt{ -> } e_2 \\
& \quad \vdots \\
& \quad | \text{Bundle-ik (Wire-i1 } \text{pat}_{(j,1)}) \cdots (\text{Wire-ik } \text{pat}_{(j,k)}) \texttt{ -> } e_j
\end{aligned}$$

and each constructor Bundle-ik, Bundle-oh, Wire-i1, ..., Wire-ik, Wire-o1, ..., Wire-oh is assumed to be unique to box *box*, except for the wire constructors which may lead to another box inside the Schopenhauer program.

We can now define when a Schopenhauer program is well-typed. Given a set of identifiers Var of a certain Schopenhauer program, this program is well-typed if and only if

a) for all functions *fid* $\in$ Var with

$$\Sigma(fid) = \big(e_f; [y_1, \ldots, y_k]; A_1, \ldots, A_k \rightarrowtail C\big)$$

there exists a finite type derivation such that $y_1{:}A_1, \ldots, y_k{:}A_k \vdash e_f : C$ holds.

b) For all boxes *box* $\in$ Var with

$$\Sigma(box) = e_b; y; A \rightarrowtail C; \text{fairness}; B_x; e_x$$

there exists a finite type derivation such that $y{:}A \vdash e_b : C$ and $err{:}Err \vdash e_x : C$

c) For all pairs of boxes sharing a wire, type assigned to each wire is identical.

Chapter 8

Worst-Case Execution Time Analysis

Steffen Jost, Hans-Wofgang Loidl and Kevin Hammond

Abstract

This chapter describes the amortised analysis for worst-case execution time, giving a formal definition of the rules for Schopenhauer. It is based on the time costs for the formal Schopenhauer Semantics (Chapter 6), and related to actual execution costs for the Renesas M32C/85U microprocessor. The analysis serves as an exemplar for all of our resource analyses: the stack and heap analyses have a similar structure, differing only in the precise parameters and resource metrics of interest. We give some examples, and show how our analysis can be used to produce linear costing information.

8.1 Introduction

We describe a formally-based static analysis for determining Worst-Case Excecution Time (WCET) for Schopenhauer, based on the Time costs as defined in the formal Schopenhauer Semantics (Chapter 6). Our analysis consists of an elaborate type system, whose typings yield strict upper bounds on the Worst-Case Execution Time of the typed terms. Our approach is as follows:

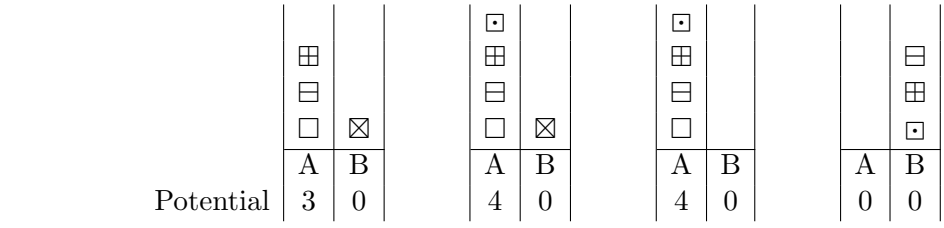
a) We enhance the standard type system for the Schopenhauer language (Chapter 7), which forms a common basis for constructing analyses, for Worst Case Execution Time (described here), Stack-space usage [81] and Heap space usage [80]. This type system is enhanced with an amortised cost analysis for Worst-Case Execution Time. in accordance to the Time costs defined in the formal Schopenhauer semantics (Chapter 5). The enhancement consists of added annotations to each type, referred to as *"Potential"*, and a number of side-conditions to each type-rule governing the use of the *potential* Worst-Case Execution Time.

b) Abstraction of the *potential* WCET then leads to a standard linear program, which can be solved easily by standard techniques. Any solution to the derived linear program that is found by the solver can then be used to assign a concrete potential to the input of a program; this potential then furnishes an upper bound on the WCET for the computation of the analysed program for that input.

Through the nature of a type-based approach to generating constraints, and since we are able to solve the generated linear program at compile time using standard, efficient solver technology this directly yields an *efficient* static WCET analysis for Schopenhauer.

8.1.1 Basic Principle of an Amortised Analysis

Our WCET analysis uses an amortised analysis approach. In amortised analysis, data structure(s) are assigned an arbitrary non-negative number, representing the *potential*, usually denoted by Φ. The amortised cost of an operation is its total cost (time or space) plus the difference in potential before and after the operation. The sum of the amortised costs plus the potential of the initial data structure then bounds (from above) the actual cost of a sequence of operations. If the potential is chosen carefully, then the amortised cost of individual operations can be either zero or a constant even when their actual cost is difficult to determine.



Operation	$\overrightarrow{\texttt{enqu}(\boxdot)}$	$\overrightarrow{\texttt{dequ}(\,) = \boxtimes}$	$\overrightarrow{\texttt{dequ}(\,) = \square}$
Actual Cost	1	1	5
Change of Potential	1	0	-4
Amortised Cost	2	1	1

Figure 8.1: Simulating a Queue by using two Stacks

The simplest example to show the usefulness of an amortised analysis is an implementation of a queue using two stacks A and B. Enqueuing is performed on stack A, and dequeuing is performed on

stack B, unless B is empty, in which case the entire contents of A are moved to B prior to dequeuing. A sequence of example operations on such a simulated queue is shown in Figure 8.1. The costs shown in this example here are for the time to execute a queue operation, but the principle can equally be used for a different resource or other countable properties as well. We have chosen to use a time rather than heap metric here since this conveys the intuition behind our approach in a straightforward way: it is easy to see that enqueuing (`enqu`$(\,)$) has a time cost of one; while dequeuing (`dequ`$(\,)$) has a variable cost, which is often one but may sometimes be proportional to the size of A.

This variable cost is unpleasant, since costing a sequence of queue operations will apparently require us to track the current state of stack A. However, if we decree that the size of A is the potential of the data then enqueuing will always have an amortised cost of 2 (one for the actual cost, one for the increase in potential) and dequeuing will always have an amortised cost of 1, since the cost of moving A over to (the empty stack) B cancels out against the decrease in potential. Thus, the actual cost of a sequence of operations is bounded by the initial size of A, plus twice the number of enqueue operations, plus the number of dequeue operations. In this case, we can also see this directly by observing that each element is moved exactly three times: once into A, once from A to B, once out of B. Therefore we require only the initial state, but do not have to model the state changes in order to determine the cost of a sequence of operations. In the above queue example, both stacks have the same type, but each element of A contributes 1 to the overall potential, whereas each element of B contributes a potential of 0. Our idea is now to record this information within the type by adding a number to each type constructor, which denotes the potential carried by each constructor of that type. So given an element of such an annotated type, we can compute its assigned potential by summing over all nodes times their assigned factor.

It is important to note that, while our analysis is entirely done at compile-time, the absolute potential can in fact be only computed at runtime, when the concrete data and its layout is known. However, we will never actually compute the potential, but rather concern ourselves with the hypothetical *change* of potential along all possible paths of computations, as expressed in our annotated type rules, which we will now define below.

8.2 Notational Preliminaries

We denote the non-negative rational numbers by $\mathbb{Q}^+$. We denote $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$, i.e., the set of non-negative real numbers together with an element ∞. Ordering and addition on $\mathbb{R}^+$ extend to $\mathbb{D}$ by $\infty + x = x + \infty = \infty$ and $x \leq \infty$. If U is a subset of $\mathbb{D}$ we write $\sum U$ for the (possibly infinite) sum over all its elements. Since $\mathbb{D}$ contains no negative numbers, questions of ordering and non-absolute convergence do not play a role; it is also the case that any subset of $\mathbb{D}$ has a sum, perhaps ∞. We write $\sum_{i \in I} x_i$ for $\sum\{x_i \mid i \in I\}$ and use other similar standard notations.

For index numbers ranging over the natural number $\mathbb{N}$, we commonly omit the limits if they are clear from the context, i.e. if $A_1, \ldots, A_k$ are defined in the surrounding context, then stating that formula Ψ_i is true for all i between 1 and k can be abbreviated as $\forall i \,.\, \Psi_i$. Furthermore, writing $\forall i \,.\, \mathcal{A}_i = \{1, \ldots, (i-1)\}$ states that $\mathcal{A}_1$ is empty, $\mathcal{A}_2$ is the singleton set containing the number one, and so on, eventually stating that $\mathcal{A}_k$ contains all natural numbers from 1 up to and excluding k.

The disjoint union of sets is denoted by $\dot\cup$. Since we are often dealing with multisets, we write $\uplus$ for the multiset union in order to distinguish it from the ordinary union of sets $\cup$.

For partial maps, we use the following notations when used to model stacks, heaps, or typing contexts: Let f be a partial map. The domain, co-domain and range (i.e. the image of the domain) of f are denoted by $\mathrm{dom}(f)$, $\mathrm{codom}(f)$ and $\mathrm{ran}\,(f)$ respectively. We may abbreviate $x \in \mathrm{dom}(f)$ by $x \in f$ and sometimes $f(x)$ by writing f_x. We denote by $f[x \mapsto y]$ the partial map that sends x to y and acts like f otherwise. Conversely, $f \setminus x$ denotes the map which is undefined for x and acts like f otherwise. The restriction of f on $\mathcal{X}$ is written $f \restriction \mathcal{X}$, i.e. the map that acts like f for all $x \in \mathcal{X}$ and

that is undefined otherwise. The empty map $\varnothing$ is often omitted, i.e. $[x \mapsto y]$ denotes the singleton map sending x to y. We write f, g for the disjoint union of the partial maps f and g. The expression f, g is undefined if the domains of f and g are not disjoint. In the special case of typing contexts, we allow ourselves to write simply $x{:}A$ for the partial maps otherwise denoted by $[x \mapsto A]$.

We write $[\,]$ to denote the empty list and $[a, b, c]$ for the list containing the elements a, b and c. The concatenation of two lists is written $l_1 +\!\!+ l_2$. If h is a suitable element and t is a (possibly empty) list, then $h :: t$ is the list obtained by prepending h to t; while $t +\!\!+ [h]$ denotes the list obtained by attaching element h at the end of list t. The cardinality of a list l is denoted by $|l|$, e.g. $|[a, b, c]| = 3$. We identify each list with its own index map, i.e. if $l = [a, b, c]$ then $l_2 = b$ according to our previously introduced abbreviations. The expression l_5 is undefined in this example. We may sometimes write $\vec{l}$ for a list l to enhance readability.

Finally, for readability within program examples, we allow ourselves to replace unimportant free variables by the underscore symbol '_', as in Hume. Multiple occurrences of the underscore symbol '_' stand for different unnamed variables as usual, hence they are not connected with each other in any way.

8.3 Determining Worst-Case Execution Time

The basic definitions of annotated types given in Chapter 7 also hold for Worst-Case Execution Time (WCET). The constructor of each variant-type is annotated with a resource variable $\in \mathsf{CV}$, representing a factor determining the potential associated with all values of this type. Note that the notion of potential is entirely independent of the resource that is being analysed. Therefore, any valid annotated type derivation will require a different *valuation*, i.e. that different values $\in \mathbb{Q}^+$ are assigned to the resource variables for different types of resource.

The type rules for expressions retain the form

$$\Sigma; \Gamma \vdash^{t}_{t'} e : A \mid \psi$$

where Γ is a context mapping variables belonging to the set Var to enriched Schopenhauer types $m, m' \in \mathsf{CV}$, e is the Schopenhauer expression, A is an enriched Schopenhauer type, and ψ is a set of constraints involving resource variables $\in \mathsf{CV}$ and constant values $\in \mathbb{Q}^+$. However, the meaning of this statement, when derived using the type rules detailed below, is now as follows: For all valuations v mapping all resource variables to $\mathbb{Q}^+$ such that the constraint set $v(\psi)$ is satisfied, the Schopenhauer expression e has type $v(A)$ in the context $v(\Gamma)$. Furthermore, for all memory configurations consisting of environment ρ and heap $\mathcal{H}$ fitting the context Γ, executing the expression e will require at most $v(m) + \Phi^{v}_{\mathcal{H}}\big(\rho : v(\Gamma)\big)$ time units (where a time unit is usually defined as a single clock cycle of the processor, but other units such as nanoseconds are also possible if desired).

Furthermore, if the computation finishes with heap $\mathcal{H}'$ and result value v, then at least $v(m') + \Phi^{v}_{\mathcal{H}'}\big(v : v(\Gamma)\big)$ time units remain unused after the computation has finished. This notion of unused time units is required for compositionality. This can be formulated in the following theorem:

Theorem 1 (Correctness)**.** *Fix a Schopenhauer program.*

$$\Gamma \vdash^{q}_{q'} e{:}A \mid \phi \tag{1.A}$$

$$\rho, \mathcal{H} \vdash e \rightsquigarrow \ell, \mathcal{H}' \tag{1.B}$$

$$\mathcal{H} \vDash_{v} \rho{:}\Gamma \tag{1.C}$$

$$v : \mathsf{CV} \to \mathbb{Q}^+, \ \textit{satisfying } \phi \tag{1.D}$$

If the four statements above are all satisfied, then it follows that for all $r \in \mathbb{Q}^+$ and $p \in \mathbb{N}$ such that $p \geq v(q) + \Phi^{v}_{\mathcal{H}}\big(\rho : v(\Gamma)\big) + r$ holds, there exists $p' \in \mathbb{N}$ satisfying $p' \geq v(q') + \Phi^{v}_{\mathcal{H}'}\big(v v(A)\big) + r$ and also $\mathcal{H}, \rho \vdash^{t}_{t'} e \rightsquigarrow_{\diamond} v, \mathcal{H}'$ for some $p, p' \in \mathbb{Q}^+$.

The statement $\mathcal{H}, \rho \vdash^{t}_{t'} e \rightsquigarrow_{\diamond} v, \mathcal{H}'$ refers to the formal Schopenhauer operational semantics (Chapter 6), as defined for Worst-Case Execution Time. This specifies that e evaluates successfully in the given memory configuration with at least t time-units available and that at least t' time-units remain unused after the evaluation is finished. It therefore *measures* the exact cost of an evaluation given some input, whereas the annotated type rules which we will now define *bound* the Worst-Case Execution Time for evaluating an expression for *all* well-typed inputs. We now define the annotated type rules for Schopenhauer expressions for *bounding* Worst-Case Execution Time:

$$\frac{}{\varnothing \vdash^{\texttt{Tmkunit}}_{0} \texttt{()} : \mathsf{unit} \mid \emptyset} \quad \text{(Unit)}$$

$$\frac{b \in \mathbb{B}}{\varnothing \vdash^{\texttt{Tmkbool}}_{0} b : \mathsf{bool} \mid \emptyset} \quad \text{(Bool)}$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash^{\texttt{Tmkint}}_{0} n : \mathsf{int} \mid \emptyset} \quad \text{(Int)}$$

$$\frac{r \in \mathbb{R}}{\varnothing \vdash^{\texttt{Tmkfloat}}_{0} r : \mathsf{float} \mid \emptyset} \quad \text{(Float)}$$

$$\frac{c \texttt{ is a character}}{\varnothing \vdash^{\texttt{Tmkchar}}_{0} c : \mathsf{char} \mid \emptyset} \quad \text{(Char)}$$

$$\frac{s \texttt{ is a string}}{\varnothing \vdash^{\texttt{Tmkstring}(|s|)}_{0} s : \mathsf{string} \mid \emptyset} \quad \text{(String)}$$

$$\frac{}{x{:}A \vdash^{\texttt{Tpushvar}}_{0} x : A \mid \emptyset} \quad \text{(Var)}$$

$$\frac{\mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash^{\texttt{Tcallprim(op)}}_{0} x \ \mathrm{op} \ y : \mathsf{int} \mid \emptyset} \quad \text{(PrimBOp Int)}$$

$$\frac{\mathrm{op} \in \{\texttt{-}\}}{x{:}\mathsf{int} \vdash^{\texttt{Tcallprim(op)}}_{0} \mathrm{op} \ y : \mathsf{int} \mid \emptyset} \quad \text{(PrimUOp Int)}$$

$$\frac{\mathrm{op} \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{x{:}\mathsf{float}, y{:}\mathsf{float} \vdash^{\texttt{Tcallprim(op)}}_{0} x \ \mathrm{op} \ y : \mathsf{float} \mid \emptyset} \quad \text{(PrimBOp Float)}$$

$$\frac{\mathrm{op} \in \{\texttt{-.}\}}{x{:}\mathsf{float} \vdash^{\texttt{Tcallprim(op)}}_{0} \mathrm{op} \ y : \mathsf{float} \mid \emptyset} \quad \text{(PrimUOp Float)}$$

$$\frac{\mathrm{op} \in \{\texttt{==}, \texttt{>=}, \texttt{<=}\}}{x{:}A, y{:}A \vdash^{\texttt{Tcallprim(op)}}_{0} x \ \mathrm{op} \ y : \mathsf{bool} \mid \emptyset} \quad \text{(PrimBOp Eq)}$$

$$\frac{\mathrm{op} \in \{\texttt{and}, \texttt{or}\}}{x{:}\mathsf{bool}, y{:}\mathsf{bool} \vdash^{\texttt{Tcallprim(op)}}_{0} x \ \mathrm{op} \ y : \mathsf{bool} \mid \emptyset} \quad \text{(PrimBOp Bool)}$$

$$\frac{\mathrm{op} \in \{\texttt{not}\}}{x{:}\mathsf{bool} \vdash^{\texttt{Tcallprim(op)}}_{0} \ \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \qquad \text{(PrimUOp Bool)}$$

$$\frac{\Sigma^{\mathcal{F}}(\mathit{fid}) = \left(\_;\_;\ \frac{t}{t'}\!\rightarrow C\right)}{\varnothing \vdash^{t}_{t'} \mathit{fid} : C \mid \emptyset} \qquad \text{(App0)}$$

$$\frac{\Sigma(\mathit{fid}) = \left(\_;\_;\forall \alpha \in \psi.A_1,\ldots,A_a \xrightarrow{\frac{t}{t'}} C\right) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t'} \mathit{fid}\ y_1 \cdots y_k : C \mid \psi} \qquad \text{(App)}$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \left(\_;\_;\forall \alpha \in \psi.A_1,\ldots,A_a \xrightarrow{\frac{t}{t'}} C\right) \qquad k \geq 1 \qquad k < a \\ \forall i \leq k\,.\,\left(\phi_i = \curlyvee(A_i \,|\, A_i, A_i)\right) \qquad \beta = \alpha \setminus \mathrm{FV}\left(A_1,\ldots,A_k\right)\end{array}}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{\texttt{Tmkfun(k)}}_{0} \mathit{fid}\ y_1 \cdots y_k : \forall \beta \in \psi.A_{k+1},\ldots,A_a \xrightarrow{\frac{t + \texttt{Tcall}}{t' + \texttt{Treturn} + \texttt{Tslide}}} C \mid \bigcup_i \phi_i} \qquad \text{(Under App)}$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \left(\_;\_;\forall \alpha \in \psi.A_1,\ldots,A_a \xrightarrow{\frac{t}{t'}} C\right) \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1,\ldots,y_a{:}A_a \vdash^{t}_{t'} \mathit{fid}\ y_1 \cdots y_a : C \mid \phi \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1},\ldots,y_k{:}A_k \vdash^{t' - \texttt{Tap}}_{t''} x\ y_{a+1} \cdots y_k : E \mid \chi\end{array}}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t'' - \texttt{Tslide}} \mathit{fid}\ y_1 \cdots y_k : E \mid \phi \cup \psi \cup \chi} \qquad \text{(Over App)}$$

$$\frac{D = \forall \alpha \in \psi.A_1,\ldots,A_a \xrightarrow{\frac{t}{t'}} C \qquad k \geq 1 \qquad k = a}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t'} z\ y_1 \cdots y_k : C \mid \psi} \qquad \text{(App Var)}$$

$$\frac{\begin{array}{c}D = \forall \alpha \in \psi.A_1,\ldots,A_a \xrightarrow{\frac{t}{t'}} C \qquad k \geq 1 \qquad k < a \\ \forall i \leq k\,.\,\left(\phi_i = \curlyvee(A_i \,|\, A_i, A_i)\right) \qquad \beta = \alpha \setminus \mathrm{FV}\left(A_1,\ldots,A_k\right) \\ B = \forall \beta \in \psi.A_{k+1},\ldots,A_a \xrightarrow{\frac{t + \texttt{Tcall}}{t' + \texttt{Treturn} + \texttt{Tslide}}} C\end{array}}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{\texttt{Tmkfun(k)}}_{0} z\ y_1 \cdots y_k : B \mid \bigcup_i \phi_i} \qquad \text{(Under App Var)}$$

$$\frac{\begin{array}{c}D = \forall \alpha \in \psi.A_1,\ldots,A_a \xrightarrow{\frac{t}{t'}} C \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1,\ldots,y_a{:}A_a \vdash^{t}_{t'} z\ y_1 \cdots y_a : C \mid \phi \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1},\ldots,y_k{:}A_k \vdash^{t' - \texttt{Tap}}_{t''} x\ y_{a+1} \cdots y_k : E \mid \chi\end{array}}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t'' - \texttt{Tslide}} z\ y_1 \cdots y_k : E \mid \phi \cup \psi \cup \chi} \qquad \text{(Over App Var)}$$

$$\frac{\begin{array}{c}c \in \mathsf{Constrs} \qquad C = \mu X.\{\cdots \mid c : q; B_1,\ldots,B_k \mid \cdots\} \\ A_i = B_i \vee (A_i = C \wedge B_i = X) \text{ (for } i = 1,\ldots,k)\end{array}}{x_1{:}A_1,\ldots,x_k{:}A_k \vdash^{q + \textsc{Size}(c)}_{0} c\ x_1 \ldots x_k : C \mid \emptyset} \qquad \text{(Constr)}$$

$$\frac{\Gamma \vdash^{t - \texttt{Tiftrue}}_{t'} e_t : A \mid \phi \qquad \Gamma \vdash^{t - \texttt{Tiffalse}}_{t' + \texttt{Tgoto}} e_f : A \mid \psi}{\Gamma, x{:}\mathsf{bool} \vdash^{t}_{t'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \phi \cup \psi} \qquad \text{(Conditional)}$$

$$\frac{\mathrm{ck} \in \{\texttt{exprcase}, \texttt{funcase}\} \qquad \forall i\,. \begin{cases} A \vdash^{\frac{t_{(i-1)} - \texttt{Tmatchrule}}{t_i}} \mathrm{pat}_i \mathrel{?\triangleright} \Delta_i; \pi_i; \psi_i \\ \Gamma, \Delta_i \vdash^{\frac{t_i' - \texttt{Tmatchedrule}}{t''}} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + t_i \geq t_i'\} \end{cases}}{\Gamma, x{:}A \vdash^{\frac{t_0 + \texttt{Tcall} + \texttt{Tcreateframe}}{t'' - \texttt{Tslide} - \texttt{Treturn} - \texttt{Tgoto}}} \texttt{case}\ \mathrm{ck}\ x\ \texttt{of}\ \mathrm{pat}_1\ \texttt{->}\ e_1 \mid \cdots \mid \mathrm{pat}_k\ \texttt{->}\ e_k : B \mid \bigcup_i \psi_i \cup \phi_i \cup \chi_i} \quad \text{(Case Expr/Fun)}$$

$$\frac{\begin{array}{c}\mathrm{ck} = \texttt{boxcase} \\ \chi_0 = \{t \geq t_0 + \texttt{Tcall} + \texttt{Tcreateframe}\}\end{array} \qquad \forall i\,. \begin{cases} A \vdash^{\frac{t_{(i-1)} - \texttt{Tmatchrule}}{t_i}} \mathrm{pat}_i \mathrel{?\triangleright} \Delta_i; \pi_i; \psi_i \\ \Gamma, \Delta_i \vdash^{\frac{t_i' - \texttt{Tmatchedrule}}{t''}} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + t_i \geq t_i' + t_0\} \end{cases}}{\Gamma, x{:}A \vdash^{\frac{t_0}{t'' - \texttt{Tslide} - \texttt{Treturn} - \texttt{Tgoto}}} \texttt{case}\ \mathrm{ck}\ x\ \texttt{of}\ \mathrm{pat}_1\ \texttt{->}\ e_1 \mid \cdots \mid \mathrm{pat}_k\ \texttt{->}\ e_k : B \mid \bigcup_i \psi_i \cup \phi_i \cup \chi_i} \quad \text{(Case Box)}$$

Note the different definition of χ_i here compared with that used for expressions: A coordination-layer case-expression has no initial potential for the cost of the pattern match itself. Hence, we use a new variable t to pay for this cost, which must then be justified from the potential that is gained during the pattern match, which we know must have succeeded for this rule to be used.

$$\frac{\forall i\,. \begin{cases} \Delta_i = \{x_1{:}A_1^i, \ldots, x_{(i-1)}{:}A_{(i-1)}^i\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \psi_i = \curlyvee(A_i \mid \mathcal{A}_i) \\ \Delta_i, \Gamma_i \vdash^{\frac{t_{(i-1)} - \texttt{Tmakevar}}{t_i}} e_i : A_i \mid \phi_i \end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{\frac{t_0}{t_{k+1}}} \texttt{LET}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{IN}\ e : A \mid \bigcup_i \phi_1 \cup \psi_i} \quad \text{(Ghost let)}$$

$$\frac{\forall i\,. \begin{cases} \Delta_i = \{x_1{:}A_1^i, \ldots, x_{(i-1)}{:}A_{(i-1)}^i\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \psi_i = \curlyvee(A_i \mid \mathcal{A}_i) \\ i \neq (k+1) \implies \Delta_i, \Gamma_i \vdash^{\frac{t_{(i-1)} - \texttt{Tmakevar}}{t_i}} e_i : A_i \mid \phi_i \end{cases} \qquad \Delta_{(k+1)}, \Gamma_{(k+1)} \vdash^{\frac{t_k}{t_{(k+1)}}} e_{k+1} : A_{k+1} \mid \phi_{(k+1)}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{\frac{t_0 + \texttt{Tcall} + \texttt{Tcreateframe}}{t_{(k+1)} - \texttt{Tgoto} - \texttt{Treturn}}} \texttt{let}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{in}\ e_{k+1} : A_{k+1} \mid \bigcup_i \phi_i \cup \psi_i} \quad \text{(Let)}$$

Exceptions

$$\frac{\Gamma \vdash^{\frac{t}{t'}} e : \mathit{Err} \mid \psi}{\Gamma \vdash^{\frac{t + \texttt{Traise}}{t'}} \texttt{raise}_{\,\mathrm{exn}}\ e : A \mid \psi} \quad \text{(Raise)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\frac{t}{t' + \texttt{Tgoto} + \texttt{Tdonewithin}}} e : A \mid \phi \\ \Delta \vdash^{\frac{t' + \texttt{Traisewithin}}{t''}} \texttt{raise}_{\,\mathrm{exn}}\ e_x : A \mid \psi\end{array}}{\Gamma, \Delta \vdash^{\frac{t + \texttt{Twithin}}{t''}} e\ \texttt{within}\ q\ \texttt{time raise}_{\,\mathrm{exn}}\ e_x : A \mid \phi \cup \psi} \quad \text{(Within Time)}$$

$$\frac{\Gamma \vdash^{t}_{t' + \texttt{Tgoto} + \texttt{Tdonewithin}} e : A \mid \phi \quad \Delta \vdash^{t' + \texttt{Traisewithin}}_{t''} \texttt{raise}_{\,\text{exn}}\, e_x : A \mid \psi}{\Gamma, \Delta \vdash^{t + \texttt{Twithin}}_{t''} e\ \texttt{within}\ q\ \texttt{stack raise}_{\,\text{exn}}\, e_x : A \mid \phi \cup \psi} \quad \text{(Within Stack)}$$

$$\frac{\Gamma \vdash^{t}_{t' + \texttt{Tgoto} + \texttt{Tdonewithin}} e : A \mid \phi \quad \Delta \vdash^{t' + \texttt{Traisewithin}}_{t''} \texttt{raise}_{\,\text{exn}}\, e_x : A \mid \psi}{\Gamma, \Delta \vdash^{t + \texttt{Twithin}}_{t''} e\ \texttt{within}\ q\ \texttt{heap raise}_{\,\text{exn}}\, e_x : A \mid \phi \cup \psi} \quad \text{(Within Heap)}$$

Substructural rules

$$\frac{\Gamma, x{:}B \vdash^{t}_{t'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{t}_{t'} e : C \mid \psi \cup A <: B} \quad \text{(Supertype)}$$

$$\frac{\Gamma \vdash^{t}_{t'} e : C \mid \psi}{\Gamma \vdash^{t}_{t'} e : D \mid \psi \cup C <: D} \quad \text{(Subtype)}$$

$$\frac{\Gamma \vdash^{r}_{r'} e : A \mid \psi}{\Gamma \vdash^{t}_{t'} e : A \mid \psi \cup \{t \geq r, t - r \geq t' - r'\}} \quad \text{(Relax Time)}$$

$$\frac{\Gamma \vdash^{t}_{t'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{t}_{t'} e : C \mid \psi} \quad \text{(Weak)}$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash^{?}_{?} e : C \mid \phi}{\Gamma, z{:}A \vdash^{t}_{t'} e[z/x, z/y] : C \mid \phi \cup \curlyvee(A \,|\, A_1, A_2)} \quad \text{(Share)}$$

The definitions for sharing $\curlyvee(A \,|\, A_1, A_2, \dots)$ and subtyping $A <: B$ of Chapter 7 remain unaltered since they deal with an abstract notion of potential obtained by manipulating resource variables in a general, resource-independent manner

8.3.1 Annotated Type Rules for Pattern Matches

The annotated type rules dealing with pattern matches have the form

$$A \vdash^{t}_{t'} \text{pat} \ ?\triangleright\ \Gamma; \pi; \psi$$

where *pat* is a Schopenhauer pattern to be tested against an object of enriched Schopenhauer type A. If the pattern matches successfully, then the resulting bindings are given by the context Γ; the released potential is given in π as a linear combination of resource variables; and any constraints that arise are collected in ψ. The time required to test this pattern is at most t time-units, of which at least t' remain after the pattern match is finished, regardless of the outcome of the test.

$$\frac{}{\mathsf{unit} \vdash^{\texttt{Tmatchunit}}_{0} () \ ?\triangleright\ \varnothing; 0; \emptyset} \quad \text{(Pattern Unit)}$$

$$\frac{b \in \mathbb{B}}{\mathsf{bool} \vdash^{\texttt{Tmatchbool}}_{0} b \ ?\triangleright\ \varnothing; 0; \emptyset} \quad \text{(Pattern Bool)}$$

$$\frac{n \in \mathbb{Z}}{\mathsf{int} \vdash^{\frac{\texttt{Tmatchint}}{0}} n \ ?\triangleright \ \varnothing; 0; \emptyset} \quad \text{(Pattern Int)}$$

$$\frac{r \in \mathbb{R}}{\mathsf{float} \vdash^{\frac{\texttt{Tmatchfloat}}{0}} r \ ?\triangleright \ \varnothing; 0; \emptyset} \quad \text{(Pattern Float)}$$

$$\frac{c \ \texttt{is a character}}{\mathsf{char} \vdash^{\frac{\texttt{Tmatchchar}}{0}} c \ ?\triangleright \ \varnothing; 0; \emptyset} \quad \text{(Pattern Char)}$$

$$\frac{s \ \texttt{is a string}}{\mathsf{string} \vdash^{\frac{\texttt{Tmatchstring}(|s|)}{0}} s \ ?\triangleright \ \varnothing; 0; \emptyset} \quad \text{(Pattern String)}$$

$$\frac{}{A \vdash^{\frac{\texttt{Tmatchvar}}{0}} x \ ?\triangleright \ x{:}A; 0; \emptyset} \quad \text{(Pattern Var)}$$

$$\frac{}{A \vdash^{\frac{\texttt{Tmatchany}}{0}} \_ \ ?\triangleright \ \varnothing; 0; \emptyset} \quad \text{(Pattern Wild)}$$

$$\frac{\forall i \,.\, \left(B_i \vdash^{\frac{t_{i-1}}{t_i}} \mathrm{pat}_i \ ?\triangleright \ \Gamma_i; \pi_i; \phi_i\right)}{\mu X.\{\cdots \mid c : q; B_1, \ldots, B_k \mid \cdots\} \vdash^{\frac{t_0 + \texttt{Tcopy} + \texttt{Tunpack} + \textsc{PatSize}(c)}{t_k - \texttt{Tpop}}} c \ \mathrm{pat}_1 \cdots \mathrm{pat}_k \ ?\triangleright \ \Gamma_1, \cdots, \Gamma_k; q + \Sigma_i \pi_i; \bigcup_i \phi_i} \quad \text{(Pattern Constr)}$$

Substructural Rules

$$\frac{A \vdash^{\frac{r}{r'}} \mathrm{pat} \ ?\triangleright \ \Gamma; \pi; \phi}{A \vdash^{\frac{t}{t'}} \mathrm{pat} \ ?\triangleright \ \Gamma; \pi; \phi \cup \{t \geq r, t - r \geq t' - r'\}} \quad \text{(Pattern Relax Time)}$$

For WCET the rules for pattern matches require the auxiliary definition of $\textsc{PatSize} : \mathsf{Constrs} \to \mathbb{Q}^+$, which maps a constructors c to the number of time-units required to match any value to this constructor, which is in general equal to `Tmatchcon`. However, in the case of Tuples or Vectors, the WCET for a pattern match is significantly lower. This auxiliary definition is not required for the space analyses.

8.3.2 Annotated Type Rules for Boxes and Declarations

Given a set of identifiers Var of a certain Schopenhauer program, this program is well-typed if and only if

a) for all functions $\mathit{fid} \in \mathsf{Var}$ with

$$\Sigma(\mathit{fid}) = \left(e_f; [y_1, \ldots, y_k]; A_1, \ldots, A_k \xrightarrow{\frac{t}{t'}} C\right)$$

there exists a finite type derivation such that $y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{\frac{t}{t'}} e_f : C$ holds.

b) For all boxes $\mathit{box} \in \mathsf{Var}$ with

$$\Sigma(\mathit{box}) = e_b; y; A \xrightarrow{\frac{t}{t'}} C; \text{fairness}; B_x; e_x \mid \phi$$

there exists a finite type derivation such that $y{:}A \vdash^{\frac{t_x}{t'}} e_b : C \mid \phi$ and $\mathit{err}{:}\mathit{Err} \vdash^{\frac{t}{t_x}} e_x : C \mid \psi$

c) For all pairs of boxes sharing a wire, type assigned to each wire is identical, including the resource variable denoting the potential communicated through the wire.

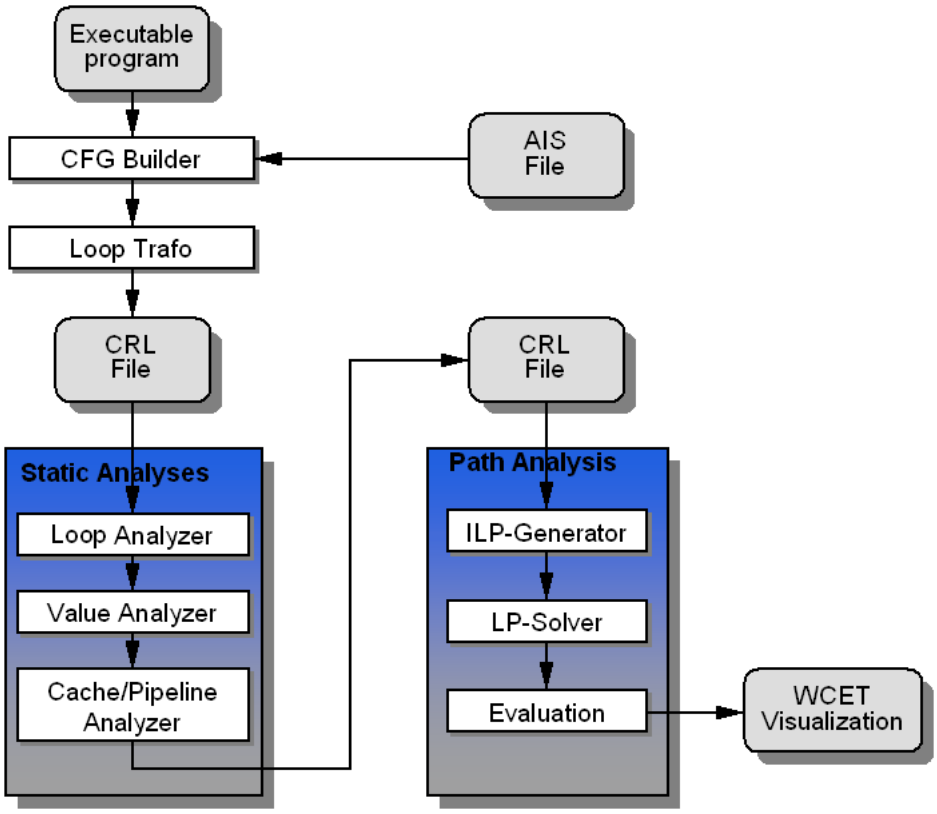


Figure 8.2: Phases of WCET computation

8.3.3 Timing Complete Hume Programs

The model we have provided is capable of determining all time costs within a Schopenhauer box. Clearly, it is not sensible, in general, to time a complete Hume program, since Hume programs are not usually expected to terminate. However, it is sensible to determine reactivity times to specific inputs. By using the models derived here, we are able to determine this for individual Schopenhauer boxes, since we can determine upper bound times on all costs from matching inputs on wires to producing wire outputs. In this way, we are, in principle, able to verify timing constraints provided by Schopenhauer programmers within a single box, for programs that include data structures and recursion.

Extending these costs to cover sequences of multiple boxes involves obtaining measurements of inter-box scheduling time, and then combining these using models of multiple box behaviours. Since boxes are combined using a static process network, and the range of inputs and outputs accepted by/produced by each rule can also be determined statically, it follows that we would then be able to construct a complete model of input-output WCET for any input-output combination. We intend to study this extension in the longer term.

8.4 Concrete Time Costs for the Renesas M32C/85 Processor

In the WCET analysis described in the previous sections, we have used a set of manifest constants rather than actual costs. This allows our static amortised WCET analysis to be generalised to different types of processor. In this section, we describe how we have obtained preliminary costs for the Renesas M32C/85 microprocessor using the **aiT** tool, and verified these against concrete measurements. Further details can be found in Bonenfant et al. [15].

8.4.1 Determining WCET using the aiT tool

The **aiT** tool determines the worst-case execution time of a program task in several phases, as shown in Figure 8.2. These phases are:

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from an executable binary program;
- **Value Analysis** computes address ranges for instructions accessing memory;
- **Cache Analysis** classifies memory references as cache misses or hits [40];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [89];
- **Path Analysis** determines a worst-case execution path of the program [144].

The cache analysis phase uses the results of the value analysis phase to predict the behavior of the (data) cache based on the range of values that can occur in the program. The results of the cache analysis are then used within the pipeline analysis to allow prediction of those pipeline stalls that may be due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of specific program paths. By separating the WCET determination into several phases, it becomes possible to use different analysis methods that are tailored to the specific subtasks. Value analysis, cache analysis, and pipeline analysis are all implemented using abstract interpretation [31], a semantics-based method for static program analysis. Integer linear programming is then used for the final path analysis phase.

Whilst the analysis works at a level that is more abstract than simple basic blocks, it is not capable of managing the complex high-level constructs that we require. It can, however, provide useful and accurate worst-case time information about lower level constructs. We are thus motivated to link the two levels of analysis, combining information on recursion bounds and other high-level constructs that we will obtain from the Hume source analysis we are constructing, with the low-level worst-case execution time analysis that can be obtained from the AbsInt analysis. In order to achieve this, we will eventually require two-way information flow between the analyses. In the short-term, it is sufficient to provide one-way flow from the language-level analysis to the lower-level analysis. The use of an abstract machine as the analysis target represents a new challenge for the **aiT** tool, since the structure of instructions that need to be analysed can be significantly different from those that are hand-produced, and the associated technical problems in producing cost information can therefore be more complex.

8.4.2 HAM Instruction WCET Costs for Renesas M32C/85

Figure 8.3 lists guaranteed worst-case execution time results, in clock cycles, for a subset of Hume Abstract Machine instructions, ordered alphabetically. These timings were obtained using the **aiT** tool from code generated using the **ham2c** Hume to C compiler, cross-compiling through either gcc Version 3.4 or the IAR C compiler [138] to the Renesas M32C. As expected from a commercial compiler targeting a few architectures, the IAR compiler generally produces more efficient code than gcc, with our results being 42% lower on average, and up to 5.92 times more efficient in the case of **MatchInt**. In a few cases, the **aiT** tool was unable to provide timing information directly, requiring additional information such as loop bounds to be provided in order to produce timing results. In the long term, we anticipate that we will be able to provide this information by analysis of Hume source constructs, and by modifying the HAM code to include type information and other information that can be exploited by the **aiT** tool. In the short term, we have calculated the information by hand, as far as possible. For some instructions, however, we were unable to provide this information for the IAR-compiled code, and results for these instructions are therefore given only for gcc-produced code.

Instruction	gcc	IAR	Ratio
Call	73	70	1.04
Copy	43		
CopyArg	40	35	1.14
CreateFrame	76	72	1.06
Goto	5	3	1.67
If (true)	41	32	1.28
If (false)	41	32	1.28
MakeVar	43	36	1.19
MatchExn	808		
MatchedRule	11	11	1.00
MatchInt	811	137	5.92
MatchRule	22	22	1.00
MatchVar	46	36	1.28
MkBool	136		

Instruction	gcc	IAR	Ratio
MkChar	136		
MkCon 2	348	242	1.44
MkFun 0	198	165	1.20
MkInt	136	91	1.49
MkNone	26	21	1.24
MkVector 3	392	205	1.91
Pop	13		
Push	12	11	1.09
PushVar	40	35	1.14
Return	1756		
Schedule	410	602	0.68
Slide	62	53	1.17
SlideVar	94		
TailCall	91	178	0.51

Figure 8.3: aiT HAM analysis: gcc and IAR compiled code

8.4.3 Timing Results

Figure 8.4 shows average execution and worst-case execution times obtained using the timing approach described above, for HAM instructions compiled using the IAR compiler. Each average and worst-case entry has been obtained from 10000 individual timings. We can see from the table that the worst-case times and average-case times are very similar for most instructions, indicating that the instruction timings are highly consistent in practice. Since certain instructions are parameterised on some argument (for example, **MkVector** is parameterised on the vector size), in these cases, we have measured several points and applied linear interpolation to obtain a cost formula. It is interesting to note that in these case, the linear factor is identical for both WCET and average times and the constants are also very close. In each case, we have subtracted the least time obtained from timing the empty sequence of instructions (39 clock cycles), in order to give a conservative worst-case time. Since the worst-case time for the empty sequence was 42 cycles, this means that the worst-case may, in fact, be up to three cycles less than the numbers reported here.

Since we must save and restore system state, we needed to develop code that does this correctly. A few abstract machine instructions have therefore not been costed, mainly because they perform more complex state changes that may require additional intervention. It is worth noting that the values included in this table give a good timing predictor, but one that could only be used to provide absolute worst-case guarantees under some statistical probability.

8.4.4 Quality of the Static Analysis using the aiT Tool

Figure 8.5 compares the upper bounds on worst-case execution timing obtained using the aiT tool from Figure 8.3 with the corresponding measured worst cases from Figure 8.4. We can see that in all cases apart from **MatchRule**, the static analyis gives an upper bound that is greater than or equal to the measured execution time. For **MatchRule**, the static analysis yields an upper bound that is one cycle smaller than our measured worst-case. Since our worst case timings are conservative, and may have an experimental error of up to three clock cycles, as described above, we conclude that the static analysis correctly yields upper bounds on execution costs for these HAM instructions. For the instructions we have compared, the bound given by the static analyis is at most 50% greater than the

Instructions	AVG	WCET	Ratio
Ap	760	761	1.00
Call	61	62	1.02
Callprim			
== Bool	246	251	1.02
∗ Float	240	242	1.01
+ Float	262	267	1.02
== Float	255	260	1.02
− Int	114	119	1.04
∗ Int	130	132	1.02
/ Int	168	177	1.05
+ Int	114	119	1.04
< Int	215	220	1.02
== Int	216	221	1.02
> Int	217	223	1.03
Consume	27	31	1.24
Copy	27	31	1.15
CopyArg	27	30	1.11
CreateFrame	51	57	1.12
Goto	1	2	2.00
If (true)	24	29	1.21
If (false)	24	26	1.08
MakeVar	26	31	1.19
MatchAny	6	10	1.67
MatchAvailable	7	10	1.43
MatchBool	24	29	1.21
MatchCon	22	26	1.18
MatchedRule	8	12	1.50
MatchExn	22	28	1.27
MatchFloat	24	29	1.21
MatchInt	23	29	1.26

Instructions	AVG	WCET	Ratio
MatchNone	6	10	1.67
MatchRule	18	23	1.28
MatchString n	3 × n + 45	3 × n + 47	
MatchTuple	6	10	1.67
MatchVar	26	31	1.19
MaybeConsume	20	28	1.40
MkBool	63	70	1.11
MkChar	63	70	1.11
MkCon n	41 × n + 84	41 × n + 89	
MkFun n	42 × n + 108	42 × n + 113	
MkInt	64	65	1.02
MkNone	15	21	1.40
MkString n	13 × n + 133	13 × n + 140	
MkTuple n	41 × n + 63	41 × n + 66	
MkVector n	41 × n + 63	41 × n + 65	
Pop	6	9	1.50
Push	6	9	1.50
PushVar	27	30	1.11
PushVarF	37	40	1.08
Raise	374	377	1.01
Return	112	116	1.04
Slide	41	44	1.07
SlideVar	58	63	1.09
Unpack	114	118	1.04

Figure 8.4: Experimental average and worst-case timings for HAM instructions

Instructions	aiT bound	Measured WCET	Ratio
Call	70	62	1.13
CopyArg	35	30	1.17
CreateFrame	72	57	1.26
Goto	3	2	1.50
If (true)	32	29	1.10
If (false)	32	26	1.23
MakeVar	36	31	1.16
MatchRule	**22**	**23**	**0.96**

Instructions	aiT bound	Measured WCET	Ratio
MatchVar	36	31	1.16
MkCon 2	242	170	1.42
MkFun 0	165	113	1.46
MkInt	91	65	1.40
MkNone	21	21	1.00
Push	11	9	1.22
PushVar	35	30	1.17
Slide	53	44	1.20

Figure 8.5: Quality of the aiT Analysis

measured worst-case (for **Goto**, representing a difference of only one clock cycle); the mean difference is 22%, with a standard deviation of 16%. We conclude that the static analysis provides an accurate upper bound on execution time.

8.5 Simple Analysis Examples

In this section we present several examples of running our analysis on some simple Hume programs. We first consider the costs of simple expressions and functions, then extend this to programs including boxes.

For illustrative purposes, we have chosen to use the *measured* worst-case execution times for the Renesas M32C/85 in this section rather than those obtained using the **aiT** tool. Since we have only obtained analytical **aiT** timings for a relatively small subset of the HAM instructions, but have a near-complete set of measured WCET timings, this approach allows us to cover a wider range of applications, and to demonstrate more features of our analytical framework and implementation. We would emphasise that the figures given here are consequently only worst-case, and not formally guaranteed bounds on execution time. However, we will easily be able to replace these timings with the analytical timings, once they become available.

8.5.1 Example: factorial function

Our first example is the factorial function, implemented over floating point numbers rather than natural numbers. We chose to use a floating point representation because such values are commonly used in the computer vision domain that is one of our targets, and also because floating-point values tend to be more difficult to handle in analyses than, for example, natural numbers. Additionally to annotating all user-defined datatypes with potentials, as defined in the previous section, we employ a datatype-independent interval analysis, which aims to statically deduce ranges for the possible value of a variable at runtime, and propagate this information to the recursive call in the function body. Thus, we are able to deal with recursion not restricted to special types such as natural numbers or linearly structured datatypes such as lists, and in particular we can handle floats. A current limitation of the interval analysis is its intra-procedural (or -functional) nature. We plan to extend it to cope with more complex recursion patterns in the near future. The Hume code for the obvious factorial function is:

```
program

type _float = float 32;

fac n = if (n==0.0)
          then 1.0
          else n * (fac (n - 1.0));

expression (fac);
```

In the first stage of the analysis this code is translated into an intermediate format:

```
program

-- type of main
val main :: float, ->float
-- Functions
{fac :: float, ->float (n :: float) =
 glet ?z_1    = 0.0
 in glet ?z_2    = n==?z_1
    in if ?z_2
         then 1.0
         else glet ?z_3    = 1.0
              in glet ?z_4    = n-.?z_3
```

```
                in glet ?z_5    = (fac <> ?z_4)
                   in n*.?z_5}
-- Boxes
-- Expression:
(fac)
```

Desugaring the program into our core syntax produces a new function body for `fac`, where (possibly nested) case expressions are used to match against given patterns (if any). Note that in this format all functions are in let-normal-form, with internally created variables always starting with an `?`.. Furthermore, all overloaded binary operators have been instantiated with their monomorphic counterparts (where `*.` stands for multiplication on floating-point numbers etc). Function calls are specially annotated to indicated whether they have the specified number of arguments or are over- or under-saturated. In this example, we see that we have a call to `fac` with the specified number of arguments (1), which is indicated by the `<>` operator. The worst-case time consumption is given in terms of the following (rich) type of the main function:

```
ARTHUR3 typing for resource "Time":
  30, (float<1043>) -501/0-> float<59> ,0
```

This type indicates that, for an input value of n, the execution of the main function will require $1043n + 501$ machine cycles, plus 30 cycles to set-up the main expression (for the other examples we will ignore this set-up time). An interval analysis, which aims to statically deduce ranges for the possible value of a variable at runtime, is performed on the floating-point variables, to generate linear (in-)equality constraints. These constraints are then solved by a separate LP-solver.

We now analyse a variant of the factorial function, for which our heap inference was able to give tighter bounds.

```
program

type _float = float 32;

fac :: _float -> _float;
fac 0.0 = 1.0 ;
fac n = n * (fac (n - 1.0));

expression (fac);
```

The only difference to the previous version is the use of top-level pattern matching rather than an explicit conditional in the body. When translated to intermediate code this gives:

```
program

-- type of main
val main :: float, ->float
-- Functions
{fac :: float, ->float (?arg_11 :: float) =
 case ?arg_11 of
   (0.0) -> 1.0|
   (n) -> glet ?z_1    = 1.0
          in glet ?z_2    = n-.?z_1
             in glet ?z_3    = (fac <> ?z_2)
                in n*.?z_3
```

```
 esac}
-- Boxes
-- Expression:
(fac)
```

The key difference for the analysis is that now the variable `n` is used in only one branch, whereas before it was used outside the recursion case, namely in the head of the conditional, as well. Such usage requires a sharing of the associated potential and causes weaker bounds to be inferred.

For this version the rich type of the main expression is:

```
ARTHUR3 typing for resource "Time":
  30, (float<785>) -245/0-> float<116> ,0
```

i.e. for an input n, in total $785n + 245$ cycles are needed. So, we see a considerable improvement of runtime (as well as heap usage) for this version, wheras only slightly more stack space is required. It seems that this version is the clear winner in this head-to-head comparison.

8.5.2 Example: sum-over-list

The next example infers the costs for a list-traversing function, computing the sum over a list of float values.

```
type _float = float 32;

data flist = Cons _float flist | Nil;

sum11 :: flist -> _float;
sum11 (Nil) = 0.0 ;
sum11 (Cons f fs) = f + (sum11 fs);

expression sum11;
```

The intermediate code for this example shows how a function with pattern matching is translated into (possibly nested) case statements. The overloaded multiplication operation on Hume-level is instantiated to a monomorphic `*.` over floats.

```
program

type flist = Cons {-2-} float  flist | Nil {-0-}

-- type of main
val main :: flist, ->float

-- Functions
{sum11 :: flist, ->float (?arg_11 :: flist) =
 case ?arg_11 of
   ((Nil)) -> 0.0|
   (Cons f  fs) -> glet ?z_1    = (sum11 <> fs)
                   in f+.?z_1
 esac}
-- Boxes
-- Expression:
sum11
```

Unsurprisingly the time consumption of the main expression, namely the function `sum11`, is linear in the length of the input list, as shown by the following (rich) type:

```
ARTHUR3 typing for resource "Time":
  30, (flist[934;float<0>,#|0]) -476/126-> float<0> ,0
```

For every `Cons` node of the list, represented as `float<0>,#` in the type, 934 cycles are needed to perform the computation. In total, this gives a worst-case time consumption of $934n + 476$ for an input list of length n.

8.5.3 Example: multiplication (box- vs expression-level)

One powerful technique to infer costs for complex (recursive) functions is to transform the program by "lifting" functions from the expression-level to the box-level. This has been frequently used in the past to cost Hume programs. The following example implements multiplication over floating-point numbers in terms of repeated addition operations. By lifting the recursion to box level, and encoding the state of the recursion in 3 wires, we obtain an iterative version with 2 boxes. The `mult2` box drives the computation by feeding the two input values that should be multiplied along the output wires `iter1`, `iter2` and `iter3` to the worker box `itermult`. The latter takes input from wires `i1`, `i2` and `i3`, and uses the output wires `iter1'`, `iter2'` and `iter3'` to hold the state of the computation by feeding them back to its own input wires `iter1`, `iter2` and `iter3`.

```
program

type _float = float 64;

stream stdin1 to "std_in";
stream stdin2 to "std_in";
stream stdout to "std_out";

-- takes 2 floats as input and initialises the inputs
-- for the itermult box, which does the main work
box mult2
in (i::_float, j::_float)
out (iter1 ::_float, iter2 ::_float, iter3 ::_float)
match
(x,y) -> (0.0,x,y);

stream output to "std_out";

wire mult2 (stdin1, stdin2)
           (itermult.i1,itermult.i2,itermult.i3);

-- computing (_,x,y,_,_,_,_) -> x*y, using the other last 3 inputs
-- and first 3 outputs as state via feedback wires
box itermult
in  (i1::_float, i2::_float, i3::_float, iter1::_float, iter2::_float, iter3::_float)
out (iter1'::_float, iter2'::_float, iter3'::_float, r::_float)
match
 (r,x,y,*,*,*) -> (r,x,y,*) |
 (*,*,*,r,x,y) -> if y==0.0
                  then (  *, *,       *, r)
                  else (r+x, x, y - 1.0, *);
```

```
wire itermult
 (mult2.iter1,mult2.iter2,mult2.iter3,itermult.iter1',itermult.iter2',itermult.iter3')
 (itermult.iter1,itermult.iter2,itermult.iter3,output);
```

It is easy to see that the computation in both boxes represents code on FSM Hume level, since no recursion is used. Inferring the worst-case execution time we get the following rich type:

```
ARTHUR3 typing for resource "Time":

Box: mult2
 ?v_8: wire1float[1073;float<0>|*], ?v_9: wire1float[0;float<1659.5>|*]
---621/0--->
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[0;float<1659.5>|*]

Box: itermult
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[0;float<1659.5>|*],
 ?v_4: wire1float[0;float<0>|*], ?v_5: wire1float[0;float<0>|*], ?v_6: wire1float[796.5;float<1659.5>|*]
---3588.5/0--->
 ?v_4: wire1float[0;float<0>|*], ?v_5: wire1float[0;float<0>|*], ?v_6: wire1float[796.5;float<1659.5>|*],
 ?v_7: wire1float[0;float<0>|*]
```

In this case the time consumption of the `itermult` box is linear in the input wires `?v_3` and `?v_6`. Altogether for input `y` (along wire `?v_3`) of size n and input `y` (along wire `?v_6`) of size m $1659.5n + 1659.5m + 796.5$ cycles are needed by the `itermult` box. This bound is in fact significantly weaker than we had hoped for, and we are currently examining the exact reason.

To demonstrate the usefulness of our resource inference, we now look at a function, directly implementing the computation as recursive program on the Hume expression level.

```
program

stream stdin1 to "std_in";
stream stdin2 to "std_in";
stream stdout1 to "std_out";
stream stdout2 to "std_out";

type _float = float 32;

dec :: _float -> _float;
dec x = x - 1.0;

mult :: _float -> _float -> _float -> _float;
mult r x y = if y==0.0
               then r
               else mult (r+x) x (dec y);


box mult13
in  (i  :: _float, j  :: _float)
out (i' :: _float, o :: _float)
match
 (x, y) -> (1.0, mult 0.0 x y);
```

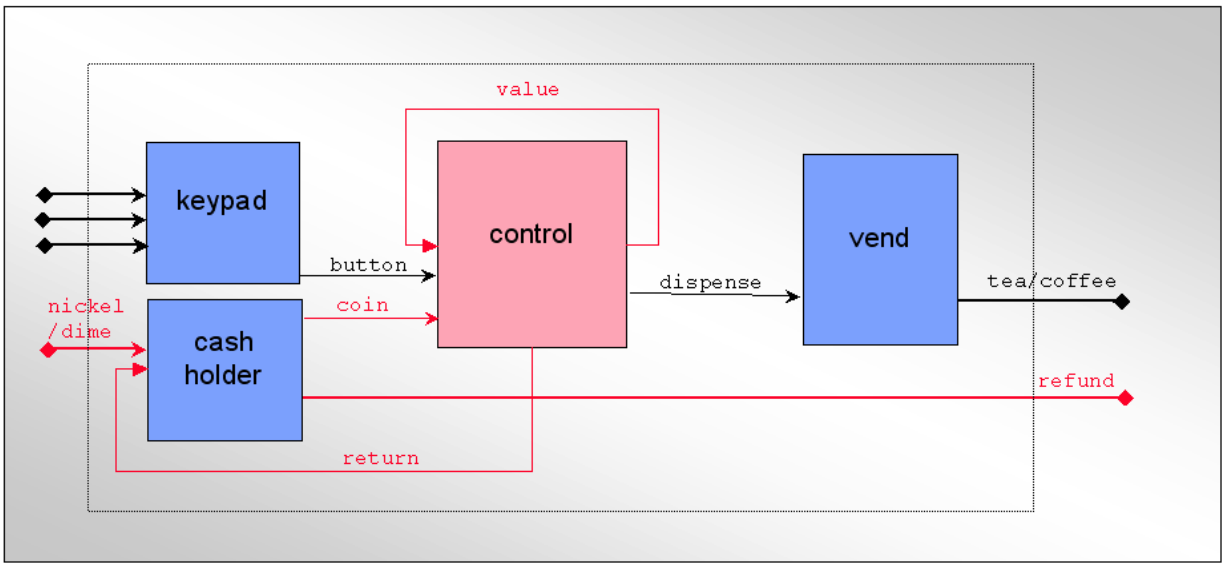


Figure 8.6: Hume example: vending machine box diagram

```
wire mult13 (stdin1, stdin2) (stdout1, stdout2);
```

Now `mult` is a recursive function and the overall program is on PR Hume level. Still, we can derive an upper bound for the box `mult13`, which directly calls this recursive function:

```
Solution has been written to file constraints.solved
ARTHUR3 typing for resource "Time":

Box: mult13
    ?v_1: wire1float[1428;float<0>|*], ?v_3: wire1float[0;float<1068>|*]
   ---621/0--->
    ?v_2: wire1float[0;float<0>|*], ?v_4: wire1float[0;float<0>|*]
```

The costs are linear in the second input and constant in the first input. For an input `j` (along wire `?v_3`) of size n the box needs $1068n + 1468 + 621$ cycles. We see that with this inference, the programmer can write much simpler code to implement the program, and still obtain upper bounds for the resource consumption.

8.5.4 Example: vending machine

The next example simulates the behaviour of a simple drinks vending machine [**?**]. A system diagram is shown in Figure 8.6. We will show Hume code only for the most important 3 boxes of the system. The *control* box (`coffee`) responds to inputs from the keypad box (`inp`) and the cash holder box representing presses of a button (for tea, coffee, or a refund) or coins (nickels/dimes) being loaded into the cash box. If a drinks button (tea/coffee) is pressed, then the controller determines whether a sufficient value of coins has been deposited for the requested drink using the `vend` function. If so, the vending unit (`outp`) is instructed to produce the requested drink. Otherwise, the button press is ignored.

```
-- Vending machine example, (c) K.Hammond, University of St Andrews
-- after specification by Pieter Koopman, CEFP July 2005
```

```
type _float = float 32;

data coins =  Nickel | Dime;
data drinks = Coffee | Tea ;
data buttons = BCoffee | BTea | BCancel;

-- input handling box
box inp
in ( c :: char )
out ( coin :: coins, button :: buttons )
match
    'N' -> ( Nickel, * )
  | 'D' -> ( Dime,   * )
  | 'C' -> ( *,      BCoffee )
  | 'T' -> ( *,      BTea )
  | 'X' -> ( *,      BCancel )
  | _   -> ( *,      * )  ;

-- coffee vending box

vend :: drinks->_float->_float->(drinks,_float,_float);
vend drink cost v = if v >= cost then (drink, v-cost, *) else (*, v, * );

box coffee
in  ( coin :: coins, button :: buttons, value :: _float )
out ( drink :: drinks, value' :: _float, return :: _float )
match
    ( Nickel, *,       v )  ->  ( *, v + 5.0, * )
  | ( Dime,   *,       v )  ->  ( *, v + 10.0, * )
  | ( *,      BCoffee, v )  ->  vend Coffee 10.0 v
  | ( *,      BTea,    v )  ->  vend Tea 5.0 v
  | ( *,      BCancel, v )  ->  ( *, 0.0, v )  ;

showdrink :: drinks->(int 2);
showdrink Coffee = 0;
showdrink Tea = 1;

-- output handling box
box outp
in ( drink :: drinks, return :: _float )
out (d :: int 2, r :: _float)
match
  ( d, * ) -> (showdrink d, *)
| ( *, r ) -> (* , r)
;

stream stdout to "std_out";
stream stderr to "std_err";
stream stdin  from "std_in";

wire inp    ( stdin )
            ( coffee.coin, coffee.button );
wire coffee ( inp.coin, inp.button, coffee.value' initially 0.0 )
            ( outp.drink,  coffee.value, outp.return );
```

```
wire outp   ( coffee.drink, coffee.return )
            ( stdout, stderr );
```

Below we show only the interesting part of the generated intermediate code:

```
program

type wire1float = W1float {-1-} float| NOVAL1float {-0-}
type wire1int = W1int {-1-} int| NOVAL1int {-0-}
type wire1drinks = W1drinks {-1-} drinks | NOVAL1drinks {-0-}
type wire1buttons = W1buttons {-1-} buttons | NOVAL1buttons {-0-}
type wire1coins = W1coins {-1-} coins | NOVAL1coins {-0-}
type wire1char = W1char {-1-} char| NOVAL1char {-0-}
...
type bundle2coinsbuttons = B2coinsbuttons {-2-} wire1coins  wire1buttons
type bundle3drinksfloatfloat = B3drinksfloatfloat {-3-} wire1drinks  wire1float  wire1float
...
type tuple2coinsbuttons = T2coinsbuttons {-2-} coins   buttons
type tuple3drinksfloatfloat = T3drinksfloatfloat {-3-} drinks   float  float
...

type coins = Nickel {-0-}| Dime {-0-}
type drinks = Coffee {-0-}| Tea {-0-}
type buttons = BCoffee {-0-}| BTea {-0-}| BCancel {-0-}

-- Functions
{vend :: drinks,float,float, ->bundle3drinksfloatfloat (drink :: drinks)  (cost :: float)  (v :: float) =
 glet ?z_21    = v>=cost
 in if ?z_21
      then glet ?z_22    = v-.cost
           in glet ?q_23    = NOVAL1float
              in glet ?w_58    = W1drinks drink
                 in glet ?w_57    = W1float ?z_22
                    in B3drinksfloatfloat ?w_58  ?w_57  ?q_23
      else glet ?q_24    = NOVAL1drinks
           in glet ?q_25    = NOVAL1float
              in glet ?w_56    = W1float v
                 in B3drinksfloatfloat ?q_24  ?w_56  ?q_25;
... }

-- Boxes
val ?arg_00 :: bundle1char
box inp
in  (?v_6 :: wire1char) -- bundle1char
out (?v_2 :: wire1coins)  (?v_1 :: wire1buttons) -- bundle2coinsbuttons
match
 case ?arg_00 of
   (B1char (W1char ('N'))) -> glet ?z_9    = (Nickel)
                              in glet ?q_10    = NOVAL1buttons
                                 in glet ?zz_0    = W1coins ?z_9
                                    in B2coinsbuttons ?zz_0  ?q_10|
   (B1char (W1char ('D'))) -> glet ?z_11    = (Dime)
                              in glet ?q_12    = NOVAL1buttons
                                 in glet ?zz_0    = W1coins ?z_11
                                    in B2coinsbuttons ?zz_0  ?q_12|
   (B1char (W1char ('C'))) -> glet ?q_13    = NOVAL1coins
                              in glet ?z_14    = (BCoffee)
                                 in glet ?zz_1    = W1buttons ?z_14
                                    in B2coinsbuttons ?q_13  ?zz_1|
   (B1char (W1char ('T'))) -> glet ?q_15    = NOVAL1coins
                              in glet ?z_16    = (BTea)
```

```
                                in glet ?zz_1    = W1buttons ?z_16
                                   in B2coinsbuttons ?q_15  ?zz_1|
   (B1char (W1char ('X'))) -> glet ?q_17    = NOVAL1coins
                              in glet ?z_18    = (BCancel)
                                 in glet ?zz_1    = W1buttons ?z_18
                                    in B2coinsbuttons ?q_17  ?zz_1|
   (B1char ?wild_55) -> glet ?q_19    = NOVAL1coins
                        in glet ?q_20    = NOVAL1buttons
                           in B2coinsbuttons ?q_19  ?q_20
 esac
;
...
```

Note that in the intermediate code we do not have special data structures for tuples, vectors etc. All these data structures are mapped into user-defined data structures, which are automatically inserted at the begin of the program. To deal with values passed along wires between boxes, we automatically generate 2 forms of datatypes in the intermediate code. A unary *wire datatype*, with a constructor of the form `W1<type>` to represent the wire itself, and nullary constructor `NOVAL1<type>` representing the absence of input. A *bundle datatype* collects all input- or output-wires of a box into one datatype. Both forms need to be monomorphic in the current system and are therefore instantiated for all (combinations of) types needed in the particular program. These data-structures are handled specially by the costing module of the inference to avoid over-estimates due to the inserted constructors. With these in-place, we can use the same inference machinery for boxes as is used for functions, in particular we can attach potentials to the constructors of wire data-types. Also note, that in this intermediate code the names of the wires between boxes match up, thus implicitly representing the connections between the boxes. For the above example the inferred costs are:

```
ARTHUR3 typing for resource "Time":

Box: inp
    ?v_6: wire1char[1809;char|*]
   ---2603/0--->
    ?v_2: wire1coins[0;coins[0|0]|*], ?v_1: wire1buttons[1741;buttons[0|402|0]|*]

Box: coffee
    ?v_2: wire1coins[0;coins[0|0]|*], ?v_1: wire1buttons[1741;buttons[0|402|0]|*],
    ?v_5: wire1float[0;float<0>|*]
   ---4790/0--->
    ?v_3: wire1drinks[1104;drinks[0|207]|*], ?v_5: wire1float[0;float<0>|*],
    ?v_4: wire1float[626;float<0>|*]

Box: outp
    ?v_3: wire1drinks[1104;drinks[0|207]|*], ?v_4: wire1float[626;float<0>|*]
   ---1180/0--->
    ?v_7: wire1int[687;int|*], ?v_8: wire1float[0;float<0>|*]
```

In contrast to the situations for heap and stack, the time consumption is not constant over the inputs (but it is independent of the size of the float value, as shown by the 0 potential in `float<0>`). On the wire `?v_1` we see a cost of 402 cycles attached to the `BTea` case, whose processing expands to a call of the function `vend`.

8.5.5 Example: core of Canny edge detection

As final example we take the computational core of the Canny edge detection algorithm [20], implemented by the LASMEA partner group (see also EmBounded Deliverable 07 [134] on real-time testbed

applications). The full algorithm detects edges in a 2-dimensional image, by comparing the contrast values of neighbouring points in a window that is traversing the image. The core of this algorithm is a function `mapapply job limit next sr si img` of type
`(img->img) -> (img->img) -> (img->img) -> float -> float -> img -> img`.
Here `job` describes the function that should be applied to each window, `limit` selects the window based on the current position, `next` moves to the next position, `sr` and `si` are boundary values to prevent the window from overruning the end of the image, and `img` is the image to process. This function is instantiated several times in the Canny code with convolution operations using the Gaussian filters as worker functions. The original Hume code for this function is:

```
mapapply job limit next sr si img =
  if (sr<=si)
  then (job (limit img)):(mapapply job limit next sr (si-1) (next img))
  else [];
```

Several versions have been implemented, based on lists and vectors. As test case for the analysis we chose a list-based version. We instantiate the function parameters with concrete functions to get a first order version of the code (it should be noted, however, that the analysis itself can deal with higher-order functions, but due to limitations of the translation into intermediate code, this hasn't been used so far). Also, to simplify the program we use a list of floats rather than a list of lists of floats, i.e. we consider only a 1-dimensional image. The program we are analysing is this:

```
type _float = float 32;
data flist = NNil | CCons _float flist;

hd_flist :: flist -> _float;
hd_flist l =
   case l of
     (CCons h t) -> h;

tl_flist :: flist -> flist;
tl_flist l =
   case l of
     (CCons h t) -> t;

inc :: _float -> _float;
inc x = x + 1.0;

max3 :: flist -> _float;
max3 xs = case xs of (CCons x1 xs1) ->
          case xs1 of (CCons x2 xs2) ->
          case xs2 of (CCons x3 xs3) ->
           if (x1<x2)
            then if (x2<x3) then x3 else x2
            else if (x1<x3) then x3 else x1;


take3 :: flist -> flist;
take3 xs = case xs of (CCons x1 xs1) ->
           case xs1 of (CCons x2 xs2) ->
           case xs2 of (CCons x3 xs3) ->
           (CCons x1 (CCons x2 (CCons x3 NNil)));

next :: flist -> flist;
```

```
next xs = tl_flist xs;

mapit :: _float -> _float -> flist -> flist;
mapit sr si l =
  if (sr<=si)
   then CCons (max3 (take3 l)) (mapit sr (si - 1.0) (next l))
   else NNil;

expression (mapit);
```

Note that this program has a significantly more complex computational structure compared to the vending machine in the previous section. By performing only one pass over the data structure we still can hope to derive an upper bound using our approach of linear programming for solving the generated constraints. Indeed, when we analyse the code we get the following upper bound:

```
ARTHUR3 typing for resource "Time":
  30, (float<0>,float<567>,flist[0|4605;float<0>,#]) -4236/0-> flist[3891|0;float<0>,#] ,0
```

In total the `mapit` function takes $567m + 4605n + 4236$ cycles for an input list of length n and a float value `si` of size m. Again, the inference can capture linear bounds in several input arguments.

Chapter 9

Validation of Analysis Results

Steffen Jost and Kevin Hammond

Abstract

This chapter assesses the quality of the results produced by our prototype implementation of the parameterised worst-case execution time, stack- and heap-space analyses for Schopenhauer. We apply our prototype implementation of the analyses to several example programs, half of them being theoretical and half being practical program examples, and compare the obtained results with measurements of test runs of these programs.

The upper bounds by our fully automatic space analysis tools prove to be highly accurate in almost all cases, especially for stack space. One program example, the RobuCAB messaging subsystem, unfortunately requires some manual intervention to produce acceptable bounds on heap space usage.

The general simplicity of applying our prototype space analysis tools and the good quality of the results produced is very encouraging and shows the maturity of the tools that we have produced to support our analyses.

9.1 The parameterised time, heap- and stack-space analyses

Our analysis is defined formally as a set of type rules for Schopenhauer expressions, in the form:

$$\Sigma; \Gamma \vdash^{t}_{t'} \vdash^{p}_{p'} \vdash^{m}_{m'} e : A \mid \psi$$

where e is a Schopenhauer expression of type A, Σ and Γ define the typing context for the e. If all constraints in ψ are satisfied, then the execution of e will succeed when at least t time, p stack- and m heap-units were available at the start of the execution[1]. Furthermore, t' time, p' stack- and m' heap-units will be unused after e has been evaluated.

In essence, every valuation v that satisfies all constraints within ψ gives rise to a linear cost formula bounding from above the space requirements to evaluate the Schopenhauer expression e.

These type rules (Chapter 7) were encoded to give an automatic time and space analysis for Schopenhauer expressions that takes an expression and contextual information as its inputs, and produces a parameterised typing with its associated constraints. The constraints are then solved by a standard constraint solver, whose solutions allow the generation of the linear cost formula as the output of the analysis.

9.1.1 How the Analyses Works

To recapitulate, our analyses work as follows:

a) The Schopenhauer program to be analysed is first translated into a simplified subset of Schopenhauer, which we refer to as Core-Hume. This pre-processing exposes several compiler decisions and removes any ambiguity for the evaluation order of sub-expressions. This translation also clearly exposes all stack operations from source language constructs, removing any ambiguities in the source language which are traditionally decided by the compiler. This knowledge then enables us to determine rigid bounds on stack usage by our analysis. Note that from a theoretical viewpoint Core-Hume must still be regarded as a high-level language, albeit a de-sugared[2] one, as opposed to a bytecode or machine-level language. The entire process is described in Chapter 7.

   Note that in order to obtain a useful WCET bound, *we also need precise knowledge about the memory configuration,* since each memory operation will also have an associated time cost. Therefore the main contribution of the translation is to expose stack operations from source language constructs.

b) The Core-Hume program is then analysed by our amortised WCET and space analyses. Our amortised analysis is type-based and thus first constructs a standard type derivation. All occurring types are then enriched by inserting resource variables ranging over the non-negative rational numbers, denoting potential. Afterwards all annotated type rules, are then matched to the unannotated counterparts and the arising constraints among the resource variables are gathered. These gathered constraints form a linear constraint program, containing parameters for differing actual costs (e.g. heap- or stack-space costs).

c) In order to solve this linear program, the cost parameters must be instantiated with reliable values for the target architecture of interest. These are taken from the Schopenhauer operational semantics (Chapter 6). We have used AbsInt's **aiT** tool to determine WCET bounds for individual HAM instructions on the Renesas M32C/85U microprocessor. Although the **aiT** tool is a powerful and sophisticated tool that requires a trained user, the cost parameters for each HAM instruction only need to be determined once per target architecure.

[1] in addition to the potential of Γ, which is always zero at top level

[2] in the sense that a Human would not want to write Core-Hume by Hand

The analyses for WCET, heap- and stack-space usage are performed independently: We simply rerun this step, exchanging the cost parameter table according to the desired resource, i.e. either the cost parameters for heap-space or stack-space usage.

d) The linear program is then solved using a standard LP-solver. The complexity of the linear programs that we generate is not a problem for current LP-solver technology and can be solved efficiently.

The solution to the linear program then gives rise to an annotated typing of the original program. This annotated typing in turn allows bounds on WCET, stack- and heap-space consumption to be computed as a simple linear formula over the program's principal input sizes. The generation of this human-readable cost formula has been done by hand here, but we have now automated this feature.

Step a) has already been incorporated into the prototype Schopenhauer to HAM compiler, since it is dependent on several compiler decisions. Essentially it branches out an intermediate form of the program along its compilation process, augmented by several annotations to safe redundancies (e.g. the number of variables contained in a pattern match, etc.).

The implementation of Steps b)–d) has been written as a standalone tool in Haskell, a modern, powerful functional language [39, 44] [79]. Thanks to using comprehensive cost parameters that cater for all imaginable occasions, the implementation of Step b) is entirely generic. These cost parameters are instantiated using an appropriate cost table, which can be for time or space, or other metrics, as required. For the remainder of this document we shall be concerned with the cost tables for stack- and heap-space consumption, as described by the operational semantics from Chapter 6. and verified by the cost model (Chapter 5). These actual cost parameters, as used in the remainder of this chapter, represent the space costs for executing Schopenhauer code on the Renesas M32C/85U processor. However, unlike the cost parameters for worst-case execution time, these cost parameters only depend marginally on the target architecture, since space management is largely due to the design of Schopenhauer and its compiler. Therefore the presented results are likely to hold for a vast number of similar architectures.

The final step (Step d), solving and interpreting the linear program, is also automatically performed by our prototype analysis tool. The generated linear programming problem is solved by calling a standard external LP-solver (*lp-solve* [10]), which is available under the GNU Lesser General Public License.

During execution of the analysis, the generic cost parameters, concrete cost parameters for WCET, heap- and stack-space and the constraints that are generated are all saved to files in a human readable format. This gives the user the opportunity to verify and understand the analysis process, if required.

9.1.2 Assessing the quality of the results

Our analysis yields a formally guaranteed upper bound on the worst-case execution time, and the stack- and heap-space usage for the analysed program. However, obtaining a poor upper bound may result in an overly conservative use of resources, and we therefore want to minimise any discrepancy between our predicted bound and the actual WCET and space usage of a program. A perfect analysis would produce bounds which are matched precisely by at least one test run.

However, such perfect results are unlikely to be obtained for the following reasons:

1) In general, it is undecidable whether all computational paths through a program are in fact valid or not. This is a well-known problem and our analysis does not even attempt to solve it. For example, in the expression

`if` FALSE `then` ⟨high-cost⟩ `else` ⟨low-cost⟩

our analysis will not recognize that the then-branch is never executed and will use *high-cost* as the overall cost of the entire expression.

While it might initially appear desirable to recognize such trivial instances in a more sophisticated version of our analysis, simple cases such as this can easily be recognized by a simple optimizing compiler and can thus be eliminated from the input to the analysis. However, recognizing *all* dead computational paths accurately is not possible in general.

2) The cost parameters that are obtained using the **aiT** tool for each HAM instructions are themselves already individual worst-case bounds, with some margin of inaccuracy. It follows that summing these individual costs will lead to some over-estimation of the actual WCET. For example, on the Renesas M32C/85U processor reading or writing data at an odd memory address takes one more clock cycle than reading or writing data at an even address. Since we may not know the actual runtime memory addresses, we conservatively assume that each read/write operation may refer to an odd address. Although this assumption leads to a guaranteed upper bound on WCET, it is highly unlikely that we would encounter a program where every memory access was to an odd address. While this particular issue could be resolved by careful code generation/memory mapping, there are many other, similar examples which are not so easily resolved.

3) Even if we were to obtain the actual least upper bound on execution time, we still might not be able to generate test cases that would expose this cost.

4) Some usage may depend on parametes other than input size. For example, stack-space usage often depends on the depth of the data rather than its overall size.

Fortunately, the analysis is capable of warning us where such an imprecision might occur, and as opposed to worst-case execution time, such instances are rare enough for space usage in order to help us in the generation of actual worst-case test cases for stack- and heap-space usage, thus lessing the impact of this particular problem. The test results in the following pages show that our space analyses deal very well with these problems, and that surprisingly only the last of the described problems is likely to apply.

Our approach is to apply our prototype implementation of our analyses to several testbed programs. We will then measure their worst-case runtime on varying test inputs and use these to validate the analysis results we obtain. The measurements for heap- and stack-space usage of the simple examples in Sections 9.2 and 9.3 were performed by using the cost model described in Chapter 5. Unlike for worst-case execution time, the cost model is exact for heap- and stack-space costs, as shown in our validation of the cost model (EmBounded Deliverable D28 [96]). The high precision of the cost model is unsurprising, since the cost-counting instrumentation has full access to the run-time heap and stack. Using the cost model for obtaining the space measurements therefore obsoletes the need for time consuming tedious test runs using an actual Renesas M32C/85U board. Exploiting the cost model in this way was highly beneficial for the completion of the work described here.

Unfortunately, the examples discussed in Sections 9.5 and 9.6 depend on further hardware. Hence for these the heap- and stack space usage had to measured on the actual Renesas M32C/85U board. This also means that we could not search for a test case that actually exhibited true worst-case space usage behaviour. Instead, we only have the worst-case space usage that occurred during the normal operation – which may or may not included the worst-case, the last of the three problems described above.

Another important advantage over worst-case execution time is the deterministic nature of space consumption. For a specific input a single test run suffices already to determine the overall space usage accurately.

9.2 Example: Simple Photographer

```
type _smallint = int 16;                                    -- Type declarations
type _bool = int 1;
type _pos = (_smallint, _smallint);

data _op = P | L | R | U | D;                               -- Camera instructions

pos_initial :: _pos -> _bool;
pos_initial (xpos, ypos) =
  if xpos==0 && ypos == 0                                   -- Test whether camera is
  then 1                                                    -- in neutral position (1)
  else 0;                                                   -- or not (0).

action :: (_smallint, _smallint, [_op], [_pos]) -> ([_pos], _bool);
action (xpos,ypos,rest,aps) =
  case rest of
    []     -> (aps, pos_ok (xpos, ypos))                  -- (1) end
  | (P:xs) -> step (xpos,  ypos,   xs, ((xpos,ypos):aps)) -- (2) take picture
  | (L:xs) -> step (xpos-1,ypos,   xs, aps)               -- (3) turn left
  | (R:xs) -> step (xpos+1,ypos,   xs, aps)               -- (4) turn right
  | (U:xs) -> step (xpos,  ypos-1, xs, aps)               -- (5) tilt upwards
  | (D:xs) -> step (xpos,  ypos+1, xs, aps);              -- (6) tilt downwards
```

Figure 9.1: Simple Photographer Example

This first example is sufficiently simple to permit in-depth study: a code snippet that processes a list of operations for simulating the capture of photos with an electric pan & tilt camera. The camera can be tilted up and down or turned left and right in in fixed-size increments. The current orientation of the camera is represented by two integers. Furthermore the camera can be instructed to take a picture at the current position. So this simple example code snippet can be imagined to be a tiny part in a more complex program, for example to track a moving object. Similar applications might be found in control settings, such as the controller of a circuit board drilling robot, etc. The commented HUME code is shown in Figure 9.1 and the automatically-produced Core-Hume program that was fed to the analysis is shown in Appendix 9.7.1.

The function `action` takes as its input the current orientation of the camera, a list of instructions and an accumulator of camera orientations (i.e. a protocol of past operations). The datatype of camera instructions consists of 5 elements: `P` for capturing a picture at the current position; `L` and `R` to turn left and right; and `U` and `D` to tilt up and down respectively.

Executing the function `action` processes the given list of instructions in a tail-recursive way, using a 6-way case discrimination on the command issued: if a command to move the camera is issued (`L`, `R`, `U` or `D`), then the orientation is adjusted accordingly; if the command `P` is given (meaning that a photograph should be taken), then the current position is added to the list of positions at which photographs have already been taken (`aps`), and if there are no further actions, then the machine stops, having first checked that that the camera has now returned to its initial position $(0, 0)$.

For example, `action(0,0,[R,R,U,P,L,P,L,P,D],[])` specifies that the camera starts with the initial orientation $(0, 0)$ and should first rotate right twice, then tilt upwards, take a picture, turn left, take a picture, turn left once more, taking a third picture and eventually tilting downwards again.

```
ARTHUR3 typing for HumeHeapBoxed:
  pos_ok:
    (tuple2intint[T2intint<0>:int,int]) -(12/0)-> int

  action:
    (tuple4intintlist1_oplist1tuple2intint[
      T4intintlist1_oplist1tuple2intint<0>:
        int,
        int,
        list1_op[
          C1_op<10>:_op[L<0>|R<0>|P<4>|U<0>|D<0>],#|
          N1_op<20>],
        list1tuple2intint[
          C1tuple2intint<0>:tuple2intint[T2intint<0>:int,int],#|
          N1tuple2intint<0>] ] )
   -(0/0)->
    tuple2list1tuple2intintint[
      T2list1tuple2intintint<0>:
        list1tuple2intint[
          C1tuple2intint<0>:tuple2intint[T2intint<0>:int,int],#|
          N1tuple2intint<0>],
        int]
```

Figure 9.2: Raw output of heap-space analysis for the Photographer example

Hence the call will evaluate to `([(2,-1),(1,-1),(0,-1)],True)`, meaning that the camera took three pictures at the orientations $(2,-1),(1,-1)$ and $(0,-1)$ and returned to its initial position afterwards.

9.2.1 Assessing the quality of the heap-space bound

Running the space analysis with the Renesas M32C/85U cost parameters for heap space usage directly yields the output shown in Figure 9.2. In order to obtain this output, the analysis generated a constraint set consisting of 199 proper constraints over 351 variables. Solving such a small and sparse constraint set is a trivial task for any current LP-solver, which are easily capable of dealing with problems that may be a thousand times larger. The time required to perform the entire analysis for this example on a contemporary Laptop, including the constraint solving, is very small (≤ 0.06s) and dominated by the IO-operations to read the Schopenhauer program from the harddisk and to write the constraint set to the harddisk. We will thus discuss the runtime of the analysis only for the larger program examples in Section 9.5 and Section 9.6.

The output in Figure 9.2 requires a little practice to read, but it can be straightforwardly be transformed into a simple linear cost formula for each function. We first see that each call to the auxiliary function `pos_ok` will consume 12 heap units, regardless of the input provided. However, this result is uninteresting as this cost is also entailed in the cost formula for the main function `action`, which eventually calls function `pos_ok`. The cost formula for function `action` can be more easily read in the following simple form:

$$20 + 14 \cdot \#\mathtt{P} + 10 \cdot \#\mathtt{L} + 10 \cdot \#\mathtt{R} + 10 \cdot \#\mathtt{U} + 10 \cdot \#\mathtt{D} \tag{9.2.1}$$

where #P denotes the number of P-instructions, #L denotes the number of L-instructions, etc., which are contained in the input of the call to main function `action`.

Note again that the above formula represents the worst-case heap space consumption incurred by a completed call to function `action`. This includes the cost of all subsequent calls, both recursive calls to itself as well as calls to other function, in this case only `pos_ok`, which by itself consumes 12 heap units per call. It does not entail the heap space occupied by the input. The analysis can be instructed to cost the setup of a certain input as well, in which case the result is a single constant number. In other words, *analysing a function like* `action` *yields an implicit cost formula, while analysing an expression like* `action(0,0,[L,P,R],[])` *automatically applies that cost formula to the provided input and returns a single number.* This single number then also includes the cost of generating the provided input in the first place. For example, the input `action(0,0,[L,P,R],[])` occupies 32 heap units: 2 heap units for each (boxed) integer constant, 20 heap units for the command list `[L,P,R]`, 2 heap unit for the (boxed) empty list of integer pairs, and 6 heap units for the quadruple itself that stores the pointers to these four objects. In general each n-tuple requires $2 + n$ heap units for storage, since all values are boxed in the underlying memory model chosen for Schopenhauer. A list of length n requires $2 + 4n$ heap units for storage, plus the heap required to store the boxed objects, as the list itself only holds the pointers. Since a command instruction occupies 2 heap units, a command list of length n requires $2 + 6n$ heap units in total, thus 20 for a command list of length 3.

In order to assess the quality of out space analyses, we have *applied the analysis once* to obtain the general cost formula as shown above (9.2.1). We then *executed the Schopenhauer code several times* with different concrete inputs and measured the actual consumption. Note that unlike for worst-case execution time, running the program for a specific input once is enough to determine both the heap- and stack-space usage exactly. From the measured figure, we deducted the cost for generating the input and compared that value to the bound derived from the general cost formula for the cost incurred by the call to the function. For added safety and to ensure that the we deducted the corrected amount, we also ran the analysis once more for the complete call including the specific input.

Function Input		Heap Usage		Analysis Bound		Ratios	
Actual	Heap Size	Overall	Call	Overall	Call	Overall	Call
`[]`	14	34	20	34	20	1.0	1.0
`[P]`	20	54	34	54	34	1.0	1.0
`[D]`	20	50	30	50	30	1.0	1.0
`[D,P]`	26	70	44	70	44	1.0	1.0
`[D,P,U]`	32	86	54	86	54	1.0	1.0
`[P,P,P,P,P]`	44	134	90	134	90	1.0	1.0
`[U,U,U,U,U]`	44	114	70	114	70	1.0	1.0
5P 3L 3R 2U 2D	104	294	190	294	190	1.0	1.0
72P 176{L,R,U,P}	1502	4290	2788	4290	2788	1.0	1.0
72P 178{L,R,U,P}	1514	4322	2808	4322	2808	1.0	1.0

Table 9.1: Heap-space analysis results for Photographer example

The result is shown in Table 9.1: the first two columns shows the specific input together with the number of heap units required to store that input (the last three entries were shortened by only listing the number of occurrences of each command – the actual order showed to be irrelevant anyway); the next column, "Heap Usage Overall" shows the measured consumption, immediately followed by "Heap Usage Call" which is simply the overall amount minus the input size; Column "Analysis Bound Overall" shows the bound obtained by the individual analysis run, including the cost of generating the input; Column "Analysis Bound Call" shows the bound as calculated from the general formula obtained by the initial single run of the analysis, which did not know which input would be applied, so this is the important column; and the last two columns show the ratios for convenience.

```
ARTHUR3 typing for HumeStackBoxed:
  pos_ok:
    (tuple2intint[T2intint<0>:int,int]) -(9/9)-> int

  action:
    (tuple4intintlist1_oplist1tuple2intint[
      T4intintlist1_oplist1tuple2intint<0>:
        int,
        int,
        list1_op[
          C1_op<0>:_op[L<0>|R<0>|P<0>|U<0>|D<0>],#|
          N1_op<5>],
        list1tuple2intint[
          C1tuple2intint<0>:tuple2intint[T2intint<0>:int,int],#|
          N1tuple2intint<0>] ] )
   -(19/23)->
    tuple2list1tuple2intintint[
      T2list1tuple2intintint<0>:
        list1tuple2intint[
          C1tuple2intint<0>:tuple2intint[T2intint<0>:int,int],#|
          N1tuple2intint<0>],
        int]
```

Figure 9.3: Raw output of stack-space analysis for the Photographer example

The result shows that the heap space consumption for the Photographer program example can be predicted *exactly* in *all* cases. The reason for this perfect result is that each possible branch of computation is associated with an individual element of the input. This allows for the existence of an exact linear formula to describe the overall heap space usage in the same form as the formulas produced by our analysis tool. The analysis thus demonstrates that is indeed capable of finding this formula.

9.2.2 Assessing the quality of the stack-space bound

Repeating the analysis with the stack space cost parameters for the Renesas M32C/85U yields the output shown in Figure 9.3, which required solving 223 proper constraints over 399 variables, an excerpt of which is shown in Appendix 9.7.2, followed by an excerpt of one of the debug trace outputs generated during the analysis process.

The stack-space cost formula for the main function `action` reduces to the single constant 24, i.e. the stack space usage does not depend on the given input in any way according to our analysis.

The overall maximum stack size measured is 31, which turns out to be indeed independent of the input as predicted. Again, we have to deduct the stack space required for the initial setup and the generation of the input. It turns out that a maximum of 10 stack units is required to generated an input having a command list of arbitrary length. Of these 10 stack units, three are reclaimed by generating the quadruple (whish pop 4 pointer and pushes one) immediately before the call, thus revealing that the maximum stack space usage of the call is indeed 24 as predicted by our analysis.

Similarly to the previous section assessing heap-space usage, we applied the analysis also to the full program expression containing the input generation. Again, it accurately predicts a maximum stack usage of 31 units in accordance to the measurements.

For consistency we repeat the cost/measurement-table for stack space usage in Table 9.2, but omit

Function Input		Maximum Stack		Analysis Bound		Ratios	
Actual	Max Stack	Overall	Call	Overall	Call	Overall	Call
[]	10	31	24	31	24	1.0	1.0
[P]	10	31	24	31	24	1.0	1.0
[D]	10	31	24	31	24	1.0	1.0
[D,P]	10	31	24	31	24	1.0	1.0
...	⋮	⋮	⋮	⋮	⋮	⋮	⋮
72P 176{L,R,U,P}	10	31	24	31	24	1.0	1.0
72P 178{L,R,U,P}	10	31	24	31	24	1.0	1.0

Table 9.2: Stack-space analysis results for Photographer example

```
ARTHUR3 typing for resource "TimeM32":

pos_ok: (tuple2intint[T2intint:0;int<0>,int<0>]) -(1288/0)-> int<3>

action:
  (tuple4intintlist1_oplist1tuple2intint[
     T4intintlist1_oplist1tuple2intint<0>:
      int<0>,
      int<0>,
      list1_op[
        C1_op<2013>:_op[P<27>|L<0>|R<263>|U<526>|D<789>],#
        |N1_op<790>],
      list1tuple2intint[
        C1tuple2intint<0>: tuple2intint[T2intint<0>:int<0>,int<0>],#
        |N1tuple2intint<0>]])
 -(2115/0)->
   tuple2list1tuple2intintint[
    T2list1tuple2intintint<0>: list1tuple2intint[
      C1tuple2intint<0>:tuple2intint[T2intint<0>:int<0>,int<0>],#
      |N1tuple2intint<0>],
     int<3>]
```

Figure 9.4: WCET analysis for the Photographer example

several trivially repeated rows. Again, the analysis accurately predicts the maximum stack space usage of the program in all cases.

9.2.3 Assessing the quality of the WCET results

The automatically-produced Core-Hume program that was input to the analysis is shown in Appendix 9.7.1. Running the WCET analysis with the Renesas M32C/85U cost parameters yields the output shown in figure 9.4. In order to enhance generalities, these figures are given as multiples of the basic clock cycle used by the processor. Since we are using a 32MHz version of the M32C/85U, absolute execution times in seconds can be obtained by multiplying the number of clock cycles by $3.125 \cdot 10^{-5}$.

The constraint set contains 224 constraints over 409 different variables. Solving this is a trivial

task for any current LP-solver, which are easily capable of dealing with problems that may be a thousand times larger. The annotated type printout that is returned from the analysis presently requires some practice to decipher. The lengthy function names were introduced by the automated translation from Hume to Core-Hume, and include full type information, as in the JVM. Every function arrow is annotated with two values: the first value denotes the maximum number of clock cycles that are required when computing the function, the second value denotes at least how many of these remain unused once the computation is finished. These values correspond to the potentials that are manipulated by our formal definition of the analysis. For example, the annotation `-(x/y)->` means that the function requires at most $(x - y)$ clock cycles to compute. The second value is mainly needed for function composition, and is usually zero for the WCET analysis. We will see a use for the second annotation in the next example (Section 9.3).

We can see from the output that a call to the function `pos_ok` takes at most 1288 cycles (the first value annotation on the arrow of the function's type).[3] We do not need to worry about the cost of `pos_ok`, since this function is only called by the main function `action`, whose WCET already accounts for *all* subsequent calls to other functions, including tail calls to itself. However, the numbers on the arrow only denote the input-independent part of the WCET bound. The *total* WCET for a function also contains an input-dependent part (the values enclosed in angle brackets in Figure 9.4). These values represent the weight that each individual argument has with respect to the overall WCET for that function. These weights can be straightforwardly translated into the simple linear formula

$$2905 + 2040 \cdot \#\texttt{P} + 2013 \cdot \#\texttt{L} + 2276 \cdot \#\texttt{R} + 2539 \cdot \#\texttt{U} + 2802 \cdot \#\texttt{D}$$

where #`P` denotes the number of `P`-instructions, #`L` denotes the number of `L`-instructions, etc., which are contained in the input of the call to `action`. The formula is computed by simple addition from the annotations given in figure 9.4: We see there that processing each element of the instruction list costs at most 2013 clock cycles, plus a certain amount depending of the actual instruction. For example, processing an `R` instruction requries an additional 263 cycles, yielding a total of 2276 as shown in the formula. While it might be expected that the final four cases (`L`, `R`, `U` and `D`) would all give identical results, since their code is identical, the general pattern-matching code requires earlier patterns to have been tested, and failed to match before later ones are tried. Reordering the rules would therefore change the cost assigned to each instruction. The fixed value of 2905 clock cycles for the `action` function as a whole, consists of the 2115 cycles from the arrow annotation plus a weight of 790 for the `[]` that terminates each sequence of actions.

For example, the call `action(0,0,[R,R,U,P,L,P,L,P,D],[])` will complete in at most $22944 = 2666 + (3 \cdot 2040) + (2 \cdot 2013) + (2 \cdot 2276) + (1 \cdot 2539) + (1 \cdot 2802)$ clock cycles. We have verified that the automatic WCET analysis for this code snippet corresponds to our formal analysis by performing the same analysis by hand. We will now investigate how our theoretical WCET bound relates to actual measured worst-case execution times. More details on how we actually derive our cost values in given in Sections 9.7.2 and 9.7.4.

[3] The annotation `int<3>` also shows that the call needs three cycles less if `True` is returned: the else-branch requires three additional clock cycles to allow for the unconditional branch around the then-branch.

9.3 Example: List Folding

```
-- Program: sum over list-of-floats
-- Variant: using high-order function fold

type num = float 32;
--type num = int 32;

add :: num -> num -> num;
add x y = x + y;

fold :: (num -> num -> num) -> num -> [num] -> num;
fold f n []     = n;
fold f n (x:xs) = fold f (f x n) xs;

sum :: [num] -> num;
sum xs = fold add 0 xs;
```

Figure 9.5: Summing lists using the higher-order fold function

Our second simple example is the standard folding function over lists (also known as the `reduce` function): `fold f n xs` applies function `f` across every element in `xs`, plus an accumulated result for the rest of the list, starting with the value `n`. For example, we can define the function `sum` that sums lists of numbers as `fold add 0 xs`, or the function `product` as `fold times 1 xs`. This function is interesting in determining that our analysis works for higher-order definitions (`fold` is parameterised on function `f`) as well as first-order definitions. The corresponding Schopenhauer program is shown in Figure 9.5.

We performed the analysis twice, once for summing integers and once for floating-point numbers. However, the results were identical, since a float occupies the same amount of heap space as an integer in our current memory management system. Even if the sizes would differ, stack space usage would remain unaffected, since all values are boxed and the size of a pointer does not depend on the object it is pointing to.

Furthermore there is no effect on the actual numbers used, hence for the remainder of this section, we will distinguish the input only by its length, i.e. the length of the list of numbers that is to be summed up.

9.3.1 Assessing the quality of the heap-space bound

```
ARTHUR3 typing for HumeHeapBoxed:

  add:  (int,int) -(2/0)-> int

  fold: ((int,int) -(0/0)-> int, int, ?list1int[?C1int<0>:int,#|?N1int<0>])
        -(0/0)-> int

  sum:  (?list1int[?C1int<2>:int,#|?N1int<0>]) -(6/0)-> int
```

Figure 9.6: Raw output of heap-space analysis for the list-folding example

The output of applying our analysis for heap space usage to the example provided the output shown in Figure 9.6. The constraint set generated contained about 66 constraints over 135 variables. For a

number list of length n we thus obtain the simple heap space cost formula

$$6 + 2n \tag{9.3.1}$$

for running the function `sum`. It is noteworthy that the individual analysis of the `fold` function shows an argument function having zero heap usage costs, while function `add`, which will fill the argument role, shows a cost of two. There are always several annotations admissible by the constraint sets produced by the analysis. The reported results always try to show numbers as low as possible. Plugging functions together, like applying `add` to `fold` will restrict the number of possible choices for the annotated type of `fold`. Inndeed, we obtain

```
fold add : (int,?list1int[?C1int<2>:int,#|?N1int<0>]) -(0/0)-> int
```

for this as expected. However, it is important to note that each function is analysed only *once*! Each individual use of a function simply produces another copy of the constraints that had been generated for that function, allowing a differing annotated type.

For evaluating the measurement, we have to deduct the cost of creating and storing the input in the first place again. This cost is $2 + 6n$ for a list of number of length n, consisting of 2 heap units for the single list end element, 4 heap cells for each list node that holds two pointers and another two heap cells for the actual number stored.

Also as for the previous example, we again applied the analysis to the full expression as well, i.e. including the cost of the creation of the input in the analysed code.

Function Input		Heap Usage		Analysis Bound		Ratios	
Length	Heap Size	Overall	Call	Overall	Call	Overall	Call
0	2	8	6	8	6	1.0	1.0
1	8	16	8	16	8	1.0	1.0
2	14	24	10	24	10	1.0	1.0
3	20	32	12	32	12	1.0	1.0
4	26	40	14	40	14	1.0	1.0
5	32	48	16	48	16	1.0	1.0
50	302	408	106	408	106	1.0	1.0

Table 9.3: Heap-space analysis results for list folding example

The results are shown in Table 9.3 and again the heap space analysis is flawless, despite the use of high-order functions, which make use of resource parametric resource by necessity.

9.3.2 Assessing the quality of the stack-space bound

Similarly applying our analysis for stack space usage produces the annotated typing as shown in Figure 9.7. The constraint set generated contained about 76 constraints over 79 variables. The number of variables is much lower as for heap space usage, since this time we choose to omit the introduction of slack variables. This is a fine-tuning that happens to provide a slightly better annotated typing in this instance.

The constraint set usually allows many different annotated typings, each describing an admissible cost formula. Solely for the purpose of conveying this information to a human, the analysis employs an heuristic which tries to find the "best" annotated typing. We are forced to use an heuristic here, since examples can be constructed were "best" depends on additional knowledge – mostly our intentions on how to use a program.

```
ARTHUR3 typing for HumeStackBoxed:

  add: (int,int) -(8/9)-> int

  fold: ((int,int) -(0/ANY)-> int,int,?list1int[?C1int<0>:int,#|?N1int<0>])
        -(14/16)-> int

  sum: (?list1int[?C1int<0>:int,#|?N1int<0>]) -(27/27)-> int
```

Figure 9.7: Raw output of stack-space analysis for the list-folding example

This heuristic also shows in the individual typing of function `fold`, containing the value `ANY`, which stands for any arbitrary positive number. The meaning that that `fold` has the best resource usage if it is given a function as an argument, that requires no additional stack space to run, but is miraculously capable of freeing an arbitrary amount of stack space upon completion. Clearly, such a function cannot exist. However, this does not imply that there is no admissible cost formula if we provide a function like `add`, which requires 8 additional stack units to run and frees up 9 afterwards (by popping its arguments from the stack, `add` is capable of lowering the current stack-space usage in effect by one). In fact, such an annotated typing does exist, since otherwise there would be no annotated typing for `sum`.

Note however that influencing the heuristic which simply choose *an* annotated typing among many admissible ones is a non-essential tweak. As we have seen in the previous subsection, plugging functions together will usually reduce the set of admissible annotated typings already enough for the heuristic to choose the "best" solution among all possible annotated typings. For example, for an expression which evaluates to a base value, such as `sum[1,2,3]`, the cost formula is just a constant. Clearly, the best constant cost formula for the upper bound on resource usage is the *least* upper bound, i.e. the smallest constant.

According to the result in Figure 9.7 above, summing a list of numbers requires at most 27 stack units, regardless of the length of the list to be summed up. So in this case the result is a constant cost formula.

Again for evaluating the measurement, we have to deduct the cost of creating and storing the input in the first place again. The basic setup require 6 stack space units, plus 1 if the input is just an empty list and plus 2 if the input is a non-empty list of arbitrary length. However, in the latter case one stack space might be reclaimed immediately before the call, therefore we always may deduct 7 stack space units from the overall figure for the cost which is related to the call only.

Also as for the previous example, we again applied the analysis to the full expression as well, i.e. including the cost of the creation of the input in the analysed code.

Function Input		Maximum Stack		Analysis Bound		Ratios	
Length	Max Stack	Overall	Call	Overall	Call	Overall	Call
0	7	24	17	24	27	1.00	1.59
1	8	34	27	34	27	1.00	1.00
2	8	34	27	34	27	1.00	1.00
3	8	34	27	34	27	1.00	1.00
4	8	34	27	34	27	1.00	1.00
5	8	34	27	34	27	1.00	1.00
50	8	34	27	34	27	1.00	1.00

Table 9.4: Stack-space analysis results for list folding example

```
add  : (int,int) -(565/0)-> int
fold : ((int,int) -(832/169)-> int,int,list1int[
          C1int<1620>:int,#|N1int<76>])
       -(563/169)-> int
sum  : (list1int[C1int<1620>:int,#|N1int<76>]) -(987/0)-> int
```

Figure 9.8: WCET analysis results for summing integers

The comparison of the stack space analysis with the actual maximum stack usage is shown in Table 9.4. The analysis is exact, if it is also provided with the input for the computation. The generic cost formula obtained by a single run of the analysis, which had no knowledge about the input, is exact in all cases, except for the empty input list, in which case it safely overestimates stack usage by less than 60%.

This overestimation is due to the heuristic choosing the generic formula. Running the analysis with default option of inserting slack variables that are to be minimised in all inequalities yields the cost formula $17 + 11n$, where n is the length of the input list. This formula is exact for the empty list, but otherwise rather bad. The problem is that our heuristic for choosing a solution among all admissible one treats both cases as equal, so at one time we eliminate the imprecision for the case where the empty list is received and at the other we eliminate the imprecision for the case where the non-empty list is received. As a result of this investigation, the option of not introducing slack variables has become the default for stack-space analysis.

9.3.3 Assessing the quality of the WCET results for the List folding examples

We performed the analysis twice, once for integers (Figure 9.8) and once for floating-point numbers. (Figure 9.9). We see that summing a list of n integers on the Renesas M32C/85U processor takes at most $1063 + n \cdot 1620$ clock cycles. Summing a list of n floats similarly requires at most $1071 + n \cdot 2610$ clock cycles. The constraint sets generated contained about 60 constraints over 125 variables.

It is interesting to observe that there is a difference of 267 clock cycles between using `add` on its own and as an argument of `fold`, regardless of the type of number that is being summed. This is because `add` can only be given as an argument to `fold` once it is turned into a function closure, and this has some cost (267 clock cycles in this instance).

Table 9.5 gives WCETs for both integer and floating-point summations on the M32C/85U, comparing the measured WCET against that predicted by the analysis. For integers, the measured WCETs conform precisely to a time formula of $995 + 1250 * \#cycles$. The analysis predicts costs that are within 30% of these figures. In fact, we observe that our time series for the analysis will converge to give a (safe) over-prediction of at most 29.6% compared with the measured WCETs that we expect.

Since floating-point arithmetic on the Renesas processor is performed using a software library rather than by hardware instructions, we are not able to obtain such a precise result for floating-point summation. Here,our time series converges to give an over-predicton of at most 88.2% compared with the extrapolated formula for measured WCETs. This difference is revealed by the low level timing

```
add  : (float,float) -(1555/0)-> float
fold : ((float,float) -(1822/169)-> float,float,list1float[
          C1float<2610>:float,#|N1float<76>])
       -(563/169)-> float
sum  : (list1float[C1float<2610>:float,#|N1float<76>]) -(995/0)-> float
```

Figure 9.9: WCET analysis results for summing floating point numbers

list-	integer			floating point		
size	measured	analysis	ratio	measured	analysis	ratio
0	995	1063	1.07	1008	1071	1.06
1	2245	2683	1.20	2305	3681	1.60
2	3495	4303	1.23	3696	6291	1.70
3	4745	5923	1.25	5083	8901	1.75
4	5995	7543	1.26	6477	11511	1.78
5	7245	9163	1.26	7874	14121	1.79
6	8495	10783	1.27	9271	16731	1.80
7	9745	12403	1.27	10668	19341	1.81
8	10995	14023	1.28	12072	21951	1.82
9	12245	15643	1.28	13479	24561	1.82
∞			≤ 1.296			< 1.88

Table 9.5: Measured WCET results for summation on the M32C/85U

information that we have obtained. For example, the **aiT** tool indicates that floating-point additions can take up to 1106 clock cycles, whereas the worst case that we measured was 267 clock cycles. For floating-point multiplication, there is much less discrepancy: the **aiT** yields a predicted bound of 356 clock cycles versus a measured worst case of 242 clock cycles, i.e. an excess of 47.1%. Although our measured WCETs for floating-point addition are all within 1.2%, for a number of arbitrarily-chosen floating-point arguments, any worst-case analysis must account for a number of rarely-encountered execution paths in the library code. It entirely possible that these paths were not exercised by the test cases that we have chosen to use.

9.4 Example: PID Controller

Our third example is taken from the set of real-time testbeds that we developed in the EmBounded project, the simple 2-degrees-of-freedom PID controller for the classic "ball and beam" experiment. This system consists of a control system for a simulation of the classic "ball and beam" experiment. The simulation is based upon the open-source dynamics engine "OpenODE"[4] simulator with a minimal environment based upon openGL. The source code for this example is given in Appendix 9.8. We have only considered WCET results for this example.

Previous work included a simple control system for a water tank using an initial version of the data socket interface between the OCaml-based Hume interpreter and a simple graphics-based tank simulation. The new simulation is a fully-featured physical simulation of a ball and beam with actuator control and direct measurement of beam angle and ball position. It uses a development of the original socket interface generalized to read structured data from URI-defined data sources. Figure 9.10 shows the graphical output from the simulation.

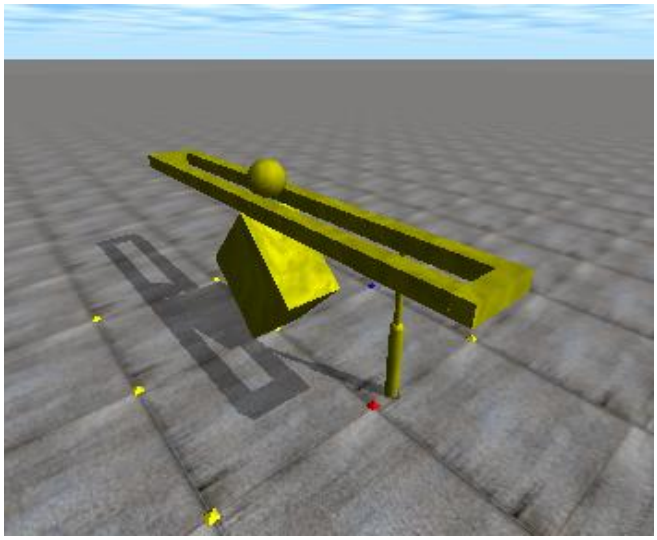


Figure 9.10: Ball and beam simulation with OpenODE and OpenGL

A simple PID controller in Hume Two control systems were written in Hume. The first was a simple 2 degree-of-freedom PID controller, Figure 9.11. Our Hume code is a direct implementation of these recursive equations for a 2-DOF controller with rectangular integration, see Figure 9.12. There are only three basic actions;

1) initialize the controller, detected by an initial `false` for an initialization flag, subsequently set to `true`,

2) reset the simulation if the ball comes off the beam, a command code of `'2'` to the simulation which provides a status flag (`onbeam`) for the ball's on-beam status, and

3) the main control loop which performs the integration using previous instances of loop variables to implement unit delays.

This type of controller is known not to perform well with the ball and beam simulation and is difficult to tune for this application since the ball and beam system is strongly non-linear. For example, the

[4] `http://www.ode.org`

$$
\begin{aligned}
P_k &= K_p * (e_k - e_{k-1}) \\
I_k &= Ki * T * e_k \\
D_k &= (K_d/T) * (e_k - 2 * e_{k-1} + e_{k-2}) \\
CV_k &= CV_{k-1} + P_k + I_k + D_k \\
e_k &= SP_k - PV_k
\end{aligned}
$$

Figure 9.11: Equations for a 2 degree-of-freedom PID controller with rectangular integration

```
match
   (c,(false,ek1,ek2,lc),(vSPk,vCVk1,vPVk,phi,onbeam)) ->
     (('x',vCVk1),*,(true,0.0,0.0,c))
 | (c,(true,ek1,ek2,lc),(vSPk,vCVk1,vPVk,phi,0)) ->
     (('2',vCVk1),(vSPk,vCVk1,vPVk,phi,0,c),(true,0.0,0.0,c))
 | (c,(true,ek1,ek2,lc),(vSPk,vCVk1,vPVk,phi,1)) ->
   let vT = c - lc in
   let ek = vPVk - vSPk in
   let vPk = vKp * (ek - ek1) in
   let vIk = vKi * vT * ek in
   let vDk = ((vKd/vT) * (ek - 2.0*ek1 + ek2)) in
   let vCVk = vCVk1 + limitPID(vPk + vIk + vDk) in
     (('x',vCVk),(vSPk,vCVk1,vPVk,phi,1,vT),(true,ek,ek1,c));
```

Figure 9.12: Hume code for the 2-DOF PID controller

optimal control parameters depend upon the size of the window the simulation is running in. Thus, a second control system was written using the well-known cascaded PID loop controller using the PID controller from the first version as a building block, Figure 9.13. This, in effect works by driving the actuator very hard from the beam angle in the inner loop and using the beam angle to finely control the ball position in the outer loop.

A cascaded PID controller For this application the inputs from and outputs to the plant (the ball and beam simulation) have been separated into a dedicated box (`plant`) whose job is to coordinate the controllers and the simulation. Each of the PID controllers (boxes `pid1` and `pid2`) are identical to the controller in the simple PID application except that the setpoint (`SP`) has been separated from the inputs. The `plant` box measures the sampling rate which is added to the input parameters for each of the controllers. It is then a simple case of wiring the controllers in the cascaded configuration illustrated in Figure 9.13.

This controller works much better but is still prone to diverging if the system alters, for example if the window is resized. The original intention was to develop an adaptive controller for this simulation, which works much better with this type of system and, in fact, this may still be done as a precursor to the control system used on the guided vehicles.

Again, this application has been adapted for the `humec` compiler. Some minor changes to the source were required, for instance since the control loop runs about a hundred times faster the sampling rate for the controller is set by the control loop rather than simply being measured as for the interpreted code. Times measured outwith a predefined sampling interval range are ignored.

Our WCET results are shown in Figure 9.14. The analysis generated 459 constraints over different 793 variables. By adding all input weights, we can see that the PID controller box requires at most 36682 clock cycles per iteration. We can also see that the first branch of the PID controller box can be

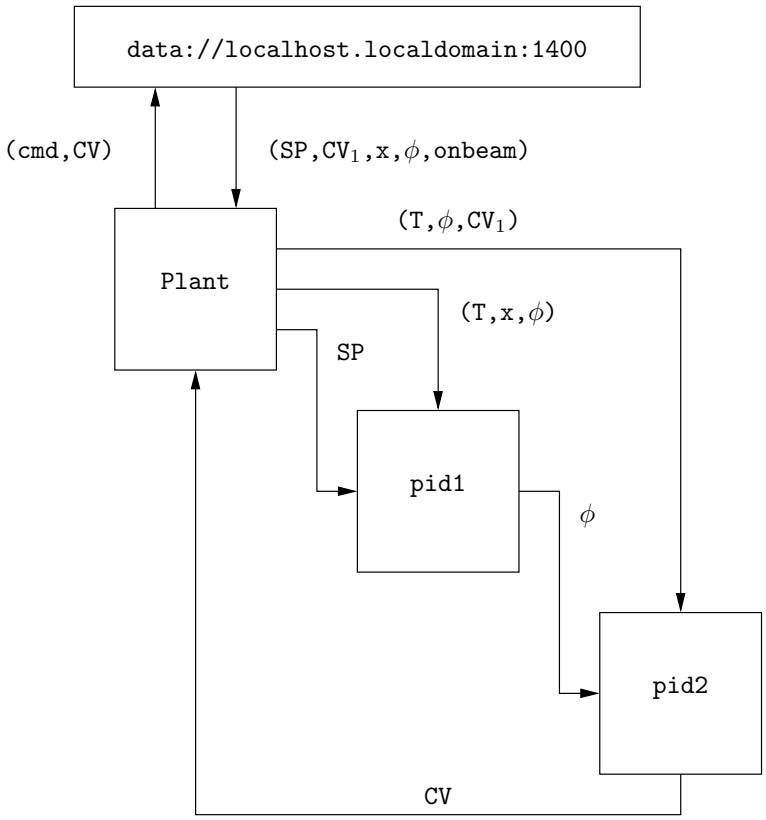


Figure 9.13: Structure of cascaded PID controller

computed much faster (in at most 15081 cycles), since the weight of 21601 attached to the value `True` for the *init-flag* of the `lvs`-wire does not apply for the initial case.

An interesting detail in the analysis result is that we obtain an arbitrary weight (`<ANY>`) for the

```
Box: ctlr
  ctlr.p     : wire1tuple0[W1tuple0<13149>:tuple0[T0<0>]|NOVAL1tuple0],
  ctlr.sp    : wire1int[W1int<0>:int|NOVAL1int],
  ctlr.pv    : wire1int[W1int<0>:int|NOVAL1int],
  ctlr.lvs   : wire1tuple5boolfloatfloatfloatint[
    W1tuple5boolfloatfloatfloatint<0>:tuple5boolfloatfloatfloatint[
      T5boolfloatfloatfloatint<0>:bool[True<21601>|False<0>],float,float,float,int]
    |NOVAL1tuple5boolfloatfloatfloatint]
---1932/0--->
  ctlr.p2    : wire1int[W1int<0>:int|NOVAL1int],
  ctlr.cv    : wire1int[W1int<0>:int|NOVAL1int],
  ctlr.err   : wire1int[W1int<0>:int|NOVAL1int],
  ctlr.lvs'  : wire1tuple5boolfloatfloatfloatint[
    W1tuple5boolfloatfloatfloatint<0>:tuple5boolfloatfloatfloatint[
      T5boolfloatfloatfloatint<0>:bool[True<0>|False<ANY>],float,float,float,int]
    |NOVAL1tuple5boolfloatfloatfloatint]
```

Figure 9.14: WCET analysis results for the PID controller

Branch	Repetitions	Measured execution time			Analysis	Ratio
		Minimum	Average	Maximum		
Rule 1	1	11674	11674	11674	15081	1.2918
Rule 0	999	27473	27742	28406	36682	1.2913

Table 9.6: Measured WCET results for PID-controller on M32C/85U

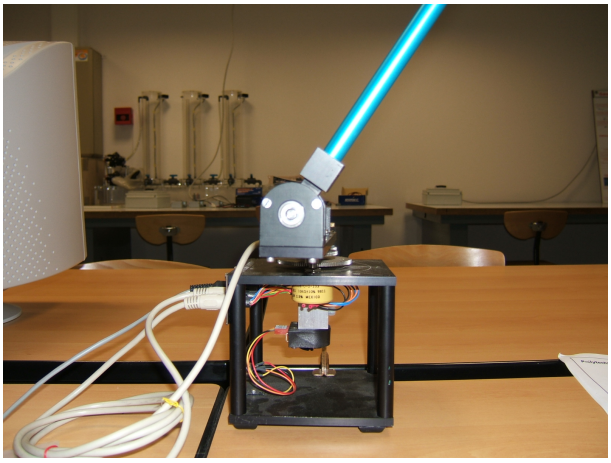
value `False` as the output of the *init-flag* to the `lvs'`-wire. An arbitrary potential, such as this, that occurs in the output of a program must be justified somehow, usually by a corresponding `<ANY>` weight in one of the input values. Since there is no `ANY` weight on any of the input wires, we can deduce that a `False` value is never actually produced. A quick glance at the code in appendix 9.8 easily confirms this, since both branches of the box have the constant `True` at this position. This *init-flag* is only used to signal the start (or resetting) of the PID controller, causing it to initialize itself. Therefore it is always set to `True` if the box has run at least

9.4.1 Assessing the quality of the WCET results for the PID Controller example

We have measured best-case, worst-case and average case times for individual iterations of the controller box on the M32C/85U. The box contains two rules, one for initialisation and one that actually controls the outputs. The results of our measurements are shown in Table 9.6.

The initialisation branch was executed once, giving a measured WCET of 11674 clock cycles against a predicted WCET bound of 15081 clock cycles, a discrepancy of 29.18%. The more interesting control branch was executed 999 times, with an average runtime of 27742 clock cycles and a measured WCET of 28406 clock cycles. The guaranteed WCET bound that we predict is 36682 clock cycles, a discrepancy of 29.13%.

9.5 Example: Inverted Pendulum



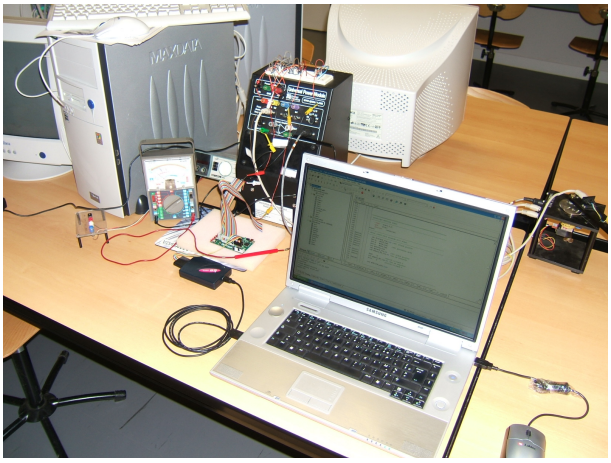


Figure 9.15: Inverted Pendulum controlled by Renesas M32C/85U

Our next example is the inverted pendulum controller from EmBounded Deliverable D7 [135]. This implements a simple, real-time control engineering problem. A pendulum is hinged upright at the end of a rotating arm. The controller receives as input the angles of the pendulum and the rotating arm and should produce as its output the electric potential for the motor that rotates the arm in such a way that the pendulum remains in an upright position. The sensor outputs as well as the input to the motor may range in voltage between -10V and +10V. Since the Renesas M32C/85U only accepts a voltage range between 0V and 5V, three simple operational amplifier (op-amp) voltage scaling circuits were used to interface the pendulum to the development board.

The original Schopenhauer code for the inverted pendulum controller, required some minor modifications to suit it for analysis. We also added some timing code. We were then able to successfully apply our analysis to the code. The revised Schopenhauer code is shown in Appendix 9.9. It comprises about 180 lines of code, which are translated into about eight hundred lines of Core-Hume (not shown here).

9.5.1 Heap space analysis results for the Inverted Pendulum example

The automated analysis for heap space usage of the inverted pendulum controller code generates 499 proper linear constraints over 1010 different resource variables. The overall runtime required for the analysis of these 600 lines of Core-Hume code, including writing the constraints to harddisk and solving them, requires less than (0.22s) on a standard laptop with an 1.8GHz single core Intel Pentium M processor and 1GB of memory.

The output of the analysis is shown in Figure 9.16. This yields the simple constant cost formula that 299 heap units are required per control loop (which are also reclaimed at the end of each control loop, as the box heap is cleared after each run).

The measurements on the actual Renesas M32C/85U reveal a largely differing heap usage for each loop, depending on the inputs received and the necessary actions taken by the control loop. We measured a heap space usage between 143 and 299 units in the worst case.

Therefore, the analysis predicts the worst case exactly. The analysis was unable to give us an input depend cost formula, that would reveal the reduced heap usage in certain cases. This is due to the inputs being numeric values, whose value does correspond linearly to the heap space usage. Thus we assume that a simple cost formula does not exists for the inverted pendulum controller, that would

```
ARTHUR3 typing for HumeHeapBoxed:

Box: regu
  regu.b1:    wire1tuple0[W1tuple0<0>:tuple0[T0<0>]|NOVAL1tuple0]
  regu.b2:    wire1tuple0[W1tuple0<0>:tuple0[T0<0>]|NOVAL1tuple0]
  regu.b3:    wire1tuple0[W1tuple0<0>:tuple0[T0<0>]|NOVAL1tuple0]
  regu.clk:   wire1float[W1float<299>:float|NOVAL1float]
  regu.alpha_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.theta_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.etat:  wire1tuple7boolboolfloatfloatfloatwordword[
    W1tuple7boolboolfloatfloatfloatwordword<0>:
      tuple7boolboolfloatfloatfloatwordword[
        T7boolboolfloatfloatfloatwordword<0>:
          bool[True<0>|False<0>],bool[True<0>|False<0>],float,float,float,int,int]|
        NOVAL1tuple7boolboolfloatfloatfloatwordword]
---0/0--->
  regu.moteur_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.erreur_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.compteur_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.etat': wire1tuple7boolboolfloatfloatfloatwordword[
    W1tuple7boolboolfloatfloatfloatwordword<0>:
      tuple7boolboolfloatfloatfloatwordword[
        T7boolboolfloatfloatfloatwordword<0>:
          bool[True<0>|False<ANY>],bool[True<0>|False<0>],float,float,float,int,int]|
        NOVAL1tuple7boolboolfloatfloatfloatwordword]
```

Figure 9.16: Raw output of heap space analysis for the inverted pendulum example

directly link heap space usage to the sensor voltage. However, this is unimportant from a practical viewpoint, since it is only the worst case heap space usage that needs to be accounted for in practice. The analysis is clearly capable of fulfilling this purpose.

9.5.2 Stack space analysis results for the Inverted Pendulum example

The automated analysis for stack space usage of the inverted pendulum controller code generates 628 proper linear constraints over 651 different resource variables. The reduced number of variables is again due to the choice of not introducing explicit slack variables which are minimised.

The output of the stack space analysis for the inverted pendulum controller is shown in Figure 9.17. This yields the simple constant cost formula that at most 93 stack units are required per control loop pass, which are of course released at the end of each loop pass.

In contrast to heap space usage, our measurements on the actual Renesas M32C/85U board show that the maximum stack space usage varies only a little with the input, alternating between 70 and 93 stack units. Again, the worst case bound predicted by the analysis is accurate.

```
ARTHUR3 typing for HumeStackBoxed:

Box: regu
  regu.b1:    wire1tuple0[W1tuple0<0>:tuple0[T0<0>]|NOVAL1tuple0]
  regu.b2:    wire1tuple0[W1tuple0<0>:tuple0[T0<0>]|NOVAL1tuple0]
  regu.b3:    wire1tuple0[W1tuple0<0>:tuple0[T0<0>]|NOVAL1tuple0]
  regu.clk:   wire1float[W1float<86>:float|NOVAL1float]
  regu.alpha_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.theta_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.etat:  wire1tuple7boolboolfloatfloatfloatwordword[
    W1tuple7boolboolfloatfloatfloatwordword<0>:
      tuple7boolboolfloatfloatfloatwordword[
        T7boolboolfloatfloatfloatwordword<0>:
          bool[True<0>|False<0>],bool[True<0>|False<0>],float,float,float,int,int]|
        NOVAL1tuple7boolboolfloatfloatfloatwordword]
---7/83--->
  regu.moteur_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.erreur_w:
              wire1word[W1word<0>:int|NOVAL1word]
  regu.compteur_w:
              wire1word[W1word<9>:int|NOVAL1word]
  regu.etat': wire1tuple7boolboolfloatfloatfloatwordword[
    W1tuple7boolboolfloatfloatfloatwordword<0>:
      tuple7boolboolfloatfloatfloatwordword[
        T7boolboolfloatfloatfloatwordword<0>:
          bool[True<0>|False<ANY>],bool[True<0>|False<0>],float,float,float,int,int]|
        NOVAL1tuple7boolboolfloatfloatfloatwordword]
```

Figure 9.17: Raw output of stack space analysis for the inverted pendulum example

		Measured execution time				
Branch	Repetitions	Minimum	Average	Maximum	Analysis	Ratio
Match 0	18	3217	3218	3241	3146	0.971
Match 1	18	3749	3749	3749	4390	1.395
Match 2	18	4432	4432	4432	5750	1.297
Match 3	108	2830	2830	2830	6443	2.277
Match 4	58	2837	2837	2837	7171	2.528
Match 5	3804	2669	2669	2669	7702	2.886
Match 6	5976	36118	42222	47635	63678	1.337

Table 9.7: WCET measurements for the inverted pendulum

9.5.3 Assessing the quality of the WCET results for the Inverted Pendulum example

The automated analysis of the inverted pendulum controller code generates 2558 linear constraints (of which 304 were immediately recognized as trivial and discarded) over 4518 different variables. The overall runtime required for the analysis, including solving the generated linear programming problem, took less than 2.7 seconds[5] on a standard laptop with an 1.8Ghz Intel Pentium M processor and 1GB of memory.

We have measured the best-case, worst-case and average clock times required to process a single branch of the controller separately. The box contains six rules: the first four simply handle button presses for starting and stopping the experiment, the fifth represents a halted state, while the sixth represents the actual controlling code. Both our measurements and our analysis predictions are shown in Table 9.5.3.

We see that our analysis gives a high over-approximation for branches 3-5. This happens because our prototype implementation of the analysis over-estimates the time required for processing each pattern match by applying an overly pessimistic (but safe) assumption about match failure. Since the time required to process the actual body of these branches is very small when compared to the cost of performing the pattern match, this over-estimation dominates the costs for these branches. Performing the analysis separately on the body of these branches, *excluding* any pattern matching costs yields a reasonable bound of 2711 clock cycles in each case.

The result for the last branch, the actual controlling loop, is much better, since its overall processing cost is high as compared to the time required for performing the pattern matches. Therefore the over-estimation of pattern match costs becomes negligble, and we observe over-estimation of 33.7% compared with measured WCET. This is consistent with our observations for the previous examples.

Since we have used a chip with a clock frequency of 32Mhz, the measured loop time is 1.488ms, while our predicted loop time would be 1.989ms.

[5]The runtime of the analysis increases to 3.2 seconds if safety assertion checking is enabled.

9.6 Example: RobuCAB electric vehicle

Figure 9.18: RobuCAB electric vehicle

The last example is the messaging subsystem of the RobuCAB control system, described in detail in EmBounded Deliverable D33 [130], which deals with reading messages from the vehicle and generating control messages from instructions that are to be send to the vehicle. The general requirements of this system component are:

1) arrange for the asynchronous status messages from the vehicle to be started and stopped as required;

2) decode the incoming messages and build correctly formatted messages for output;

3) manage the status information being sent asynchronously by the vehicle and store it in a form accessible to the rest of the program;

4) allow outgoing command messages to be inserted into the current message traffic;

5) send a regular (possibly null) message at least every half-second to avoid an automatic safety cut-out;

The messaging subsystem of the RobuCAB, which comprises of two primary boxes, `robucab_in` and `robucab_out`, contains a total of 229 lines of code, which are automatically translated into 1317 lines of Core-Hume. Box `robucab_in` processes the messages sent by the RobuCAB vehicle to the Schopenhauer control system, while `robucab_out` encodes and sends instructions to the vehicle. We will compare the analysis' results with actual measurements gathered from testruns of the Schopenhauer control system on the actual RobuCAB vehicle. EmBounded Deliverable D33 [130] contains more detail on this example.

9.6.1 Heap space analysis results for the RobuCAB messaging subsystem

Performing the automated heap space analyses on the RobuCAB messaging subsystem generates 1608 constraints over 3226 variables for `robucab_in` and 235 constraints over 477 variables for for `robucab_out`. The overall runtime for analysing the heap space usage for both boxes is less than a single second on an average laptop with an 1.8GHz single core Intel Pentium M processor and 1GB of memory.

Heaps space usage for Box `robucab_in`

The analysis reports that the heap space usage of `robucab_in` is non-linear, i.e. it is not possible to construct a cost formula for heap space usage based on the input sizes. However, reading the produced traces immediately shows the culprit: one of the auxiliary functions called in the box `robucab_in` has a non-linear cost. The function recursively calculates a checksum and has a large heap space consumption, due to all numeric values being boxed (and thus requiring heap space) and those are not reclaimed until the box has run to completion. The analysis is in fact capable of analysing this function on its own, using a new experimental feature that assigns potential to numeric values. However, this feature is currently unsafe to use in general and can thus no be applied to the whole program code. A manual inspection showed that it is safe to use on the checksum function itself. It turns out that the heap space usage of the checksum function admits a linear cost formula in terms of the input values (an integer input has a fixed size, but a large range of possible values). The analysis thus yields a cost formula of $48 + 26 \cdot z$ for the heap cell consumption of the checksum function on its own. The parameter z is an actual integer input value for that checksum function. Inspecting the RobuCAB protocols, we learn that this value is restricted to the interval $[1, \ldots, 17]$, hence we can derive a constant heap space usage bound of 490 heap cells for the checksum function.

The analysis for `robucab_in` with the call removed yields a heap usage bound of 64 plus either 18 or 9, depending on the message kind received. The checksum function is only called in branches with require the 9 additional cell, thus resulting in an overall heap space usage bound of $64 + 9 + 490 = 563$. The maximum actual heap space usage recorded during the test runs of the vehicle was 292, hence the analysis overestimated the heap space usage by 92.8%. This relatively large overestimation is likely to be due to z not reaching the value 17 during the test runs. Unfortunately, we are currently unable to test this hypothesis since we have no access to the RobuCAB at the moment and we were not aware of the importance of that value during the original test runs.

Heaps space usage for Box `robucab_out`

The analysis directly yields the cost formula $26 + 13 \cdot n$, where n is the length of a list among the inputs of the box. The list is used as buffer, since we initially found it necessary to buffer the output to prevent overrun at the RobuCAB end of the communications link. In practice, however, we find that only a single buffer place is required to resolve the communications timing problems so we can replace the list with an optional value. Therefore the list is either empty or contains a single element. This yields a maximum heap usage of 39 units. Compared with the measured value of 30 heap units, this yields an acceptable overestimation of 30%.

9.6.2 Stack space analysis results for the RobuCAB messaging subsystem

The automated analysis for maximum stack space usages succeeds directly with no surprises. For `robucab_in`, 1870 constraints over 1799 variables were generated; and 269 constraints over 276 variables for `robucab_out`. The analysis completed within half the time of the heap space analysis, i.e. $< 1.4s$. This reduction in time is mostly due to not introducing slack variables into the constraints, as explained in Section 9.3.2.

The bound on stack space usage for `robucab_in` is 70 and for `robucab_out` is 24. In both cases, this is just one unit above the actually measured stack space usage, 69 and 23 stack units respectively, leading to an overestimation of just 1% and 4%. We have not yet found the source of this small error, and again would like to repeat our test run on the RobuCAB in order to determine what exactly is differing and why. It is still possible that the analysis is actually correct, with the test runs not exhibiting the possible worst-case usage.

9.7 Analysis protocols for the photographer program example

In the following section we will show some excerpts of the various intermediate files produced by the analysis in its course of operation. All excerpts are from the stack-space analysis performed on the photographer program example given in Section 9.2, Figure 9.1. None of these files is intended to be seen (nor understood) by the end user of our automated analysis, i.e. the programmer that programs in Schopenhauer, making use of its guaranteed resource analyses. We provided these excerpts here solely to provide a glimpse on the inner workings of the analysis.

9.7.1 Translation to Core-Hume

This is the result of the automatic translation of the program shown in Figure 9.1 to Core-Hume (Chapter 7), and which is input to the analysis. Line numbers have been added in front of each line to ease readability and allow the tracing of the time cost parameters shown in the following subsections back to this source code.

This excerpt is meant to give a rough impression what the transformed code looks like. All identifiers that have been introduced by the analysis are preceeded by a question mark in order to distinguish them from identifiers specified by the programmer. The question mark symbol was simply chosen because it is not allowed to be at the start of an identifier in Schopenhauer.

```
1 --
2 --
3 -- Running cpp like this: cpp -P photographer15.hume >
    ↳ /tmp/photographer15_19546.hume
4 -- importing /tmp/photographer15_19546.hume
5 -- AST in geek-normal-form  1.5.2.39 20 declarations in total
    ↳ ---------------
6
7 -- 20 declarations in total
8 -- 2 function declarations, split into 2 letrec blocks
9 -- checksum: 429865
10 program
11 data ?uunit = ?Unit
12 data ?tuple0 = ?T0
13 data ?bundle4intintlist1_oplist1tuple2intint =
    ↳ ?B4intintlist1_oplist1tuple2intint ?wire1int ?wire1int
    ↳ ?wire1list1_op ?wire1list1tuple2intint
14 data ?bundle2intint = ?B2intint ?wire1int ?wire1int
15 data ?bundle4intintlistlist = ?B4intintlistlist ?wire1int ?wire1int
    ↳ ?wire1list ?wire1list
16 data ?bundle2listint = ?B2listint ?wire1list ?wire1int
17 data ?bundle2list1tuple2intintint = ?B2list1tuple2intintint
    ↳ ?wire1list1tuple2intint ?wire1int
18 data ?tuple4intintlist1_oplist1tuple2intint =
    ↳ ?T4intintlist1_oplist1tuple2intint int int ?list1_op
    ↳ ?list1tuple2intint
19 data ?tuple2intint = ?T2intint int int
20 data ?tuple4intintlistlist = ?T4intintlistlist int int ?list1_op
    ↳ ?list1tuple2intint
21 data ?tuple2listint = ?T2listint ?list1tuple2intint int
```

```
22 data ?tuple2list1tuple2intintint = ?T2list1tuple2intintint
     ⌞ ?list1tuple2intint    int
23 data ?list1tuple2intint = ?C1tuple2intint ?tuple2intint ?list1tuple2intint
24                         | ?N1tuple2intint
25 data ?list1_op = ?C1_op _op    ?list1_op
26                | ?N1_op
27 data _op = L
28          | R
29          | P
30          | U
31          | D
32 -- type of main
33 -- val main :: {- no type -}
34 -- Functions
35 {{-## FUND (non-lifted) pos_ok :: (?tuple2intint int int) ->int
     ⌞ ?tuple2intint-}
36  {- argTypes fty= (?tuple2intint int int)-}
37  pos_ok :: (?tuple2intint -> int) (?pos_ok_arg_11 :: ?tuple2intint) =
38  fcase 2 ?pos_ok_arg_11  of
39    (?T2intint xpos   ypos  ) ->
40      glet
41        ?z_55 = 0 ;
42        ?bdg_ypos_56 = ypos ;
43        ?z_57 = (?bdg_ypos_56  == ?z_55 ) ;
44        ?z_58 = 0 ;
45        ?bdg_xpos_59 = xpos ;
46        ?z_60 = (?bdg_xpos_59  == ?z_58 ) ;
47        ?z_61 = (?z_60  && ?z_57 )
48      in if ?z_61
49         then 1
50         else 0
51  esac }
52 {{-## FUND (non-lifted) action :: (?tuple4intintlist1_oplist1tuple2intint
     ⌞ int int (?list1_op _op) (?list1tuple2intint (int,int)))
     ⌞ ->(?tuple2list1tuple2intintint (?list1tuple2intint (int,int)) int)
     ⌞ ?tuple4intintlist1_oplist1tuple2intint-}
53  {- argTypes fty= (?tuple4intintlist1_oplist1tuple2intint int int
      ⌞ (?list1_op _op) (?list1tuple2intint (int,int)))-}
54  action :: (?tuple4intintlist1_oplist1tuple2intint ->
      ⌞ ?tuple2list1tuple2intintint) (?action_arg_11 ::
      ⌞ ?tuple4intintlist1_oplist1tuple2intint) =
55  fcase 4 ?action_arg_11  of
56    (?T4intintlist1_oplist1tuple2intint xpos   ypos   rest   aps  ) ->
57      glet ?bdg_rest_62 = rest
58      in
59      case 1 ?bdg_rest_62  of
60        ((?N1_op) ) ->
61          glet
62            ?bdg_ypos_63 = ypos ;
```

```
63            ?bdg_xpos_64 = xpos ;
64            ?z_65 = ?T2intint ?bdg_xpos_64   ?bdg_ypos_63  ;
65            ?f_pos_ok_66 = (pos_ok $$ ?z_65 ) ;
66            ?bdg_aps_67 = aps
67          in ?T2list1tuple2intintint ?bdg_aps_67   ?f_pos_ok_66
68                                                                |
69        (?C1_op ((L) )  xs  ) ->
70          glet
71            ?bdg_aps_68 = aps ;
72            ?bdg_xs_69 = xs ;
73            ?bdg_ypos_70 = ypos ;
74            ?z_53 = 1 ;
75            ?bdg_xpos_71 = xpos ;
76            ?z_52 = (?bdg_xpos_71  -.i ?z_53 ) ;
77            ?z_54 =
78              ?T4intintlist1_oplist1tuple2intint ?z_52   ?bdg_ypos_70
                  ⌞ ?bdg_xs_69   ?bdg_aps_68
79          in (action $! ?z_54  )
80                                |
81        (?C1_op ((R) )  xs  ) ->
82          glet
83            ?bdg_aps_72 = aps ;
84            ?bdg_xs_73 = xs ;
85            ?bdg_ypos_74 = ypos ;
86            ?z_50 = 1 ;
87            ?bdg_xpos_75 = xpos ;
88            ?z_49 = (?bdg_xpos_75  +.i ?z_50 ) ;
89            ?z_51 =
90              ?T4intintlist1_oplist1tuple2intint ?z_49   ?bdg_ypos_74
                  ⌞ ?bdg_xs_73   ?bdg_aps_72
91          in (action $! ?z_51  )
92                                |
93        (?C1_op ((P) )  xs  ) ->
94          glet
95            ?bdg_aps_76 = aps ;
96            ?bdg_ypos_45 = ypos ;
97            ?bdg_xpos_47 = xpos ;
98            ?z_43 = ?T2intint ?bdg_xpos_47   ?bdg_ypos_45  ;
99            ?z_42 = (?C1tuple2intint ?z_43   ?bdg_aps_76  ) ;
100           ?bdg_xs_77 = xs ;
101           ?bdg_ypos_44 = ypos ;
102           ?bdg_xpos_46 = xpos ;
103           ?z_48 =
104             ?T4intintlist1_oplist1tuple2intint ?bdg_xpos_46
                  ⌞ ?bdg_ypos_44   ?bdg_xs_77   ?z_42
105         in (action $! ?z_48  )
106                               |
107       (?C1_op ((U) )  xs  ) ->
108         glet
```

```
109             ?bdg_aps_78 = aps ;
110             ?bdg_xs_79 = xs ;
111             ?z_40 = 1 ;
112             ?bdg_ypos_80 = ypos ;
113             ?z_39 = (?bdg_ypos_80  -.i ?z_40 ) ;
114             ?bdg_xpos_81 = xpos ;
115             ?z_41 =
116               ?T4intintlist1_oplist1tuple2intint ?bdg_xpos_81  ?z_39
                    ⌞ ?bdg_xs_79  ?bdg_aps_78
117           in (action $! ?z_41  )
118                                  |
119         (?C1_op ((D) )  xs  ) ->
120           glet
121             ?bdg_aps_82 = aps ;
122             ?bdg_xs_83 = xs ;
123             ?z_37 = 1 ;
124             ?bdg_ypos_84 = ypos ;
125             ?z_36 = (?bdg_ypos_84  +.i ?z_37 ) ;
126             ?bdg_xpos_85 = xpos ;
127             ?z_38 =
128               ?T4intintlist1_oplist1tuple2intint ?bdg_xpos_85  ?z_36
                    ⌞ ?bdg_xs_83  ?bdg_aps_82
129           in (action $! ?z_38  )
130       esac
131   esac }
```

9.7.2 Excerpt from the generated constraints

We provide an excerpt of the generated constraint file for the auxiliary function `pos_ok` contained in the Photographer example, as generated to determine the generic cost formula for stack-space usage for this function, which also forms the input to the LP-solver. Again, this excerpt is meant to convey an overview of the nature of the generated constraints, rather than particular details.

The header of the constraint file contains a comment giving some general information, for example that the file contains just 22 constraints over 44 different variables. Any current LP-solver can deal with much larger linear programs containing more than several hundred thousand constraints. The current LP-solver used by our implementation to solve the constraints is LP-solve [10], available under the lesser general public licence (LGPL).

Each constraint is preceded by a label, which identifies the line and column number of the *transformed* program code shown in the previous section, that triggered the generation of that particular constraint. Furthermore a 3-letter code identifies its purpose, for example `MkIn` stands for "Make Integer", i.e. the generation of an integer constant.

Note that variables starting with an `s` indicate simple "slack" variables which occur only once, and which allow imprecision to be introduced where this is necessary. In other words the equations we have provided are, in fact, inequalities. However, the slack variables are used in the objective function to minimise the imprecision as much as possible.

We can also see that automatically generated comments are also present within the constraint file to help readability for the expert user. This makes it easier to understand and verify the reason behind the generation of a particular constraint. Further detailed information about how these constraints are generated is given in EmBounded Deliverable D05 [81].

```
/*
   This file is an automatically generated lp for 'lp_solve (Version 5)'.
       (Constraints for   : "photographer15.art3")
       (Filename should be: "constraints.lp.pos_ok")
       (Processing Date   : Wed Sep 24 10:27:32 BST 2008)

   Contains 22 proper constraints over 44 variables.
       (plus 4 trivial constraints)

   Resource Metric:
    Stack-space costs for executing HAM code via C, with all values being BOXED.
    Selection codes: HumeStackBoxed, StackBoxed, SB, HumecStackBoxed
    Submodels: HumeStackBoxed, HumeStackUnboxed
    Monotone: False
    Revision: 1.1.2.9
    Date: 2008/09/22 11:33:39
    Author: jost

   Annotated Type of main expression:
         pos_ok: V{20}.P(44)
            (?tuple2intint[?T2intint<0>:int,int]) -(x012/y007)-> int

 */

Min: +2 x012 - y007 +1.5 s224 +1.5 s225 +1.5 s226 +1.5 s227 +1.5 s228 +1.5 s229
     +1.5 s230 +1.5 s231 +1.5 s232 +1.5 s233 +1.5 s234 +1.5 s235 +1.5 s236
```

```
     +1.5 s237 +1.5 s238 +1.5 s239 +1.5 s240 +1.5 s241 +1.5 s242 +1.5 s243
     +1.5 s244 +1.5 s245;

CallIn_pos_ok: + x001 - x012 +3 + s245  = 0;
CallOutpos_ok: - y001 + y007 -4 + s244  = 0;
/*
  L0037C002 Function 'pos_ok':  (?tuple2intint[?T2intint:int,int]) -(x001/y001)-> int
 */
L0038C026__MAf: + y001 - y002 -3 + s243  = 0;
L0038C026__MBf: + m001 - m002 +3 + s242  = 0;
L0039C030_Mmi1: - m004 + m001 + s241  = 0;
L0039C030_Mmi2: + x002 + m002 - m004 + s240  = 0;
L0039C030_Msu1: - x001 + x002 + x003 - x005 + m002 + s239  = 0;
L0039C030_Msu2: - x001 + m004 + s238  = 0;
L0039C030_Mtly: + m002 - m003 + s237  = 0;
L0039C030_PCaf: - x004 + x005 -2 + s236  = 0;
L0039C030_PCmi: - x002 + x004 +2 + s235  = 0;
L0041C016_MkIn: - x003 + z000 +1 + s234  = 0;
L0042C023__Var: - z000 + z001 +1 + s233  = 0;
/*
  L0043C016 Binop: "?bdg_ypos_56": int, "?z_55": int{0.0~0.0}, result: bool
 */
L0043C016__OPe: - z001 + z002 -1 + s232  = 0;
L0044C016_MkIn: - z002 + z003 +1 + s231  = 0;
L0045C023__Var: - z003 + z004 +1 + s230  = 0;
/*
  L0046C016 Binop: "?bdg_xpos_59": int, "?z_58": int{0.0~0.0}, result: bool
 */
L0046C016__OPe: - z004 + z005 -1 + s229  = 0;
/*  L0047C016 Binop: "?z_60": bool, "?z_57": bool, result: bool  */
L0047C016__OPa: - z005 + z006 -1 + s228  = 0;
L0048C009__Iei: - z006 + z008 -1 + s227  = 0;
L0048C009__Iti: - z006 + z007 -1 + s226  = 0;
L0049C014_MkIn: + y002 - z007 +1 + s225  = 0;
L0050C014_MkIn: + y002 - z008 +1 + s224  = 0;
0 <= y007 <= 1.0e7;
m001 >= 0;
m002 >= 0;
m003 >= 0;
m004 >= 0;
s224 >= 0;
s226 >= 0;
```

⋮

Further non-negativity constraints for all remaining variables have been omitted from this excerpt for brevity.

9.7.3 Excerpt from the protocol of used stack cost parameters

This is an excerpt of the stack-space cost parameters used in that particular run of our analysis for the photographer example on the auxiliary function `pos_ok`.

```
L0038C026__MBf VAL    3 FOR RCmatch (FunCase (Just 2)) PBefore
L0038C026__MAf VAL   -3 FOR RCmatch (FunCase (Just 2)) PAfter
L0039C030_PCmi VAL    2 FOR RCpattern (PCon "?T2intint" [PVar "xpos",PVar "ypos"])
                                      (Just P3Middle)
L0039C030_PCaf VAL   -2 FOR RCpattern (PCon "?T2intint" [PVar "xpos",PVar "ypos"])
                                      (Just P3After)
L0041C016_MkIn VAL    1 FOR RCmk IntTyp (IntVal 0)
L0042C023__Var VAL    1 FOR RCpvar 0
L0043C016__OPe VAL   -1 FOR RCbop = IntTyp IntTyp
L0044C016_MkIn VAL    1 FOR RCmk IntTyp (IntVal 0)
L0045C023__Var VAL    1 FOR RCpvar 0
L0046C016__OPe VAL   -1 FOR RCbop = IntTyp IntTyp
L0047C016__OPa VAL   -1 FOR RCbop && BoolTyp BoolTyp
L0048C009__Iti VAL   -1 FOR RCif True PBefore
L0048C009__Iei VAL   -1 FOR RCif False PBefore
L0049C014_MkIn VAL    1 FOR RCmk IntTyp (IntVal 1)
L0050C014_MkIn VAL    1 FOR RCmk IntTyp (IntVal 0)
CallIn_pos_ok  VAL    3 FOR RCapp True "pos_ok"
                                [ConTyp "?tuple2intint" []] PBefore ExactApp 0
CallOutpos_ok  VAL   -4 FOR RCapp True "pos_ok"
                                [ConTyp "?tuple2intint" []] PAfter ExactApp 0
```

9.7.4 Excerpt from the protocol of used time parameters

This is an excerpt of the time parameters used in that particular run of our analysis for the photographer example. Following Section 9.7.2, we concentrate on WCET parameters used for the translated source code lines 40–50. As described above, we can see at label `L0040C031_MrS` the WCET parameter for a successful pattern match, and that it was mapped to the value 10 by the current cost table for executing Schopenhauer on a Renesas M32C/85U processor.

Note that some cost parameters are themselves parameterised. For example, the WCET for pushing a variable onto the top of the stack depends on whether the variable is stored inside the current frame or deeper. We see that the WCET for pushing a variable from the current frame to the top of the stack is 39 clock cycles, while pushing a variable from the previous frame has a WCET of 46 clock cycles. Chapter 8 contains more information about how these WCET parameters were obtained using AbsInt's **aiT** tool.

```
AT L0040C031_Mr_ VAL   20 FOR Tmatchrule
AT L0040C031_Pci VAL   30 FOR Tmatchcon
AT L0040C031_Pcm VAL  174 FOR (Tcopyarg + Tunpack 2)
AT L0040C031_Pci VAL   30 FOR Tmatchcon
AT L0040C031_Pvr VAL   36 FOR Tmatchvar
AT L0040C031_Pco VAL    9 FOR Tpop
AT L0040C031_Pcf VAL    9 FOR Tpop
AT L0040C031_MrS VAL   10 FOR Tmatchedrule
AT L0041C032_Var VAL   46 FOR Tpushvar 1
AT L0042C032_Var VAL   46 FOR Tpushvar 1
AT L0043C030_Var VAL   39 FOR Tpushvar 0
AT L0044C031_Var VAL   46 FOR Tpushvar 1
AT L0045C032_Var VAL   46 FOR Tpushvar 1
AT L0046C032_Var VAL   46 FOR Tpushvar 1
AT L0047C025_Con VAL  182 FOR Tmktuple 2
AT L0048C025_Con VAL  215 FOR Tmkcon 2
AT L0049C025_Con VAL  286 FOR Tmktuple 4
AT L0050C014_Apt VAL   85 FOR Ttailcall 1 1
```

9.8 Hume Code for the PID Controller Example

This appendix contains the Hume source code of the 2-degrees-of-freedom PID controller for the classic "ball and beam" experiment from Deliverable D7 (WP8a).

```
program

interrupt p  from "TIMERB0";
--interrupt b1 from "INT0";
--interrupt b2 from "INT1";
--interrupt b3 from "INT2";
memory p2 to "P2:p8";
memory ad00 from "AD00L:p8";
memory ad01 from "AD01L:p8";
memory da0 to "DA0:p8";
memory da1 to "DA1:p8";

type integer = int 16;
type real = float 32;

-- PID controller with rectangular integration:
-- Pk = Kp (ek - e(k-1))
-- Ik = Ki T ek
-- Dk = (Kd/T) (ek - 2 e(k-1) + e(k-2))
-- CVk = CVk-1 + Pk + Ik + Dk
-- where ek = SPk - PVk, and T is the sampling interval.

-- Sample rate
constant vT = 0.001; -- just a guess

-- Control constants
constant vKp = 0.2;
constant vKi = 0.5;
constant vKd = 0.008;

-- Ramp limit
constant vPIDlim = 0.001;

-- Limit function
limit x mi ma = if x < mi then (mi::real) else if x > ma then ma else x;
limitR x r = limit x (0.0 - r) r;

-- Scale inputs/outputs
-- We also limit output to 0 to 255
scale_in x = ((x :: integer) as real) / 255.0;
scale_out x = if x < 0.0 then 0
              else if x > 1.0 then 255 else ((x * 255.0) as integer);

-- Compute the error output
rabs x = if x < 0.0 then (0.0 - x) else x;
```

```
cerr sp pv = scale_out (rabs ((scale_in pv) - (scale_in sp)));

-- Inconvenient types
type lvs_t = (bool,real,real,real,integer);

-- Initial control variable state
constant cv_init = 127; -- need word<->int casts !!! 0x7f;

-- Initial loop variable for controller box
constant lvs_init = (false,0.0,0.0,0.0,0);

-- The control loop.
-- Inputs:  p   - run off the b0 timer.
--          sp  - set point (AD0).
--          pv  - process variable (AD1).
--          lvs - loop variables (init flag, errors, last time, mode).
-- Outputs: cv  - output to plant (DA0).
--          err - error output (DA1).
--          lvs - loop variables.
box ctlr
  in  (p::(), sp::integer, pv::integer, lvs::lvs_t)
  out (p2::integer, cv::integer, err::integer, lvs'::lvs_t)
match
   (_,sp,pv,(false,_,_,_,_)) ->
    (0,cv_init,cerr sp pv,(true,0.0,0.0,scale_in cv_init,0))
 | (_,sp,pv,(true,ek1,ek2,cv1,led)) ->
    let ek = (scale_in pv) - (scale_in sp) in
    let vPk = vKp * (ek - ek1) in
    let vIk = vKi * vT * ek in
    let vDk = (vKd / vT) * (ek - 2.0 * ek1 + ek2) in
    let vPIDk = limitR (vPk + vIk + vDk) vPIDlim in
    let vCVk = cv1 + vPIDk in
      (led+1,scale_out vCVk,cerr sp pv,(true,ek,ek1,vCVk,led+1));

wire ctlr (p, ad00, ad01, ctlr.lvs' initially lvs_init)
          (p2, da0, da1, ctlr.lvs);
```

9.9 Hume Code for the Inverted Pendulum Example

This section contains the source code for the inverted pendulum example from as used to perform the analysis. The single box at the end of the code has several branches: the first three deal with the pressing of button 1, starting, resetting and stopping the pendulum control loop; the fourth and fifth deal with button 2 and button 3, allowing a calibration the motor output while the control loop is stopped; the sixth applies when there are no button presses while the control loop is stopped, i.e. it does nothing; and the last is the actual control loop. Note that only the last branch performs any actual work.

```
--program

-- Streams
interrupt b1 from "INT0";
interrupt b2 from "INT1";
interrupt b3 from "INT2";
memory p2 to "P2:p8";
memory ad00 from "AD00L:p8";
memory ad01 from "AD01L:p8";
memory da0 to "DA0:p8";
memory da1 to "DA1:p8";

-- Types
type byte = word 8;
type integer = int 16;
type real = float 32;

type _vec = vector 4 of real;

-- Input/Output scaling for ADC/DAC

#define inputSensitivity 0.081415926535                 // by experiment
#define scaleIn   ((2.0 * inputSensitivity) / 255.0)    // 0x00 to 0xff
#define offsetIn  (0.0 - inputSensitivity)

b2r :: byte -> real;
b2r b = (((b :: byte) as real) * scaleIn) + offsetIn;

#define outputSensitivity  0.5                  // +/-0.5
#define minOut    (0.0 - outputSensitivity)     // 0 volts
#define maxOut    outputSensitivity             // 5 volts
#define scaleOut  ((maxOut - minOut) * 255.0)   // 0x00 to 0xff
#define offsetOut (scaleOut / 2.0)              // 127.5;

r2b :: real -> byte;
r2b r =
  if r < minOut
  then 0x00
  else if r > maxOut
       then 0xff
       else ((((r :: real) * scaleOut) + offsetOut) as byte);

-- Limit motor output

#define motorZero  0x8c
#define motorMarg  0x0c
```

```
#define motorMin   (motorZero - motorMarg)
#define motorMax   (motorZero + motorMarg)

#define motorZeroR (((motorZero as real) - offsetOut) / scaleOut)

limitMotor :: byte -> byte;
limitMotor w = if w < motorMin then motorMin else
               if w > motorMax then motorMax else w;

invertMotor :: byte -> byte;
invertMotor w =
  if w > motorZero
  then
    let delta = w - motorZero in
      if delta > motorZero
      then 0x00
      else motorZero - delta
  else
    let delta = motorZero - w in
      if delta > (0xff - motorZero)
      then 0xff
      else motorZero + delta;

-- Multiply vector by #define
mult1441 :: _vec -> _vec -> real;
mult1441 x y = (x@1)*(y@1) + (x@2)*(y@2) + (x@3)*(y@3) + (x@4)*(y@4);

{- Control #defines computed by Octave -}

-- define K = << (-0.65900), (-21.00773),  (-1.31144),  (-3.36854) >>;

{- Emulate RTC with timers -}

#define clkTick  0.001 // Time for one timer tick
#define cntrTicks  2.0 // Number of ticks between timer interrupts
-- You need to match this with the clock config on the board

#define dT  0.002      // clkTick * cntrTicks; -- Computed clock tick

box counter
  in (p::int 32, s::real)
  out (p'::int 32, s'::real, clk::real)
match
   (0,s) -> (100,s+dT,s+dT)
 | (p,s) -> (p-1,s,*);

wire counter
  (counter.p' initially 100, counter.s' initially 0.0)
  (counter.p, counter.s, regu.clk);

{- Control box -}

#define thetaMin  0x40
#define thetaZero  0x7f
#define thetaMax  0xe0
```

```
#define thetaZero_r  (thetaZero as real)
#define thetaSensitivity  0.2

type etat_t = (bool,bool,real,real,real,byte,byte);

#define offsetZero  0x3f

stop_etat :: byte -> etat_t;
stop_etat    ao = (false,false,0.0,0.0,0.0,ao,0x00);

run_etat :: byte -> etat_t;
run_etat     ao = (true, false,0.0,0.0,0.0,ao,0x00);

deriv_etat :: byte -> etat_t;
deriv_etat   ao = (true, true, 0.0,0.0,0.0,ao,0x00);

init_etat :: () -> etat_t;
init_etat u = stop_etat (offsetZero - 0x26);

applyOffset :: byte -> byte -> byte;
applyOffset offset angle =
  if offset > offsetZero
  then angle + (offset - offsetZero)
  else angle - (offsetZero - offset);

rabs :: real -> real;
rabs r = if (r::real) < 0.0 then (0.0 - r) else r;

getTO :: byte -> byte;
getTO theta_w =
  let theta_r = ((theta_w :: byte) as real) in
  let diff = rabs (theta_r - thetaZero_r) in
  let diff_w = ((diff * thetaSensitivity) as byte) in
    if theta_r < thetaZero_r
    then offsetZero + diff_w
    else offsetZero - diff_w;

box regu
  in (b1::(), b2::(), b3::(), clk::real, alpha_w::byte, theta_w::byte, etat::etat_t)
  out (moteur_w::byte, erreur_w::byte, compteur_w::byte, etat'::etat_t)
match
    (_,*,*,_,_,_,(false,false,_,_,_,ao,_))    -> (*,*,*,run_etat ao)              -- b1 (run)
  | (_,*,*,_,_,_,(true, false,_,_,_,ao,_))    -> (*,*,*,deriv_etat ao)            -- b1 (deriv)
  | (_,*,*,_,_,_,(true, true, _,_,_,ao,_))    -> (motorZero,0x00,*,stop_etat ao)  -- b1 (stop)
  | (*,_,*,_,_,_,(tf,dv,ap,tp,lc,ao,cnt)) -> (*,*,*,(tf,dv,ap,tp,lc,ao+0x01,cnt)) -- b2 (motor+)
  | (*,*,_,_,_,_,(tf,dv,ap,tp,lc,ao,cnt)) -> (*,*,*,(tf,dv,ap,tp,lc,ao-0x01,cnt)) -- b3 (motor-)
  | (*,*,*,_,alpha_w,_,(false,dv,ap,tp,lc,ao,cnt)) ->                             --   (stopped)
      (*,*,alpha_w,(false,dv,ap,tp,lc,ao,cnt))
  | (*,*,*,c,alpha_w,theta_w,(true,deriv,alphaPrec,thetaPrec,lc,alphaOffset,cnt)) --   (running)
   ->
    let te = (c - lc) in
    let theta_r = ((theta_w :: byte) as real) in
    let ek = theta_r - thetaZero_r in
    let thetaOffset = getTO theta_w in
    let aoo = applyOffset thetaOffset alphaOffset in
```

```
    let alpha_wo = applyOffset aoo alpha_w in
    let alpha = b2r alpha_wo in
    let theta = b2r theta_w in
      if theta_w < thetaMin || theta_w > thetaMax
      then (motorZero,*,*,(true,deriv,alpha,theta,c,alphaOffset,cnt))
      else
        let xest =
          if deriv
          then
            let thetaPt = (theta-thetaPrec)/te in
            let alphaPt = (alpha-alphaPrec)/te in
              << theta, alpha, thetaPt, alphaPt >>
          else << theta, alpha, 0.0, 0.0 >>
        in
        let kay = << (-0.65900), (-21.00773),  (-1.31144),  (-3.36854) >> in
        let v = 0.0 - mult1441 kay xest in
        let motorOut = invertMotor (limitMotor (r2b v)) in
          (motorOut,0x00,cnt,(true,deriv,alpha,theta,c,alphaOffset,cnt+0x01));

wire regu (b1, b2, b3, counter.clk, ad00, ad01, regu.etat' initially init_etat ())
          (da0, da1, p2, regu.etat);
```

Bibliography

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. Technical Report RS/03/13, BRICS, March 2003.

[2] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

[3] J. Apple and W. Weimer. Simulating dependent types with guarded algebraic datatypes. draft paper, 2007.

[4] D. Aspinall, L. Beringer, and A. Momigliano. Optimisation Validation. In *COCV06 — Fifth Workshop on Compiler Optimization Meets Compiler Verification*, Vienna, Austria, April 2, 2006.

[5] L. Augustsson. Cayenne — a language with dependent types. In *Proceddings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 239–250. ACM, 1998.

[6] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, 1987.

[7] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 337–354, San Diego, California, USA, 2003. Springer-Verlag, Berlin. URL `http://www.cs.unipr.it/ppl/Documentation/BagnaraHRZ03.pdf`.

[8] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

[9] N. Benton, J. Hughes, and E. Moggi. Monads and effects. Lecture Notes of the Applied Semantics Summer School, September 2000.

[10] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`.

[11] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1996.

[12] B. Bjerner and S. Holmstrm. A compositional approach to time analysis of first order lazy functional programs. In *Proceedings of the 4th International Conference on Functional Programming*

Languages and Computer Architecture, pages 157–165, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-328-0. doi: http://doi.acm.org/10.1145/99370.99382.

[13] A. Bonenfant, Z. Chen, K. Hammond, G. Michaelson, A. Wallace, and I. Wallace. Towards resource-certified software: A formal cost model for time and its application to an image-processing example. In *ACM Symposium on Applied Computing (SAC '07), Seoul, Korea, March 11-15*, 2007. URL `http://www-fp.cs.st-andrews.ac.uk/embounded/pubs/papers/sac2007.pdf`.

[14] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckman. Worst-case execution times for a purely functional language. In *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*. Springer, 2007.

[15] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-Case Execution Times for a Purely Functional Language. In *Proc. 2006 Intl. Symp. on Impl. and Appl. of Functional Langs. (IFL 2006)*, pages 235–252. Springer-Verlag LNCS 4449, 2007.

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

[17] E. Brady and K. Hammond. A dependently typed framework for static analysis of program execution costs. In *Implementation of Functional Languages (IFL) 2005*, volume 4015 of *Lecture Notes in Computer Science*, pages 74–90, Berlin/Heidelberg, 2006. Springer. ISBN 978-3-540-69174-7.

[18] S. Breitinger, R. Loogen, Y. Ortéga Mallén, and R. Peña Marí. Eden — the Paradise of Concurrent Functional Programming. In *Europar'96*, volume 1123 of *LNCS*, 1996.

[19] B. Campbell. *Type-based amortized stack memory prediction*. PhD thesis, University of Edinburgh, 2008.

[20] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.

[21] V. Chandru. Variable elimination in linear constraints. *The Computer Journal*, 36(5):463–472, 1993.

[22] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[23] W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher Order and Symbolic Computation*, 14(2-3):261–300, 2001. ISSN 1388-3690.

[24] W.-N. Chin, S.-C. Khoo, and D. N. Xu. Extending sized type with collection analysis. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM'03)*, pages 75–84, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-667-6. doi: http://doi.acm.org/10.1145/777388.777397.

[25] C. Coquand and T. Coquand. Structured type theory. In *Proceedings of the Workshop on Logical Frameworks and Meta-languages*, 1999. URL `citeseer.ist.psu.edu/article/coquand99structured.html`.

[26] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[27] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[28] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[29] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[30] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming (PLILP'92)*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[31] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[32] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[33] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 184–198, New York, NY, USA, 2000. ISBN 1-58113-125-9. doi: http://doi.acm.org/10.1145/325694.325716.

[34] U. Dal Lago and S. Martini. An invariant cost model for the lambda calculus. arXiv:cs.LO/0511045v1, November 2005.

[35] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Department of Computer Science, University of Edinburgh, 1985.

[36] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, 2008.

[37] O. Danvy. A rational deconstruction of the SECD machine. Technical Report RS/03/33, BRICS, October 2003.

[38] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):33–45, March 1992. URL `http://www.psrg.lcs.mit.edu/ftpdir/papers/loplas.dvi`.

[39] S. P. (ed.), R. Hughes, L. Augustsson, D. Barton, B. Boutel, F. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge Univerity Press, 2003.

[40] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems, Saarland University, Saarbrücken, Germany*. PhD thesis, 1997.

[41] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computerd. Programming*, 35(2-3):163–189, 1999. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(99)00010-6.

[42] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the First International Workshop on Embedded Software*, pages 469–485, London, UK, 2001. Springer-Verlag. ISBN 3-540-42673-6.

[43] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of the 2nd European Symposium on Programming*, pages 155–175. North-Holland, 1988.

[44] Ghc. The Glasgow Haskell Compiler. URL `http://haskell.org/ghc`. `http://haskell.org/ghc`.

[45] A. GmbH. http://www.absint.com. 2007.

[46] B. Grobauer. Cost recurrences for DML programs. In *Proceedings of the 6th ACM SIGPLAN international conference on Functional programming*, pages 253–264, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-415-0. doi: http://doi.acm.org/10.1145/507635.507666.

[47] G. Grov. Verifying the Correctness of Hume Programs - An Approach Combining Algorithmic and Deductive Reasoning. In *Proceedings of the* 20^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE-05)*, pages 444–447. ACM Press, 2005.

[48] G. Grov and G. Michaelson. Towards a Box Calculus for Hume. In M. Morazan and H. Nilsson, editors, *Draft Proceedings 8th Symposium on Trends in Functional Programming, TR-SHU-CS-2007-04-1, Seton Hall University*, April 2007.

[49] G. Grov, A. Ireland, and G. Michaelson. Model Checking Hardware Hume. Technical Report HW-MACS-TR-0032, School of Mathematical and Computer Sciences, Heriot-Watt University, 2005.

[50] G. Grov, R. Pointon, G. Michaelson, and A. Ireland. Coordination, Computation and Hume Scheduling. 2008. in preparation.

[51] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, Université Scientifique et Médicale de Grenoble, 1979.

[52] K. Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *Central European Functional Programming School (CEFP05)*, pages 100–134, Budapest, Hungary, July 4–15, 2005. Springer-Verlag LNCS 4164.

[53] K. Hammond. An Abstract Machine for Resource Bounded Computations in Hume. In *IFL03 — Intl Workshop on Implementation of Functional Languages*, pages 79–94, Edinburgh, Scotland, Sept. 2003. Draft proceedings.

[54] K. Hammond. Hume: a Bounded Time Concurrent Language. In *Proceedings of the IEEE Conf. on Electronics and Control Systems (ICECS '2K)*, pages 407–411, Kaslik, Lebanon, Dec. 2000. IEEE Press.

[55] K. Hammond and G. Michaelson. "Carbon Credits" for Resource-Bounded Computations using Amortised Analysis. In *Proc. FM2009: 16th International Symposium on Formal Methods Eindhoven, the Netherlands, November 2 - November 6, 2009*, Lecture Notes in Computer Science. Springer-Verlag, 2009.

[56] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[57] K. Hammond and G. Michaelson. Predictable Space Behaviour in FSM-Hume. In *14th Intl Workshop on Implementation of Functional Languages*, pages 386–403, Madrid, Spain, Sept. 2002.

[58] K. Hammond and G. Michaelson. The Hume Report. Available from: `http://www-fp.dcs.st-and.ac.uk/hume/report/hume-report.ps`. Version 0.3.

[59] K. Hammond, R. Dyckhoff, C. Ferdinand, R. Heckmann, M. Hofmann, S. Jost, H.-W. Loidl, G. Michaelson, J. Sérot, and A. Wallace. The embounded project (project paper). In *Proc. 6th Symposium on Trends in Functional Programming (TFP 2005), Tallinn, Estonia, 23-24 September 2005*, volume 6 of *Trends in Functional Programming*. Intellect, 2006.

[60] K. Hammond, C. Ferdinand, R. Heckmann, R. Dyckhoff, M. Hofman, S. Jost, H.-W. Loidl, G. Michaelson, R. Pointon, N. Scaife, J. Sérot, and A. Wallace. Towards formally verifiable WCET analysis for a functional programming language. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, number 06902 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2006/677`.

[61] K. Hammond, C. Ferdinand, R. Heckmann, R. Dyckhoff, M. Hofmann, S. Jost, H.-W. Loidl, G. Michaelson, R. Pointon, N. Scaife, J. Sérot, and A. Wallace. Towards formally verifiable resource bounds for real-time embedded systems. *ACM SIGBED Review— Special issues, ITCES06*, 3(4):27–36, October 2006. ISSN 1551-3688. doi: 10.1145/1183088.1183093.

[62] K. Hammond, G. Michaelson, and P. Vasconcelos. Bounded space programming using finite state machines and recursive functions: the hume approach. *ACM Transactions on Software Engineering Methodology*, 2006. Submitted.

[63] C. A. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon. Automatic amortised worst-case execution time analysis. In *7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, Proceedings*, pages 13–18, 2007. URL `http://www.irit.fr/wcet2007/`.

[64] C. A. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon. Automatic amortised worst-case execution time analysis. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. URL `http://drops.dagstuhl.de/opus/volltexte/2007/1186`.

[65] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. ISSN 1236-6064.

[66] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *ESOP 2006*, LNCS 3924, pages 22–37. Springer-Verlag, 2006.

[67] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197, 2003.

[68] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL'03: Symp. on Principles of Programming Langs.*, pages 185–197, New Orleans, LA, USA, Jan. 2003. ACM Press.

[69] M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In P. Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, March 2006. URL `http://www.dcs.st-andrews.ac.uk/~jost/research/`.

[70] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003. URL `http://www.tcs.informatik.uni-muenchen.de/~jost/research/full_p17-hofmann.ps`.

[71] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proc. WCET '02: Int'l Workshop on Worst-Case Execution Time Analysis*, June 19-21 2002.

[72] C. Hope and G. Hutton. Accurate Step Counting. In *Implementation and Application of Functional Languages*, volume 4015 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006. Selected papers from the 17th International Workshop on Implementation and Application of Functional Languages, Dublin, Ireland, September 2005.

[73] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34, pages 70–81, Sept. 1999.

[74] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In G. L. S. Jr, editor, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, volume 23, St Petersburg, Florida, Jan. 1996. ACM.

[75] R. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, pages 129–146. Springer-Verlag LNCS 201, 1985.

[76] Coq development team. *The Coq proof assistant — reference manual*. INRIA, 8.1 edition, 2006.

[77] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, volume 1782, pages 230–244, 2000. URL `citeseer.ist.psu.edu/jones00type.html`.

[78] S. P. Jones, D. Vytiniostis, S. Weirich, and G. Washburn. Simple unification-based type inferece for GADTs. In *Proceedings of the International Conference on Functional Programming*, 2006.

[79] S. Jost. Prototype Implementation of Space Analyses. EmBounded Project Deliverable, Feb. 2007. Deliverable D13.

[80] S. Jost, H.-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, Feb. 2007. Deliverable D11.

[81] S. Jost, H.-W. Loidl, and K. Hammond. Report on Stack-space Analysis. EmBounded Project Deliverable, Feb. 2007. Deliverable D5.

[82] S. Jost, H.-W. Loidl, K. Hammond, M. Hofmann, and A. Bonenfant. Generic amortised resource analysis for higher-order functional programs. In Preparation, 2007.

[83] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations. In *ICFP'08 — Intl. Conference on Functional Programming*, Oct. 2008. Submitted.

[84] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'91)*, pages 303–310, 1991. URL `http://www.psrg.lcs.mit.edu/ftpdir/papers/popl91.dvi`.

[85] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 126–140. Springer-Verlag, 1996. ISBN 3-540-60765-X. URL `citeseer.ist.psu.edu/kelly95transitive.html`.

[86] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental algorithms. Addison-Wesley, second edition, 1973.

[87] P. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[88] P. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, Jan. 1964.

[89] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[90] D. Le Mtayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):248–266, April 1988.

[91] H. R. Lewis and C. H. Papadimitriou. *Elements of the theory of computation.* Prentice-Hall International, 1981.

[92] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition edition, Apr. 1999.

[93] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. WCET '03: Int'l Workshop on Worst-Case Execution Time Analysis*, pages 99–102, 2003.

[94] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, pages 31–40, London, UK, 1998. Springer-Verlag. ISBN 3-540-65075-X.

[95] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998. URL `http://www.cee.hw.ac.uk/~{}dsg/gph/papers/ps/loidl-thesis.ps.gz`.

[96] H.-W. Loidl, S. Jost, and N. Scaife. Validation of Cost Model. EmBounded Project Deliverable, Apr. 2008. Deliverable D28.

[97] J. M. Lucassen. *Types and Effects: Towards the Integration of Functional an Imperative Programming.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1987.

[98] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73564.

[99] Z. Luo and R. Pollack. Lego proof development system: User's manual. Technical report, Department of Computer Science, University of Edinburgh, 1992.

[100] D. MacQueen. Standard ML of New Jersey. Web page. `http://www.smlnj.org/`.

[101] D. B. MacQueen and R. Sethi. A semantic model of types for applicative languages. In *Proceedings of the ACM symposium on LISP and functional programming*, pages 243–252, New York, NY, USA, 1982. ACM. ISBN 0-89791-082-6.

[102] C. McBride. Faking it: simulating dependent types in Haskell. *Journal of Functional Programming*, 12(5):375–392, 2002. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796802004355.

[103] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1): 69–111, 2004.

[104] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference*, pages 465–469, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/800196.806014.

[105] G. Michaelson, K. Hammond, and J. Sérot. The Finite State-Ness of FSM-Hume. In *Trends in Functional Programming*, volume 4, pages 19–28. Intellect, 2004.

[106] G. Michaelson, A. Wallace, K. Hammond, I. Wallace, A. Bonenfant, and Z. Chen. Toward resource certified image processing software. In *Systems Engineering for Autonomous Systems Defence Technology Centre (SEAS DTC), Annual Technical Conference, July 2006, Edinburgh, Conference Proceedings*, page A15. UK MoD, 2006. URL `http://www.seasdtc.com/events/2006_annual_conf_materials.htm`.

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

[108] J. C. Mitchell. Coercion and type inference. In *Proceedings of the 11th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'84)*, pages 175–185, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: http://doi.acm.org/10.1145/800017.800529.

[109] T. Mitra and A. Roychoudhury. A Framework to Model Branch Prediction for WCET Analysis. In *Proc. 2nd Workshop on Worst Case Execution Time Analysis (WCET 2002)*, June 2002.

[110] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games – Volume II*, number 28 in Annals of Mathematics Studies, pages 51–73. Princeton University Press, Princeton, New Jersey, 1953.

[111] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, New York, NY, USA, 1997. ISBN 0-89791-853-3. doi: http://doi.acm.org/10.1145/263699.263712.

[112] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, pages 114–136, 1999. URL `citeseer.ist.psu.edu/nielson99type.html`.

[113] F. Nielson, H. R. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The algorithm. In *Logical and Operational Methods in the Analysis of Programs and Systems*, pages 207–243, 1996. URL `citeseer.ist.psu.edu/article/nielson97polymorphic.html`.

[114] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[115] H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Logical and Operational Methods in the Analysis of Programs and Systems*, pages 141–171, 1996. URL `citeseer.ist.psu.edu/article/nielson97polymorphic.html`.

[116] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[117] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[118] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[119] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357172.357178.

[120] L. Pareto. Sized types. Dissertation for the Licentiate Degree in Computing Science, Department of Computing Science, Chalmers University of Technology, 1998. ISBN 91-7197-682-5.

[121] L. Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, Gteborg, 2000.

[122] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine. *J. of Functional Programming*, 2(2):127–202, 1992.

[123] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall, 1992.

[124] A. R. Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. Cost analysis using automatic size and time inference. In *Proceedings of the 14th International Workshop on Implementation of Functional Languages*, volume 2670 of *LNCS*. Springer-Verlag, 2003.

[125] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[126] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM TECS: Trans. on Embedded Computing Sys.*, 4(4):751–778, 2005. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/1113830.1113833.

[127] B. Reistad and D. K. Gifford. Static Dependent Costs for Estimating Execution Time. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming – LFP '94*, pages 65–78, Orlando, FL, June 1994.

[128] J. A. Robinson. Computational logic: The unification computation. *Machine Inteligence*, 6: 63–72, 1971.

[129] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the 1989 International Conference on Functional Programmings Languages and Computer Architecture (FPCA'89)*, pages 144–156, 1989.

[130] N. Scaife, H.-W. Loidl, G. Michaelson, and J. Sérot. Evaluation of Hume and the Hume Methodology. EmBounded Project Deliverable, Sept. 2008. Deliverable D33.

[131] S.-B. Scholz. *Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen (in German).* PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, Oct. 1996. URL `http://www.informatik.uni-kiel.de/~{}sacbase/papers/sac-design-sbs-phd-96.ps.gz`.

[132] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1986. ISBN 0-471-98232-6.

[133] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems. In *Proc. ISOLA '06: Int'l Symp. on Leveraging Applications of Formal Methods*, Paphos, Cyprus, November 2006.

[134] J. Serot and N. Scaife. Real-time testbed applications. Deliverable 07 of project EmBounded (IST-510255), Sept. 2006. Laboratoire LASMEA, Blaise Pascal University.

[135] J. Serot and N. Scaife. Real-time testbed applications. EmBounded Project Deliverable, Sept. 2006. Deliverable D7.

[136] H. Simes, K. Hammond, M. Florido, and P. Vasconcelos. Using intersection types for cost-analysis of higher-order polymorphic functional programs. In *Proceedings of the Types Project*, volume 4502 of *Lecture Notes in Computer Science*. Springer, 2007.

[137] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. 5th Intl Workshop on WCET Analysis*, pages 21–24, 2005.

[138] I. Systems. http://www.iar.com/. *Home Page*, 2006. URL `http://www.renesas.com/`.

[139] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[140] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111 (2):245–296, 1994.

[141] L. Tan. The Worst Case Execution Time Tool Challenge 2006: Technical Report for the External Test. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, 2006.

[142] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[143] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[144] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, Dec. 1998.

[145] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1988.

[146] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998. URL `citeseer.ist.psu.edu/tofte98region.html`.

[147] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997. URL `citeseer.ist.psu.edu/tofte97regionbased.html`.

[148] D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7): 751–768, July 2004.

[149] D. A. Turner. Elementary strong functional programming. In *Proceedings of the First International Symposium on Functional Programming Languages in Education*, pages 1–13, London, UK, 1995. Springer-Verlag. ISBN 3-540-60675-0.

[150] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Technical Report 538, Computer Science Dept, Indiana University, Apr. 2000.

[151] L. Unnikrishnan, S. Stoller, and Y. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In *Proc. LCTES '01: ACM workshop on Languages, Compilers and Tools for Embedded Systems*, pages 102–111, 2001. ISBN 1-58113-425-8. doi: http://doi.acm.org/10.1145/384197.384212.

[152] P. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, 2008.

[153] P. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, Feb. 2008.

[154] P. Vasconcelos and K. Hammond. Inferring cost equations for recursive, higher-order and polymorphic functional programs. In *Proceedings of the 14th International Workshop on Implementation of Functional Languages*, volume 3145 of *LNCS*. Springer-Verlag, 2004.

[155] H. L. Verge. A note on Chernikova's algorithm. *Publication interne* 635, IRISA, Campus de Beaulieu, Rennes, France, 1992.

[156] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 119–132, 1988.

[157] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993. URL `citeseer.ist.psu.edu/wadler95monads.html`.

[158] P. Wadler. The marriage of effects and monads. In *Proceddings of the International Conference on Functional Programming*, Baltimore, 1998.

[159] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[160] D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, Dec. 1993. URL `ftp://ftp.ee.byu.edu/faculty/Wilde/Polylib/report.ps.gz`. Also published as IRISA *Publication interne* 785, Rennes, France, 1993.

[161] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. Accepted for ACM Transactions on Embedded Computing Systems (TECS), 2007.

[162] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995. URL `citeseer.ist.psu.edu/wright95simple.html`.

[163] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[164] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227, 1999.