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

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

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

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

The Hume Report, Version 1.1

Kevin Hammond[1] Greg Michaelson[2] Robert Pointon[2]

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

[2]School of Mathematical and Computer Sciences
Heriot-Watt University
{greg,rpointon}@macs.hw.ac.uk, +44 131 451 3422

Contents

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 1

Introduction

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

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

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

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

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

Appendix D

Standard Prelude

Summary of Standard Hume Functions and Operators

Operations on *int p* types

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

Operations on *nat p* types

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

Operations on *word p* types

```
+, - :: Word -> Nat -> Word
==, !=, <=, <, >, >= :: Word -> Word -> Bool
rotl, rotr, lshl, lshr :: Word -> Nat -> Word
```

Operations on *float p* types

```
+, -, *, /, ** :: Float -> Float -> Float
sin, cos, tan, asin, acos, atan, log, exp, sqrt,
  ln, log10, sinh, cosh, tanh :: Float -> Float
atan2 :: Float -> Float -> Float
==, !=, <=, <, >, >= :: Float -> Float -> bool
```

Operations on vector types

```
length :: <<a>> -> Int
@ :: <<a> -> Int -> a
vecdef :: Int -> (Int->a) -> <<a>>
vecmap :: <<a>> -> (a->b) -> <<b>>
vecfoldr :: <<a>> -> b -> (a->b->b) -> b
update :: <<a>> -> Int -> a -> <<a>>
==, !=, <=, <, >, >= :: <<a>> -> <<a>> -> Bool
```

C.7 Dynamic Semantics: The Initial Environment

The initial environment comprises definitions for all functions and constructors defined in the module *Prelude*. These values must be available in all Hume programs. The meanings of other *Prelude* functions is defined by reference to Appendix D, which provides a source language definition. We assume that the meaning of basic operations (such as addition on numbers) is obvious. To define this formally would be tedious in the extreme. We will also assume without formal specification that the initial environment for some Hume program will include definitions for those functions and values that are imported into a Hume program, whether or not these were originally defined in Hume (i.e. whether or not they are "foreign" functions).

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

PrimPlusInt $\mapsto$ (a, b) -> $a + b$ $+$ is fixed-precision integer addition
PrimMulInt $\mapsto$ (a, b) -> $a \times b$ $\times$ is fixed-precision integer multiplication
...

plus the standard constructors (*BasCon*):

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

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

1.1 Motivation and objectives

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

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

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

Hume reflects these considerations in:

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

1.1.1 Important Design Characteristics

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

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

Both system level and process level exceptions are supported, including the ability to set timeouts for expression computations. Exceptions may be raised from within the expression language but

can only be handled by the process language. This reduces the cost of handling exceptions and maintains a pure expression language, as well as simplifying the expression cost calculus.

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

1.2 Language Structure

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

1.2.1 The Hume Expression Language

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

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

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

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

1.2.2 The Hume Coordination Language

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

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

1.3 Tools to Support the Hume Language

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

$$\frac{v = \langle\rangle \qquad E \vdash exp \Rightarrow v'}{E, v \models \{\ () \rightarrow exp\ \} \Rightarrow v'} \tag{139}$$

$$\forall i.\ 0 < i \leq n,\ var_i \notin \left(\bigcup_{j=1}^{n} fv(pat_j) \cup fv(exp)\right)$$

$$\frac{E, v \models \left\{ \begin{array}{l} (var_1, \ldots, var_n) \rightarrow \\ \quad \textbf{case}\ var_1\ \textbf{of}\ \{\ pat_1 \rightarrow \ldots \\ \quad \textbf{case}\ var_n\ \textbf{of}\ \{\ pat_n \rightarrow exp\ \} \ldots\ \} \end{array} \right\} \Rightarrow v'}{E, v \models \{\ (pat_1, \ldots, pat_n) \rightarrow exp\ \} \Rightarrow v'} \tag{140}$$

$$\frac{v = \langle\ v_1, \ldots, v_n\ \rangle \qquad E \overset{\rightarrow}{\oplus} \bigcup_{i=1}^{n} \{\ var_i \mapsto v_i\ \} \vdash exp \Rightarrow v'}{E, v \models \{\ (\ var_1, \ldots, var_n\ ) \rightarrow exp\ \} \Rightarrow v'} \tag{141}$$

C.6.1 Exception Handler Matches

Rules 142–143 match against sequences of exception handlers.

$$\frac{E, v \models handler \Rightarrow \langle\ \rangle \qquad E, v \models handlers \Rightarrow v'}{E, v \models handler \mid handlers \Rightarrow v'} \tag{142}$$

$$\frac{E, v \models handler \Rightarrow v'}{E, v \models handler \mid handlers \Rightarrow v'} \tag{143}$$

Finally, rules 144–145 handle matches against individual exceptions, either success or failure.

$$\frac{v = \langle\ exnid,\ v'\ \rangle \qquad E, v' \models pat \Rightarrow v''}{E, v \models exnid\ pat \rightarrow exp \Rightarrow v''} \tag{144}$$

$$\frac{v = \langle\ exnid',\ v'\ \rangle}{E, v \models exnid\ pat \rightarrow exp \Rightarrow \langle\ \rangle} \tag{145}$$

Rule (133) simplifies matches into matches of the form { pat → exp | var → exp' }.

$$\forall i.\ 1 \le i \le n.\ \ \mathrm{var}_i \notin (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_j) \cup \mathrm{fv}(\mathrm{exp}_i))$$

$$\frac{\mathrm{E}, \mathrm{v} \models \left\{ \begin{array}{l} \mathrm{pat}_1 \ \rightarrow \mathrm{exp}_1 \\ \mid \mathrm{var}_1 \ \rightarrow \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{ \\ \quad \mathrm{pat}_2 \ \rightarrow \mathrm{exp}_2 \\ \quad \mid \mathrm{var}_2 \ \rightarrow \texttt{case}\ \mathrm{var}_2\ \texttt{of}\ \{ \\ \quad \ldots \\ \quad \mid \mathrm{var}_{n-1} \rightarrow \texttt{case}\ \mathrm{var}_{n-1}\ \texttt{of}\ \{\ \mathrm{pat}_n \ \rightarrow \mathrm{exp}_n\ \} \ldots \ \} \ \} \end{array} \right\} \Rightarrow \mathrm{v}'}{\mathrm{E}, \mathrm{v} \models \{\ \mathrm{pat}_1 \ \rightarrow \mathrm{exp}_1 \mid \ldots \mid \mathrm{pat}_n \ \rightarrow \mathrm{exp}_n\ \} \Rightarrow \mathrm{v}'} \quad n \ge 1 \tag{133}$$

Rules (134)–(135) define the semantics of wildcard and variable matches.

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v}'}{\mathrm{E}, \mathrm{v} \models \{\ \_ \ \rightarrow \mathrm{exp}\ \} \Rightarrow \mathrm{v}'} \tag{134}$$

$$\frac{\mathrm{E} \overset{\rightarrow}{\oplus} \{\ \mathrm{var} \ \mapsto \mathrm{v}\ \} \vdash \mathrm{exp} \Rightarrow \mathrm{v}'}{\mathrm{E}, \mathrm{v} \models \{\ \mathrm{var} \ \rightarrow \mathrm{exp}\ \} \Rightarrow \mathrm{v}'} \tag{135}$$

Rules (136)–(140) define the semantics of matches against constructor patterns. Rules (136) and (140) are simplification rules, simplifying general constructor matches and tuple matches, respectively; the remaining rules define the matching semantics. The simplification rules are used to simplify deep pattern matches (such as [1,2]) into single-level matches.

$$\forall i.\ 1 \le i \le n,\ \ \mathrm{var}_i \notin (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_j) \cup \mathrm{fv}(\mathrm{exp}))$$

$$\frac{\mathrm{E}, \mathrm{v} \models \left\{ \begin{array}{l} \mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n \ \rightarrow \\ \quad \texttt{case}\ \mathrm{var}_1\ \texttt{of}\ \{\ \mathrm{pat}_1 \ \rightarrow \ldots \\ \quad \texttt{case}\ \mathrm{var}_n\ \texttt{of}\ \{\ \mathrm{pat}_n \ \rightarrow \mathrm{exp}\ \} \ldots \ \} \end{array} \right\} \Rightarrow \mathrm{v}'}{\mathrm{E}, \mathrm{v} \models \{\ \mathrm{con}\ \mathrm{pat}_1 \ldots \mathrm{pat}_n \ \rightarrow \mathrm{exp}\ \} \Rightarrow \mathrm{v}'} \tag{136}$$

$$\frac{\mathrm{v} = \mathrm{con} < \mathrm{v}_1, \ldots, \mathrm{v}_n > \qquad \mathrm{E} \overset{\rightarrow}{\oplus} \{\ \forall i.\ 1 \le i \le n,\ \ \mathrm{var}_i \ \mapsto \mathrm{v}_i\ \} \vdash \mathrm{exp} \Rightarrow \mathrm{v}'}{\mathrm{E}, \mathrm{v} \models \{\ \mathrm{con}\ \mathrm{var}_1 \ldots \mathrm{var}_n \ \rightarrow \mathrm{exp}\ \} \Rightarrow \mathrm{v}'} \tag{137}$$

$$\frac{\mathrm{v} \ne \mathrm{con}\ (\ \mathrm{v}_1,\ \ldots,\ \mathrm{v}_n\ )}{\mathrm{E}, \mathrm{v} \models \{\ \mathrm{con}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n \ \rightarrow \mathrm{exp}\ \} \Rightarrow \mathit{FAIL}} \tag{138}$$

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

The Hume language semantics tools comprise:

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

The abstract machine tools comprise:

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

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

1.4 The Hume Research Programme

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

Our immediate priorities are:

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

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

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

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

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

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

1.4.1 Status of the Research

The Hume research programme is an ongoing process, and the language design is still slowly evolving. The majority of the design has now been fixed and tested, however, and we are making progress on isolating the remaining research issues. The initial design decisions have generally proved to be robust ones and we have thus made fairly rapid progress on our research agenda.

As of March 2004, we have constructed the core Hume language definition, a reference interpreter and a set of reference programs. We have provided a static semantics for types and an axiomatic dynamic semantics (both included in this definition), plus an operational semantics reflecting stack and heap costs [?]. We have developed new theoretical cost models for recursive function definitions, and used these to derive an analysis capable of determining stack and heap costs for recursive Hume programs. This analysis is being validated for soundness against our operational semantics.

We have also constructed a prototype Hume abstract machine (the **pham**), with an associated abstract machine compiler and runtime implementation, including some runtime profiling information. The Hume abstract machine implementation runs on a number of systems including the Real-Time operating system RT-Linux. It provides guaranteed hard space bounds for the FSM-Hume subset of Hume [?], and has vastly superior time performance to embedded Java implementations.

We have developed a diagrammer for Hume programs, which is written in Java for portability, and which is being extended to a full integrated development environment (IDE).

Finally, we are developing a number of ~~number of~~ new realistic control and embedded applications to demonstrate the use of Hume.

The reference interpreter and abstract machine implementation have been developed in tandem and cover the key points of the language design, including all expression forms, coordination, exceptions, timeouts. At present, however, not all types are supported (notable exclusions are unicode characters, fixed-precision and exact arithmetic), type views have not been defined, and we are still clarifying issues related to interrupt handling and low-level I/O.

Our next priorities are to refine our cost models and analyses, to extend our work to cover time as well as space, and to consider machine code implementations.

1.4.2 Foundations for Bounded Space/Time Behaviour

The Hume design builds on foundational research in cost modelling newly developed at St Andrews University. We have developed new theoretical models that...

1.5 Related Work

1.5.1 Embedded Systems

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

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

There has been recent interest in using variants of Java as the basis for embedded systems, though to our knowledge there is as yet no specifically safety-critical design. Two interesting variants are Embedded Java [56] and RTJava [?], for soft real-time applications. Like Hume, both lan-

C.6 Dynamic Semantics: Matches

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

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

Rules (129–130) define sequences of matches. The first rule applies when the first match in a sequence succeeds, the second when it fails. Failure of the last match in a sequence is as defined by the specific case below, e.g. in the rule for non-matching constructors (Rule 138). Since Hume requires matches to be complete, this will never occur in practice, however. The rules return a new list of matches, with the matched rule at the end. This new list would be used to ensure fair matching on subsequent uses of the box. Rule (131) is used to ensure that the final rule in a sequence is returned if it matches, and to avoid tedious repetition in the individual cases.

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

$$\frac{\mathrm{E, v} \models \{\ \mathrm{match}\ \} \Rightarrow \mathrm{v'}, \mathrm{matches'}}{\mathrm{E, v} \models \{\ \mathrm{match} \mid \mathrm{matches}\ \} \Rightarrow \mathrm{v'}, \{\ \mathrm{matches} \mid \mathrm{match}\ \}} \tag{129}$$

$$\frac{\mathrm{E, v} \models \{\ \mathrm{match}\ \} \Rightarrow FAIL \qquad \mathrm{E, v} \models \{\ \mathrm{matches}\ \} \Rightarrow \mathrm{v}, \{\ \mathrm{matches'}\ \}}{\mathrm{E, v} \models \{\ \mathrm{match} \mid \mathrm{matches}\ \} \Rightarrow \mathrm{v}, \{\ \mathrm{match} \mid \mathrm{matches'}\ \}} \tag{130}$$

$$\frac{\mathrm{E, v} \models \{\ \mathrm{match}\ \} \Rightarrow \mathrm{v'}}{\mathrm{E, v} \models \{\ \mathrm{match}\ \} \Rightarrow \mathrm{v'}, \{\ \mathrm{match}\ \}} \tag{131}$$

Rule (132) simplifies multi-argument matches to single-argument matches.

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

$$\frac{\forall i.\ 0 < i \leq m,\quad \mathrm{var}_i \notin (\bigcup_{j=1}^{n} \mathrm{fv}(\mathrm{pat}_{ij}) \cup \mathrm{fv}(\mathrm{exp}_i)) \qquad \mathrm{E, v} \models \left\{ \begin{array}{l} (\mathrm{var}_1, \ldots, \mathrm{var}_n) \rightarrow \\ \quad \mathsf{case}\ (\mathrm{var}_1, \ldots, \mathrm{var}_n)\ \mathsf{of} \\ \qquad \{\ (\mathrm{pat}_{11}, \ldots, \mathrm{pat}_{1n}) \rightarrow \mathrm{exp}_1 \\ \qquad\quad \mid \ldots \\ \qquad\quad \mid (\mathrm{pat}_{m1}, \ldots, \mathrm{pat}_{mn}) \rightarrow \mathrm{exp}_m\ \} \end{array} \right\} \Rightarrow \mathrm{v'}}{\mathrm{E, v} \models \{\ \mathrm{pat}_{11} \ldots \mathrm{pat}_{1n} \rightarrow \mathrm{exp}_1 \mid \ldots \mid \mathrm{pat}_{m1} \ldots \mathrm{pat}_{mn} \rightarrow \mathrm{exp}_m\ \} \Rightarrow \mathrm{v'}} \quad m \geq 1, n \geq 2 \tag{132}$$

The final expression rules are used in constructing matches for case-expressions and function applications. If the expression to be matched is an exception, then the result of the match is an exception; otherwise the matching rules defined below are used. Since fair matching is never used for case-expressions, the reordered match list is discarded.

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

$$\frac{\vdash v \Rightarrow v'}{E, v \vdash \text{match} \Rightarrow v'} \quad v' \in \text{Exn} \tag{127}$$

$$\frac{\vdash v \Rightarrow () \qquad E, v \vdash \text{match} \Rightarrow v', \text{match}'}{E, v \vdash \text{match} \Rightarrow v'} \tag{128}$$

guages support dynamic memory allocation with automatic garbage collection and provide strong exception handling mechanism. The primary differences from Hume are the omission of arbitrary recursion, an absence of formal design principles, the use of a single-layered approach in which coordination is merged with computation, and of course the use of an object-oriented expression language rather than one that is purely functional. We believe that the design choices made here are more suitable for applications where safety or correctness are important. For example, the use of purely functional rather than dynamically-linked object-oriented design allows straightforward static reasoning about the meaning of programs, at the cost of convenience in modifying a running system.

1.5.2 Real-Time Safety-Critical Systems

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

Specification Languages. Safety-critical systems have strong time-based correctness requirements, which can be expressed formally as properties of *safety, liveness* and *timeliness* [7]. Formal requirements specifications are expressed using notations such as temporal logics (e.g. XCTL [16] or MTL [27]), non-temproal logics (e.g. RTL [23]), or timed process algebras (e.g. LOTOS-T [39], Timed CCS [61] or Timed CSP [50]). Such notations are deliberately non-deterministic in order to allow alternative implementations, and may similarly leave some or all timing issues unspecified. It is essential to crystallise these factors amongst others when producing a working implementation.

Non-Determinism. Although non-determinism may be required in specification languages such as LOTOS [20], it is usually undesirable in implementation languages such as Hume, where predictable and repeatable behaviour is required [7]. Hume thus incorporates deterministic processes, but with the option of fair choice to allow the definition of alternative acceptable outcomes. Because of the emphasis on hard real-time, it is not possible to use the event synchronising approach based on *delayed timestamps* which has been adopted by e.g. the concurrent functional language BRISK [17]. The advantage of the BRISK approach is in ensuring strong determinism without requiring explicit specifications of time constraints as in Hume.

Synchronicity. *Synchronous* languages such as Signal [5], Lustre [10], Esterel [8, 6] or the visual formalism Statecharts [15] obey the *synchrony hypothesis*: they assume that all events occur instantaneously, with no passage of time between the occurrence of consecutive events [4]. In contrast, *asynchronous* languages, such as the extended finite state machine languages Estelle [21] and SDL [22], make no such assumption. Hume uses an asynchronous approach, for reasons of both expressiveness and realism. Like Estelle and SDL, it also employs an asynchronous model of communication and supports asynchronous execution of concurrent processes.

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

semantics which collectively fully prescribe the outcome of a process instance given the inputs that have been provided.

Dynamic Process Networks. The initial Hume design uses a static process network, as with Petri net approaches,but unlike recent innovations such as π-calculus [40]. This simplifies the formal language semantics, and very importantly, allows the total cost to be specified for the active process network, but does prevent the direct definition of e.g. mobile processes. We do anticipate that some forms of dynamic process could be supported without destroying our overall cost semantics, but have not yet explored this issue.

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

1.5.3 Other Models Enforcing Bounded Time/Space Properties

Other than our own work, we are aware of three main studies of formally bounded time and space behaviour in a functional setting [9, 19, 60].

Embedded ML. In their recent proposal for Embedded ML, Hughes and Pareto [19] have combined the earlier *sized type system* [18] with the notion of *region types* [57] to give bounded space and termination for a first-order strict functional language [19]. Their language is more restricted than Hume in a number of ways: most notably in not supporting higher-order functions, and in requiring programmer-specified memory usage.

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

Elementary Strong Functional Programming. Turner's *elementary strong functional programming* [60] has similarly explored issues of guaranteed termination in a purely functional programming language. Turner's approach separates finite data structures such as tuples from potentially infinite structures such as streams. This allows the definition of functions that are guaranteed to be primitive recursive. In contrast with the Hume expression layer, it is necessary to identify functions that may be more generally recursive. We will draw on Turner's experiences in developing our termination analysis.

Other Related Work. Recent research by Kamareddine and Monin has formalised automatic proofs of termination of recursive functions, by augmenting proof trees with measures that establish an appropriate decreasing property [24]. They have also investigated widening the scope of automatic termination proof from inductive to non-inductive cases [25].

The next expression rules define the semantics of bracketed expressions in terms of the enclosed expression. Profiling and verification expressions are evaluated purely for their effect, and brackets are ignored, as usual.

$$\frac{E \vdash \mathtt{exp} \Rightarrow v}{E \vdash \mathtt{profile\ exp\ )} \Rightarrow v} \tag{118}$$

$$\frac{E \vdash \mathtt{exp} \Rightarrow v}{E \vdash \mathtt{verify\ exp\ )} \Rightarrow v} \tag{119}$$

$$\frac{E \vdash \mathtt{exp} \Rightarrow v}{E \vdash \mathtt{(\ exp\ )} \Rightarrow v} \tag{120}$$

Rules 121–126 extract exceptions from constructed values such as lists or tuples. The rules are applied to a value that is being matched in order to ensure that any exception that is embedded within the matched value is raised as a result of a match. If there are multiple exceptions, then the rightmost-outermost is returned. – the first such exception working from right-to-left is used the value of the constructed item. If there is no exception, this is signalled by the value $()$.

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

$$\frac{}{\vdash v \Rightarrow v}\ v \in \mathrm{Exn} \tag{121}$$

$$\frac{}{\vdash (\ v_1 \ldots, v_n\ ) \Rightarrow v_n}\ v_n \in \mathrm{Exn} \tag{122}$$

$$\frac{\vdash (\ v_1, \ \ldots, \ v_{n-1}\ ) \Rightarrow v'}{\vdash (\ v_1, \ \ldots, \ v_n\ ) \Rightarrow v'}\ v_n \notin \mathrm{Exn} \tag{123}$$

$$\frac{\vdash (\ v_1, \ \ldots, \ v_{n-1}\ ) \Rightarrow v'}{\vdash \mathrm{con}\ v_1 \ \ldots v_n \Rightarrow v'}\ v_n \notin \mathrm{Exn} \tag{124}$$

$$\frac{}{\vdash () \Rightarrow ()} \tag{125}$$

$$\frac{}{\vdash v \Rightarrow ()}\ v \in \mathrm{BasVal} \tag{126}$$

Type signatures have no dynamic component.

$$\frac{E \vdash \text{exp} \Rightarrow v}{E \vdash \text{exp} :: \text{type} \Rightarrow v} \tag{111}$$

The semantics of type coercion is defined in terms of an auxiliary *coerce* function that implements the semantics of coercion as defined in Section 2.1.3. This function is not specified here.

$$\frac{E \vdash \text{exp}' \Rightarrow v \qquad \textit{coerce}(v, \text{type}) = v'}{E \vdash \text{exp} \ \textbf{as} \ \text{type} \Rightarrow v'} \tag{112}$$

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

$$\frac{E \vdash \text{exp} \Rightarrow v}{E \vdash \texttt{raise} \ \text{exnid exp} \Rightarrow \langle \ \text{exnid}, \ v \ \rangle} \tag{113}$$

The next set of rules define the semantics of constrained expressions. If the cost of evaluating the expression (as given by the cost function) is greater than the specified constant value, then the corresponding exception is raised, otherwise the value of the within-expression is the same as the encapsulated expression.

$$\frac{\begin{array}{c} E \vdash \text{exp}_2 \Rightarrow t \qquad E \vdash \text{exp}_3 \Rightarrow h \qquad E \vdash \text{exp}_4 \Rightarrow s \\ \textit{timecost}\ (E, \ \text{exp}_1\ ) < t \qquad \textit{heapcost}\ (E, \ \text{exp}_1\ ) < h \qquad \textit{stackcost}\ (E, \ \text{exp}_1\ ) < s \\ E \vdash \text{exp}_1 \Rightarrow v \end{array}}{E \vdash \text{exp}_1 \ \texttt{within} \ \text{exp}_2 \ , \ \text{exp}_3 \ (\text{exp}_4\ ) \ [\ \texttt{raise} \ \text{exnid}\ ] \Rightarrow v} \tag{114}$$

$$\frac{\begin{array}{c} E \vdash \text{exp}_2 \Rightarrow t \\ \textit{timecost}\ (E, \ \text{exp}_1\ ) \geq t \end{array}}{E \vdash \text{exp}_1 \ \texttt{within} \ \text{exp}_2 \ , \ \text{exp}_3 \ (\text{exp}_4\ ) \ [\ \texttt{raise} \ \text{exnid}\ ] \Rightarrow \langle \ \text{exnid/Timeout}, \ \langle\ \rangle\ \rangle} \tag{115}$$

$$\frac{\begin{array}{c} E \vdash \text{exp}_2 \Rightarrow t \qquad E \vdash \text{exp}_3 \Rightarrow h \\ \textit{timecost}\ (E, \ \text{exp}_1\ ) < t \qquad \textit{heapcost}\ (E, \ \text{exp}_1\ ) \geq h \end{array}}{E \vdash \text{exp}_1 \ \texttt{within} \ \text{exp}_2 \ , \ \text{exp}_3 \ (\text{exp}_4\ ) \ [\ \texttt{raise} \ \text{exnid}\ ] \Rightarrow \langle \ \text{exnid/HeapOverflow}, \ \langle\ \rangle\ \rangle} \tag{116}$$

$$\frac{\begin{array}{c} E \vdash \text{exp}_2 \Rightarrow t \qquad E \vdash \text{exp}_3 \Rightarrow h \qquad E \vdash \text{exp}_4 \Rightarrow s \\ \textit{timecost}\ (E, \ \text{exp}_1\ ) < t \qquad \textit{heapcost}\ (E, \ \text{exp}_1\ ) < h \qquad \textit{stackcost}\ (E, \ \text{exp}_1\ ) \geq s \end{array}}{E \vdash \text{exp}_1 \ \texttt{within} \ \text{exp}_2 \ , \ \text{exp}_3 \ (\text{exp}_4\ ) \ [\ \texttt{raise} \ \text{exnid}\ ] \Rightarrow \langle \ \text{exnid/StackOverflow}, \ \langle\ \rangle\ \rangle} \tag{117}$$

Also relevant to the problem of bounding time costs is recent work on *cost calculi* [51, 53] and *cost modelling* [49, 31, 54], which has so far been primarily applied to parallel computing.

1.6 Papers and Documentation

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

1.7 Changes from Version 1.0

The main changes introduced in version 1.1 of the report are:

- vectors are now specified with a size rather than a bound;
- strings were never a synonym for `[ char ]`;
- corrections to the basic operations;
- corrections to the syntax.

1.8 Changes from Version 0.2

The main changes introduced in version 1.0 of the report are:

- added `interrupt`, `fifo`, `memory` and `operation`;
- added foreign function interfacing and `operation`;
- added `profile` and `verify` expressions;
- added descriptions of declarations;
- extended `within` expressions to space as well as time;
- included timeouts on I/O descriptors;
- removed `port`, `stream` and `bandwidth` types.

In addition, a number of issues were clarified, errors corrected, and obsolete design decisions eliminated.

Chapter 2

Hume Overview

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

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

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

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

2.1 Types

2.1.1 Base Types

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

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

2.1.2 Structured types

The Hume structured types are shown in Table 2.3, with the corresponding operations shown in Table 2.4. Streams and ports may only be associated with boxes — see below.

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

$$\frac{E \vdash (:)\ \mathrm{exp}_1\ (\ldots((:)\ \mathrm{exp}_n\ [\,])\ldots) \Rightarrow v}{E \vdash [\,\mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\,] \Rightarrow v}\ n \geq 1 \tag{102}$$

$$\frac{}{E \vdash [\,] \Rightarrow \mathtt{Nil}\ ()} \tag{103}$$

$$\frac{}{E \vdash () \Rightarrow ()} \tag{104}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ E \vdash \mathrm{exp}_i \Rightarrow v_i \qquad \vdash (\,v_1,\ \ldots,\ v_n\,) \Rightarrow v' \qquad v' \notin \mathrm{Exn}}{E \vdash (\,\mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\,) \Rightarrow (\,v_1,\ \ldots,\ v_n\,)} \tag{105}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ E \vdash \mathrm{exp}_i \Rightarrow v_i \qquad \vdash (\,v_1,\ \ldots,\ v_n\,) \Rightarrow v' \qquad v' \in \mathrm{Exn}}{E \vdash (\,\mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\,) \Rightarrow v'} \tag{106}$$

$$\frac{E \vdash (\,\mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\,) \Rightarrow v}{E \vdash \mathtt{<<}\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ \mathtt{>>} \Rightarrow (\,v_1,\ \ldots,\ v_n\,)} \tag{107}$$

The semantics of case-expressions is defined by matching the value of the expression against the match. Note that the semantics for conditional expressions (rule 109) is defined in terms of the semantics for case-expressions (rule 108).

$$\frac{E \vdash \mathrm{exp} \Rightarrow v \qquad E, v \vdash \mathrm{match} \Rightarrow v'}{E \vdash \mathtt{case}\ \mathrm{exp}\ \mathtt{of}\ \mathrm{match} \Rightarrow v'} \tag{108}$$

$$\frac{E \vdash \mathtt{case}\ \mathrm{exp}_1\ \mathtt{of}\ \{\ \mathrm{True} \rightarrow \mathrm{exp}_2 \mid \mathrm{False} \rightarrow \mathrm{exp}_3\ \} \Rightarrow v}{E \vdash \mathtt{if}\ \mathrm{exp}_1\ \mathtt{then}\ \mathrm{exp}_2\ \mathtt{else}\ \mathrm{exp}_3 \Rightarrow v} \tag{109}$$

Let-expressions have a simple semantics.

$$\frac{E \vdash \mathrm{decls} \Rightarrow E' \qquad E \overset{\rightarrow}{\oplus} E' \vdash \mathrm{exp} \Rightarrow v}{E \vdash \mathtt{let}\ \mathrm{decls}\ \mathtt{in}\ \mathrm{exp} \Rightarrow v} \tag{110}$$

C.5 Dynamic Semantics: Expressions

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

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

$$\frac{E\ (\text{var}) = v}{E \vdash \text{var} \Rightarrow v} \tag{94}$$

$$\frac{E_0\ (b) = \text{exp} \qquad E \vdash \text{exp} \Rightarrow v}{E \vdash b \Rightarrow v} \tag{95}$$

$$\frac{}{E \vdash \text{con} \Rightarrow \text{con}\ ()} \tag{96}$$

$$\frac{}{E \vdash \text{char} \Rightarrow \text{char}} \tag{97}$$

$$\frac{\mathit{val}\ (\text{string}) = v}{E \vdash \text{string} \Rightarrow v} \tag{98}$$

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

$$\frac{E\ (\text{var}) = \text{matches} \qquad \forall i.\ 1 \leq i \leq n,\ E \vdash \text{exp}_i \Rightarrow v_i \qquad E_0\ (v_1, \ldots, v_n) \vdash \text{matches} \Rightarrow v'}{E \vdash \text{var exp}_1 \ \ldots\ \text{exp}_n \Rightarrow v'} \tag{99}$$

Rules 100–101 deal with constructors by interpreting their arguments as a tuple. If the value of the tuple is an exception, then this is the value of the expression; otherwise the value of the expression is constructed as the combination of the constructor and the semantic tuple of arguments.

$$\frac{E \vdash (\text{exp}_1 \ldots, \text{exp}_n) \Rightarrow v}{E \vdash \text{con exp}_1 \ \ldots\ \text{exp}_n \Rightarrow \text{con}\ v} \quad v \notin \text{Exn} \tag{100}$$

$$\frac{E \vdash (\text{exp}_1 \ldots, \text{exp}_n) \Rightarrow v}{E \vdash \text{con exp}_1 \ \ldots\ \text{exp}_n \Rightarrow v} \quad v \in \text{Exn} \tag{101}$$

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

Table 2.1: Hume base types

`bool`	`&& ||`

`int`
`nat`
	`+ - * div mod`
	unary `-` — not provided for `nat`
	`**` — power
	`< <= == >= > !=`

`float`
`fixed`
`exact`
	`+ - * /`
	unary `-`
	`sin cos tan asin acos atan`
	`sinh cosh tanh atan2`
	`log10 ln exp`
	`sqrt`
	`**` — power
	`< <= == >= > !=`

`word`	`lshl lshr` — logical shift left/right `rotl rotr` — rotate left/right `& | ^ ~` — bitwise operations `< <= == >= > !=`
`char` `unicode`	`< <= == >= > !=`

Table 2.2: Basic operations on base types

vector	fixed length sequence of uniform type with the given bounds type: `vector <size> of <type>` where `<size>` ≥ 0 denoted by: `<< <expr1>, ... , <exprN> >>` where `N` ≥ 0
tuple	fixed length sequence of mixed type type: `( <type1>, ... , <typeN> )` where `N` ≥ 0 denoted by: `( <expr1>, ... , <exprN> )` where `N` = 0 or `N` > 1
list	variable length sequence of uniform type type: `[ <type> ]` denoted by: `[ <expr1>, ... , <exprN> ]` where `N` ≥ 0
string	variable or fixed length sequence of characters type: `string {<size>} ]` denoted by: `"<printable1> ... <printableN>"` where `N` ≥ 0
discriminated union	declared by: `data <id> = <id1> <type11> ... <type1N> | ...` where `N` ≥ 0 type: `<id> <type1> ... <typeN>` where `N` ≥ 0 denoted by: `<id> <expr1> ... <exprN>` where `N` ≥ 0
time	type: `time`

Table 2.3: Structured types

`vector`	construction by denotation
	selection by pattern matching
	`@ <expr>` — select `<expr>`th element
	`length`
	`vecdef, vecmap, vecfoldr`
	`update` — copying update
	`++` — vector concatenation
	`< <= == > >= != e`

Tuples	construction by denotation `@ <expr>` — select `<expr>`th element selection by pattern matching `< <= == > >= != e`

Lists	`:` — list constructor
	construction by denotation
	`length`
	`hd tl`
	`@ <expr>` — select `<expr>`th element
	selection by pattern matching
	`++` — list concatenation
	`== != e`

String	construction by denotation
	`length`
	`@ <expr>` — select `<expr>`th element
	`++` — string concatenation
	`== != e`

Table 2.4: Basic Operations on Structured types

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

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

$$\frac{\begin{array}{c} \mathrm{W}\ (\mathrm{boxid}) = \langle\ \mathrm{wins}, \mathrm{wouts}\ \rangle \\ n = |\mathrm{wins}| \\ \mathrm{SE} = \mathit{SE\ of\ E} \\ \mathrm{vs} = \langle\ \mathit{snd}(\mathit{SE}\ (\mathrm{wins}_1)),\ \ldots,\ \mathit{snd}(\mathit{SE}\ (\mathrm{wins}_n))\ \rangle \\ \mathrm{E}, \mathrm{vs} \vdash \mathrm{body} \Rightarrow \mathrm{vs}' \\ \mathrm{SE}^I = \{\ \forall i.\ 1 \leq i \leq n,\ \mathrm{wins}_i \mapsto \langle\ \mathit{isport}\ \mathrm{wins}_i, \mathit{tl}\ \mathrm{vs}_i\ \rangle\ \} \\ \mathrm{SE}^O = \{\ \forall i.\ 1 \leq i \leq |\mathrm{wouts}|,\ \mathrm{wouts}_i \mapsto \langle\ \mathit{true}, [\mathrm{vs}'_i]\ \rangle\ \} \end{array}}{\mathrm{E}, \mathrm{W} \vdash \langle\ \mathrm{boxid}, \mathrm{ins}, \mathrm{outs}, \mathrm{body}\ \rangle \Rightarrow \mathrm{vs}', \mathrm{SE}^I, \mathrm{SE}^O} \qquad (90)$$

The final set of process rules define the semantics of executing a single box body. There are three cases, corresponding to normal execution, an exception or a timeout respectively. In order to implement fair matching, the new rule ordering returned by the match rule should update the definition of the matches for the box in the environment. In this way, each successful fair match will change the rule ordering, thereby ensuring that each rule is matched equally, as required by the semantics.

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

$$\frac{\mathrm{E} \vdash \mathrm{time} \Rightarrow \mathrm{t} \qquad \mathit{timecost}\ (\ \mathrm{E}, \mathrm{matches}\ (\mathrm{vs})\ ) < \mathrm{t} \qquad \mathrm{E}, \mathrm{vs} \models \mathrm{matches} \Rightarrow \mathrm{v}, \mathrm{matches}'}{\vdash \mathrm{E}, \mathrm{vs} \Rightarrow \texttt{timeout}\ \mathrm{time}\ \mathrm{matches}\ \texttt{handle}\ \mathrm{handlesv}} \quad v \notin \mathrm{Exn} \qquad (91)$$

$$\frac{\begin{array}{c} \mathrm{E} \vdash \mathrm{time} \Rightarrow \mathrm{t} \qquad \mathit{timecost}\ (\ \mathrm{E}, \mathrm{matches}\ (\mathrm{vs})\ ) < \mathrm{t} \qquad \mathrm{E}, \mathrm{vs} \vdash \mathrm{matches} \Rightarrow \mathrm{v} \\ \mathrm{E}, \mathrm{v} \models \mathrm{handle} \Rightarrow \mathrm{v}', \mathrm{matches}' \end{array}}{\mathrm{E}, \mathrm{vs} \vdash \texttt{timeout}\ \mathrm{time}\ \mathrm{matches}\ \texttt{handle}\ \mathrm{handles} \Rightarrow \mathrm{v}'} \quad v \in \mathrm{Exn} \qquad (92)$$

$$\frac{\mathrm{E} \vdash \mathrm{time} \Rightarrow \mathrm{t} \qquad \mathit{timecost}\ (\ \mathrm{E}, \mathrm{matches}\ \mathrm{vs}\ ) \geq \mathrm{t} \qquad \mathrm{E}, \langle\ \mathrm{Timeout}, \langle\ \rangle\ \rangle \models \mathrm{handle} \Rightarrow \mathrm{v}, \mathrm{matches}'}{\mathrm{E}, \mathrm{vs} \vdash \texttt{timeout}\ \mathrm{time}\ \mathrm{matches}\ \texttt{handle}\ \mathrm{handles} \Rightarrow \mathrm{v}} \qquad (93)$$

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

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

$$\frac{E, W \vdash P \Rightarrow I, A \qquad E, W \vdash I, A \Rightarrow E'}{E, W \vdash P \Rightarrow E'} \tag{86}$$

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

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

$$\frac{E, W \vdash I, A \Rightarrow E', \Gamma, A' \qquad E', W \vdash \Gamma, A' \Rightarrow E''}{E, W \vdash I, A \Rightarrow E'', \Gamma, A'} \; A \neq \{\,\} \tag{87}$$

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

$$\frac{}{E, W \vdash I, \{\,\} \Rightarrow E} \tag{88}$$

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

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

$$\forall i.\ 1 \leq i \leq |A|,\ E, W \vdash A_i \Rightarrow \text{outs}_i,\ E_i^I, E_i^O$$

$$E' = \bigcup_{i=1}^{|A|} E_i^I \ \vec{\oplus} \ \bigcup_{i=1}^{|A|} E_i^O$$

$$\frac{E \ \vec{\oplus} \ E', W \vdash I \cup A \Rightarrow \Gamma, A'}{E, W \vdash I, A \Rightarrow (E \ \vec{\oplus} \ E'), \Gamma, A'} \tag{89}$$

Table 2.5: Conformancy between Hume base types.

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

Notes

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

2.1.3 Type Conversions

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

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

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

Coerced structured types:

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

2.1.4 Exceptions

Exceptions are:

- declared by: `exception <id> <type>`, within declarations (Section 2.2);
- raised by: `raise <id> <expr>` within expressions (Section 2.3.9);
- handled by: `handle <handlers>` within boxes (Section 2.4.3).

System exceptions may be handled either within a box or by a general system handler. If a box defines a handler for a system exception, and the exception is raised as a consequence of executing

that box, then the specified handler is called. If a box fails to define a handler for a system exception and that system exception occurs during the process of executing the box, then the general system handler is called. There must be precisely one general handler for each system exception. The system exceptions are:

`DivO`	division by 0
`Overflow/Underflow`	numeric overflow/underflow
`OutOfBounds`	out of bounds vector index
`HeapOverflow`	heap overflow
`StackOverflow`	stack overflow
`Timeout`	timeouts
`EndOfFile`	end of input file

Note that `HeapOverflow` and `StackOverflow` are only raised by code whose heap/stack costs have not been certified, and then in the context of a within-constraint on boxes or expressions. `Timeouts` occur through within/timeout constraints on ports, streams, wires, boxes or expressions.

2.2 The Declaration Layer

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

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

2.2.1 Function, Value and Constant Declarations

Functions and named values are introduced in a similar way to their Haskell counterparts. Values may be declared to be *constant*, in which case their value is calculated at compile-time. Such constant values must be simple calculations and may not be defined using function calls, within constraints etc. Named constants may be used in the wiring metalanguage, in expression macros, or in any other place where a constant expression is mandated. The form `varid :: <type>` indicates only that the variable has the specified type, and must be accompanied by a value or function declaration. It is not an error to omit a type declaration, however: in this case, the variable or function is assigned the most general type possible by the compiler using a standard Damas-Milner type inference algorithm [?].

```
<decl> ::= ...
        | <fundecl>
        | "constant" <varid> "=" <cexpr>

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

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

For example, we can define the ubiquitous `nfib` function as follows:

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

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

$$\frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E}, \mathrm{W} \vdash \mathrm{P}_i \Rightarrow \mathrm{I}_i, \mathrm{A}_i \qquad \mathrm{I} = \bigcup_{i=1}^{n} \mathrm{I}_i \qquad \mathrm{A} = \bigcup_{i=1}^{n} \mathrm{A}_i}{\mathrm{E}, \mathrm{W} \vdash \{\ \mathrm{P}_1, \ldots, \mathrm{P}_n\ \} \Rightarrow \mathrm{I}, \mathrm{A}} \tag{81}$$

Rules 82–85 determine whether individual inputs are available or have timed out.

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

$$\frac{\begin{array}{c}\mathrm{P} = \langle\ \mathrm{boxid}, \mathrm{ins}, \mathrm{outs}, \mathrm{body}\ \rangle \qquad \mathrm{W}\ (\mathrm{boxid}) = \langle\ \mathrm{wins}, \mathrm{wouts}\ \rangle \\ \mathrm{E} \vdash \mathrm{wins} \Rightarrow \mathrm{b}, \mathrm{b}' \qquad \mathrm{I}, \mathrm{A} = \mathit{if}\ b \vee b'\ \mathit{then}\ \{\ \}, \{\ \mathrm{P}\ \}\ \mathit{else}\ \{\ \mathrm{P}\ \}, \{\ \}\end{array}}{\mathrm{E}, \mathrm{W} \vdash \mathrm{P} \Rightarrow \mathrm{I}, \mathrm{A}} \tag{82}$$

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

$$\frac{\mathrm{E} \vdash \mathrm{id}_1 \Rightarrow \mathit{true}, \mathrm{b} \qquad \mathrm{E} \vdash \mathrm{id}_2 \ldots \mathrm{id}_n \Rightarrow \mathrm{b}', \mathrm{b}''}{\mathrm{E} \vdash \mathrm{id}_1 \ldots \mathrm{id}_n \Rightarrow \mathrm{b}', (\mathrm{b} \vee \mathrm{b}'')} \tag{83}$$

$$\frac{\mathrm{E} \vdash \mathrm{id}_1 \Rightarrow \mathit{false}, \mathrm{b} \qquad \mathrm{E} \vdash \mathrm{id}_2 \ldots \mathrm{id}_n \Rightarrow \mathrm{b}', \mathrm{b}''}{\mathrm{E} \vdash \mathrm{id}_1 \ldots \mathrm{id}_n \Rightarrow \mathit{false}, (\mathrm{b} \vee \mathrm{b}'')} \tag{84}$$

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

$$\frac{\mathrm{E}\ (\mathrm{id}) = \langle\ \mathrm{b},\ \mathrm{vs}\ \rangle \qquad \mathrm{b}' = \mathit{if}\ \mathrm{vs} = () \vee \mathit{hd}\ \mathrm{vs} \neq \langle\ \mathrm{Timeout}, \langle\ \rangle\ \rangle\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{true}}{\mathrm{E} \vdash \mathrm{id} \Rightarrow \mathrm{b}, \mathrm{b}'} \tag{85}$$

C.4 Dynamic Semantics: Processes

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

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

$$\frac{\mathrm{E_0} \oplus \mathrm{IE} \vdash \text{decls} \Rightarrow \mathrm{E} \qquad \vdash \text{boxes} \Rightarrow \mathrm{P} \qquad \vdash \text{wires} \Rightarrow \mathrm{W} \qquad ((\mathrm{E_0} \overset{\rightarrow}{\oplus} \mathrm{E}) \oplus \mathrm{IE} \oplus \mathrm{SE}),\ \mathrm{W} \vdash \mathrm{P} \Rightarrow \mathrm{E'}}{\mathrm{SE}, \mathrm{IE} \vdash \texttt{program}\ \text{decls boxes wires} \Rightarrow \mathrm{E'}} \tag{76}$$

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

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

$$\frac{\forall i,\ 1 < i \leq n,\ \vdash \text{box}_i \Rightarrow \mathrm{P}_i}{\vdash \text{box}_1 \ \ldots\ \text{box}_n \Rightarrow \bigcup_{i=1}^{n} \mathrm{P}_i} \tag{77}$$

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

$$\frac{}{\vdash \texttt{box}\ \text{boxid ins outs body} \Rightarrow \{\ \langle\ \text{boxid, ins, outs, body}\ \rangle\ \}} \tag{78}$$

`wiring` declarations are processed to give the wiring layout mapping the outputs of boxes or I/O operations to the inputs of other boxes.

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

$$\frac{\forall i,\ 1 < i \leq n,\ \vdash \text{wire}_i \Rightarrow \mathrm{W}_i}{\vdash \text{wire}_1 \ \ldots\ \text{wire}_n \Rightarrow \bigcup_{i=1}^{n} \mathrm{W}_i} \tag{79}$$

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

$$\frac{\mathrm{W} = \{\ \text{boxid} \mapsto \langle\ \text{sources, dests}\ \rangle\ \}}{\vdash \texttt{wire}\ \text{boxid sources dests} \Rightarrow \mathrm{W}} \tag{80}$$


```
nfib :: int 32 -> int 32;
nfib 0 = 1;
nfib n = 1 + nfib(n-1) + nfib (n-2);
```

and we could define a constant `arraylen`, by, e.g. `constant arraylen = 100`.

2.2.2 Type Declarations

Hume includes two kinds of type declaration, both of which are defined analagously to their Haskell counterparts. The first form introduces a new constructed data type whose alternatives are distinguished by different data constructors (a discriminated union type). The second form introduce a *type synonym*; a named type equivalent to some pre-existing type. Either form of declaration may be *polymorphic*, in which case it must be provided with a number of type variable arguments (these may then appear within the type declaration part). Constructed types may also be defined recursively.

```
<decl> ::= ...
        | "data" <typeid> <varids> "=" <constrs>
        | "type" <typeid> <varids> "=" <type>

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

So, for example, we can define a new type of polymorphic binary trees, and a version that is specialised to 32-bit integers, by:

```
data Tree a  = Leaf a | Node (Tree a) (Tree a);

type IntTree = Tree (int 32);
```

User-defined types (whether data types or type synonyms) may be used wherever pre-defined types may be used, and data constructors may be used both in pattern-matching and in expressions.

2.2.3 Exception Declarations

Hume exceptions are typed. A Hume exception is a constructed value like a data type, which is raised by a `raise` expression or as the result of a system exception, and which is handled by exception handlers introduced at the box level.

```
<decl> ::= ...
        | "exception" <exnid> "::" <exprtype>
```

For example, we can define a new exception over a string, with the corresponding `raise` and `handle` clauses as follows:

```
exception X :: string;

box B in ( ... ) out ( res :: string )
handles X
```

```
match
   ... -> ...  raise (X "overflow") ...
handle
   X s -> s;
```

2.2.4 Import/Export Declarations

Hume import and export declarations respectively introduce identifiers that have been defined in some other module, or expose identifiers from the currently defined module for use elsewhere.

```
<decl> ::= ...
        | "import" <id> <idlist>
        | "export" <idlist>
```

In the import form, `<id>` is the name of the module to be imported.

In both forms, `<idlist>` is the list of entities to be imported or exported.

So, for instance,

```
import M (a,b);
export f;
```

imports `a` and `b` defined in module `M` for use in the current module, and exports `f`.

2.2.5 Foreign Function Interfacing

Hume uses the same notation for foreign function interfacing as Haskell [?]. This allows reuse of standard interface generator tools such as **GreenCard**. External calls are specified using foreign function declarations which provide information about the calling convention to be used, whether the function is safe or unsafe, and the Hume type of the function. The optional string is used to provide additional information to the compiler related to the calling language: for C calls, this includes the name of the function if different from the Hume name plus information about files that must be included in the compiled code. Note that it may be necessary to link compiled Hume code with additional libraries or undertake other special actions as specified by the implementation in order to exploit external calls.

Note that it is not permitted to use unsafe foreign calls in Hume expressions; they may only be used in Hume operations. The safety clause is retained for backwards compatibility with the Haskell FFI. In this way referential transparency is preserved for Hume expressions even in the presence of foreign function calls.

```
<foreigndecl> ::=
          "foreign" "import" <callconv> [ <safety> ] [ <string> ] <id> "::" <exprtype>
        | "foreign" "export" <callconv> [ <string> ] <id> "::" <exprtype>

<safety> ::=
           "safe" | "unsafe"

<callconv> ::=
          "ccall" | "stdcall" | "cplusplus" | "jvm" | "dotnet"
```

Note that in `<foreigndecl>`, `<string>` typically consists of the library name followed by the name of the library entity, and `<id>` is the Hume name for the entity.

$$\frac{}{E \vdash \text{var matches} \Rightarrow \{\ \text{var} \mapsto \text{matches}\ \}} \quad (72)$$

$$\frac{E \vdash \text{op pat}_1\ \text{pat}_2 = \text{exp} \Rightarrow E'}{E \vdash \text{pat}_1\ \text{op pat}_2 = \text{exp} \Rightarrow E'} \quad (73)$$

$$\frac{E \vdash \text{exp} \Rightarrow v}{E \vdash \text{var} = \text{exp} \Rightarrow \{\ \text{var} \mapsto v\ \}} \quad (74)$$

$$\frac{}{E \vdash \text{var} :: \text{type} \Rightarrow \{\ \}} \quad (75)$$

...

C.3 Dynamic Semantics: Declarations

Declarations are processed to generate an initial environment mapping identifiers to initial values. This environment is used in the dynamic semantics for expressions to determine the value of identifers in function applications and variable expressions (rules 99, 94) and in the semantics of boxes to determine the value attached to an I/O object (rule 90). Declarations may be self-recursive or mutually recursive.

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

$$\frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E} \oplus \bigoplus_{j=1}^{n} \mathrm{E'}_j \vdash \mathrm{decl}_i \Rightarrow \mathrm{E'}_i}{\mathrm{E} \vdash \mathrm{decl}_1 \ \ldots\ \mathrm{decl}_n \Rightarrow \mathrm{E} \oplus \bigoplus_{i=1}^{n} \mathrm{E'}_i} \tag{64}$$

Each declaration is processed to produce a corresponding value environment.

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

$$\frac{}{\mathrm{E} \vdash \texttt{importmodid}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n \Rightarrow \{\ \}} \tag{65}$$

$$\frac{}{\mathrm{E} \vdash \texttt{export}\ \mathrm{var}_1\ \ldots\ \mathrm{var}_n \Rightarrow \{\ \}} \tag{66}$$

$$\frac{}{\mathrm{E} \vdash \texttt{foreign import}\ [\ \mathrm{s}\ ]\ [\ \mathrm{c}\ ]\ [\ \mathrm{str}\ ]\ \mathrm{var} :: \mathrm{type} \Rightarrow \{\ \}} \tag{67}$$

$$\frac{}{\mathrm{E} \vdash \texttt{foreign export}\ [\ \mathrm{c}\ ]\ [\ \mathrm{str}\ ]\ \mathrm{var} :: \mathrm{type} \Rightarrow \{\ \}} \tag{68}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \mathrm{v}}{\mathrm{E} \vdash \texttt{constant}\ \mathrm{id} = \mathrm{exp} \Rightarrow \{\ \mathrm{id} \mapsto \mathrm{v}\ \}} \tag{69}$$

$$\frac{(\mathrm{SE}\ \mathit{of}\ \mathrm{E})\ \mathrm{id'} = \mathrm{vs}}{\mathrm{E} \vdash \texttt{port/stream/memory/fifo}\ \mathrm{id}\ \texttt{from}\ \mathrm{id'} \Rightarrow \{\ \mathrm{id} \mapsto \langle\ \texttt{true}, \mathrm{vs}\ \rangle\ \}} \tag{70}$$

$$\frac{\mathrm{E} \vdash \mathrm{cexpr} \Rightarrow \mathrm{v} \qquad (\mathrm{SE}\ \mathit{of}\ \mathrm{E})\ \mathrm{id'} = \mathrm{vs} \qquad \mathrm{vs'} = \langle\ \mathrm{v},\ \mathrm{vs}\ \rangle}{\mathrm{E} \vdash \texttt{port/stream/memory/fifo}\ \mathrm{id}\ \texttt{from}\ \mathrm{id'}\ \texttt{initial}\ \mathrm{cexpr} \Rightarrow \{\ \mathrm{id} \mapsto \langle\ \texttt{true}, \mathrm{vs'}\ \rangle\ \}} \tag{71}$$

For example, `foreign import ccall "math.h sinh" hsin :: float 32 -> float 32` specifies a Hume function `hsin` which is defined as the C function `sinh` in the `math.h` header file.

Operations extend the foreign function interface, providing a wrapper for (possibly) unsafe foreign function calls. An operation introduces a new box with one input, named `inp`, and one output, named `outp`. The string describes the foreign function as for the optional string in a foreign function import declaration. Only the `ccall` calling convention is supported.

```
<decl> ::= ...
   | "operation" <boxid> "as" <string> "::" <type>
```

For example, `operation System as "system" :: String -> ()` introduces a new box called `System` whose purpose is to execute the `system` function call (on a Unix system, this will cause its argument to be executed as an operating system command, for instance). The call is performed synchronously, returning the unit tuple value (`()`) on completion. The `System` box is wired normally, for example,

```
wire b.syscall to System.inp;
wire System.outp to b.done;
```

wires its input to `b.syscall` and its output to `b.done`.

2.2.6 Expression Declarations

Where the result of a Hume program is a single expression rather than a set of boxes, then a special shorthand form is available.

```
<decl> ::= ...
        | "expression" <expr>
```

An expression declaration introduces a box with no input, and whose output is directed to the standard output stream. The box runs precisely once, and may use any defined function or expression form. For example, `expression nfib 100` produces the program whose purpose is to calculate

2.3 The Expression Layer

The Hume expression layer follows the design of widely used functional languages such as Standard ML [41] and Haskell [?]. Like Standard ML, Hume expressions follow a strict evaluation order. This allows tight cost functions to be derived for Hume expressions and allows a relatively simple semantics of exceptions to be specified (see Appendix C). Like Haskell, the Hume expression layer is purely functional. In order to simplify code reuse, the syntax of the Hume expression layer is broadly based on that of Haskell, and is fully described in Appendix A. The formal dynamic semantics of the Hume expression layer is given in Appendix C.

2.3.1 Constants

Constants are simple constant values covering the basic Hume types. Characters are 1-byte ASCII characters conforming to ISO-Latin-1 and have type `char`. Multi-byte characters may formed using hex constants, and have type `unicode`. So, for example, `'x22A5'` is the Unicode character representing the mathematical symbol ⊥.

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

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

2.3.2 Variables

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

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

2.3.3 Constructors

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

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

2.3.4 Tuples, Lists and Vectors

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

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

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

2.3.5 Function Applications

Hume function applications have a strict semantics. All arguments to a function are evaluated from right-to-left before the function is called. All functions in Hume must be fully applied:

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

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

Appendix C

Dynamic Semantics

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

C.1 Limitations

There are a number of limitations on the semantics given here. Firstly, we do not consider the semantics of imported values. This can be added straightforwardly by extending the initial value environment with bindings for the imported values. Secondly, the semantics of processes assumes that all active processes are scheduled for precisely one step. The status of all processes is then reassessed to determine whether each process is active or inactive. A more flexible semantics would schedule precisely one active process. This modification should not be too hard to incorporate into the semantics. Thirdly, we need to define the semantics of the *timecost*, *stackcost*, *heapcost* and *coerce* functions which are used to calculate timeouts and type coercions respectively. We anticipate that the semantics of type coercions can be defined without great difficulty. We are in the process of developing formal analyses for providing upper bound on stack and heap usage for Hume programs, including primitive recursion. We anticipate that it will be possible to extend these analyses to cover time using analytical techniques developed for other real-time languages [?, ?]. Fourthly, we have not defined the semantics for interrupts. Clearly a polling semantics is not ideal for such objects, though they may possess a similar semantics to other kinds of I/O operation? Finally, the dynamic semantics is currently defined only for the synchronous language (i.e. omitting fair matches and $*$), and does not consider higher-order functions. We anticipate extending the semantics to cover these constructs in due course.

C.2 Dynamic Semantics: Notation

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

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

Currying and partial applications are not supported. Higher-order functions *are* supported, but may have cost implications as discussed in Section ??.

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

2.3.6 Case Expressions

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

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

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

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

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

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

2.3.7 Local Declarations

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

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

2.3.8 Type Expressions and Type Coercions

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

More powerful dynamic *type coercions* can be specified using "as"-expressions. A table of types that are compatible for coercion purposes is given in Section 2.1.3. In this case, some computation

may be required to coerce a value from one type to another. Unlike the use of the "::" operator, a type coercion may not be reversible: information may be lost during the coercion process, for example. Coercions must therefore be treated with care.

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

2.3.9 Exceptions

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

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

2.3.10 Time and Space Constraints

Within-expressions are used to specify that evaluation of the associated expression must complete within the specified constraint (which must be a constant). Failure to do so causes an exception to be raised with a If not given explicitly, this exception is one of `Timeout`, `StackOverflow` or `HeapOverflow` and must be handled by the box. It is raised with a () argument. If only one space constant is specified, it represents a heap constraint, otherwise the first constraint represents a heap constraint, and the second a stack constraint.

```
<expr> ::= ...
        | <expr> "within" <constraint> [ "raise" <exnid> ]

<constraint> ::=
  timec [ "," spacec [ "(" spacec ")" ] ]
| spacec  [ "(" spacec ")" ] ]
```

2.3.11 Constant Expressions

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

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

2.3.12 Profiling and Verification

Two expression forms are used for profiling and cost verification purposes. `profile e` prints the costs of executing `e`, and returns the value of `e`. `verify e` applies the cost modeller to the expression `e` and checks that the actual costs are within those that are determined. A `StackOverflow` or `HeapOverflow` is raised as appropriate if the inferred costs are not achieved in practice. This is mainly useful to eliminate errors during the development of the cost modelling software.

B.7 Static Semantics: Types

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

$$\frac{(\mathrm{AE}\ of\ \mathrm{E})\ \alpha = n}{\mathrm{E} \vdash \alpha} \tag{59}$$

$$\frac{(\mathrm{TE}\ of\ \mathrm{E})\ \chi = n \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \tau_i}{\mathrm{E} \vdash \chi\ \tau_1\ \ldots\ \tau_n} \tag{60}$$

$$\frac{\mathrm{E} \vdash \tau \Rightarrow \qquad \mathrm{E} \vdash \tau' \Rightarrow}{\mathrm{E} \vdash \tau \rightarrow \tau' \Rightarrow} \tag{61}$$

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

$$\frac{\mathrm{E} \oplus_{AE} \{\ \alpha_1,\ \ldots,\ \alpha_n\ \} \vdash \tau}{\mathrm{E} \vdash \forall\ \alpha_1\ \ldots\ \alpha_n.\ \tau} \tag{62}$$

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

$$\frac{\mathrm{E} \vdash \forall\ \alpha_1\ \ldots\ \alpha_n.\ \tau}{\mathrm{E} \vdash \tau \Rightarrow \forall\ \alpha_1\ \ldots\ \alpha_n.\ \tau} \tag{63}$$

B.8 Static Semantics: The Initial Environment

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

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

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

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

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

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{pat}_i \Rightarrow \tau, \mathrm{VE}_i}{\mathrm{E} \vdash < \mathrm{pat}_1, \ldots, \mathrm{pat}_n > \Rightarrow \mathbf{Vector}\ \tau, \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{52}$$

B.5.1 Static Semantics: Exception Handler Matches

$$\frac{\mathrm{E} \vdash \mathrm{exnid} \Rightarrow \mathbf{Exn}\ \tau \qquad \mathrm{E} \vdash \mathrm{pat} \Rightarrow \tau, \mathrm{VE} \qquad \mathrm{E} \oplus_{VE} \mathrm{VE} \vdash \mathrm{exp} \Rightarrow \tau'}{\mathrm{E} \vdash \mathrm{exnid\ pat} \rightarrow \mathrm{exp} \Rightarrow \tau'} \tag{53}$$

$$\frac{\mathrm{E} \vdash \mathrm{handler} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{handlers} \Rightarrow \tau}{\mathrm{E} \vdash \mathrm{handler} \mid \mathrm{handlers} \Rightarrow \tau} \tag{54}$$

B.6 Static Semantics: Type Expressions

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

$$\frac{(\mathrm{VE}\ \mathit{of}\ \mathrm{E})\ (\mathit{tyvar}) = \alpha}{\mathrm{E} \vdash \mathrm{tyvar} \Rightarrow \alpha} \tag{55}$$

$$\frac{(\mathrm{VE}\ \mathit{of}\ \mathrm{E})\ (\mathit{tycon}) = \forall \alpha_1 \ldots \alpha_n.\chi\ \alpha_1 \ldots \alpha_n \qquad \forall i.\ 1 \leq i \leq n,\ \mathrm{E} \vdash \mathrm{type}_i \Rightarrow \tau_i}{\mathrm{E} \vdash \mathrm{tycon\ type}_1 \ldots \mathrm{type}_n \Rightarrow \chi\ \tau_1 \ldots \tau_n} \tag{56}$$

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

$$\frac{\mathrm{E} \vdash \mathit{type} \Rightarrow \tau}{\mathrm{E} \vdash (\mathrm{type}) \Rightarrow \tau} \tag{58}$$


```
<expr> ::= ...
        | "profile"  <expr>
        | "verify"   <expr>
```

Note that these may not be supported in all implementations.

2.4 The Coordination Layer

This section describes the Hume coordination layer and the wiring metalanguage. The formal dynamic semantics of Hume boxes is given in Appendix C.

2.4.1 Boxes

The Hume unit of coordination is the *box*. A box has a unique name, specified in its *prelude*. A box has *inputs* and *outputs* termed *ins* and *outs*. Ins and outs are fixed width sequences of *inout* type. An *inout* type is any Hume type excluding a function or exception. A box's *ins* and *outs* are specified in its prelude. Each *in* and *out* has a unique name, and is typed. The exceptions a box handles are specified in the box's prelude. It is possible to provide a within-clause to limit costs within a box execution in the same way as they are limited in an expression.

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

<prelude> ::=
        "box" <boxid>
        "in"  <inoutlist>
        "out" <inoutlist>
[ "within"  <constraint> ]
        [ "handles" <exnidlist>  ]

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

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

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

2.4.2 Box Bodies

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

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

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

1. a pattern component `<patt>` which is type consistent with the `in` declaration; and

2. an expression component `<expr>` which is type consistent with the `out` declaration.

Top-level patterns may include *s. The purpose of a * is to indicate that the corresponding input is neither matched nor consumed.

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

2.4.3 Exception Handlers

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

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

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

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

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

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

2.4.4 Wiring

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

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

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

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

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

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

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau}{\mathrm{E} \vdash \texttt{verify}\ \mathrm{exp} \Rightarrow \tau} \tag{43}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau}{\mathrm{E} \vdash (\ \mathrm{exp}\ ) \Rightarrow \tau} \tag{44}$$

B.5 Static Semantics: Matches

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

$$\frac{\mathrm{E} \vdash \{\ \mathrm{match}\ \} \Rightarrow \tau \qquad \mathrm{E} \vdash \{\ \mathrm{matches}\ \} \Rightarrow \tau}{\mathrm{E}, \mathrm{v} \vdash \{\ \mathrm{match} \mid \mathrm{matches}\ \} \Rightarrow \tau} \tag{45}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{match}_i \Rightarrow \tau_i, \mathrm{VE}_i \qquad \mathrm{E} \oplus_{VE} (\bigoplus_{i=1}^{n} \mathrm{VE}_i) \vdash \mathrm{exp} \Rightarrow \tau'}{\mathrm{E} \vdash \{\ \mathrm{pat}_1\ \ldots\ \mathrm{pat}_n\ \rightarrow\ \mathrm{exp}\} \Rightarrow \tau_1 \rightarrow\ \ldots \rightarrow \tau_n \rightarrow \tau'} \tag{46}$$

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

$$\frac{}{\mathrm{E} \vdash \_ \Rightarrow \tau, \{\ \}} \tag{47}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{var} \Rightarrow \tau, \{\ \mathrm{var} \mapsto \tau\ \}} \tag{48}$$

$$\frac{\mathrm{E} \vdash \mathrm{con} \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau' \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{pat}_i \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{E} \vdash \mathrm{con}\ \mathrm{pat}_1\ \ldots\ \mathrm{pat}_n \Rightarrow \tau', \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{49}$$

$$\frac{}{\mathrm{E} \vdash () \Rightarrow \mathbf{Tuple}_0, \{\ \}} \tag{50}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{pat}_i \Rightarrow \tau_i, \mathrm{VE}_i}{\mathrm{E} \vdash (\ \mathrm{pat}_1,\ \ldots,\ \mathrm{pat}_n\ ) \Rightarrow \mathbf{Tuple}_n\ \tau_1\ \ldots\ \tau_n, \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{51}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \tau_i}{\mathrm{E} \vdash (\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ) \Rightarrow \mathbf{Tuple}_n\ \tau_1\ \ldots\ \tau_n} \tag{32}$$

$$\frac{}{\mathrm{E} \vdash \texttt{<<}\ \ \texttt{>>} \Rightarrow \mathbf{Vector}\ \tau} \tag{33}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \tau}{\mathrm{E} \vdash \texttt{<<}\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ \texttt{>>} \Rightarrow \mathbf{Vector}\ \tau} \tag{34}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{match} \Rightarrow \tau \rightarrow \tau'}{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}\ \texttt{of}\ \mathrm{match} \Rightarrow \tau'} \tag{35}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}_1 \Rightarrow \mathbf{Bool} \qquad \mathrm{E} \vdash \mathrm{exp}_2 \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{exp}_3 \Rightarrow \tau}{\mathrm{E} \vdash \texttt{if}\ \mathrm{exp}_1\ \texttt{then}\ \mathrm{exp}_2\ \texttt{else}\ \mathrm{exp}_3 \Rightarrow \tau} \tag{36}$$

$$\frac{\mathrm{E} \vdash \mathrm{decls} \Rightarrow \mathrm{E}' \qquad \mathrm{E} \overset{\rightarrow}{\oplus} \mathrm{E}' \vdash \mathrm{exp} \Rightarrow \tau}{\mathrm{E} \vdash \texttt{let}\ \mathrm{decls}\ \texttt{in}\ \mathrm{exp} \Rightarrow \tau} \tag{37}$$

$$\frac{\mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau}{\mathrm{E} \vdash \mathrm{exp} :: \mathrm{type} \Rightarrow \tau} \tag{38}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{type} \Rightarrow \tau' \qquad \mathit{coerceable}\ (\ \tau,\ \tau'\ )}{\mathrm{E} \vdash \mathrm{exp}\ \texttt{as}\ \mathrm{type} \Rightarrow \tau'} \tag{39}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{exnid} \Rightarrow \mathbf{Exn}\ \tau}{\mathrm{E} \vdash \texttt{raise}\ \mathrm{exnid}\ \mathrm{exp} \Rightarrow \tau'} \tag{40}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}_2 \Rightarrow \mathbf{Time} \qquad \mathrm{E} \vdash \mathrm{exp}_1 \Rightarrow \tau}{\mathrm{E} \vdash \mathrm{exp}_1\ \texttt{within}\ \mathrm{exp}_2[\mathrm{raiseexnid}] \Rightarrow \tau} \tag{41}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau}{\mathrm{E} \vdash \texttt{profile}\ \mathrm{exp} \Rightarrow \tau} \tag{42}$$

2.4.5 Box Templates and Instantiation

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

```
<~~wire~~decl> ::=
        "template" <templateid> <prelude> <body>
```

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

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

2.4.6 Wiring Macros

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

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

The macros are used in place of normal wiring declarations

```
<wiringdecl> ::= "wire" <wmacid> <args>
```

Depending on usage, these arguments may be either box names or names of inputs/outputs. It is not possible to use unrestricted values such as integers as arguments to wiring macros.

For example, we can generate a wiring macro `Track` that is instantiated to wire a number of boxes with identical inputs and outputs in series as follows.

```
constant RingSize = 8;

macro predR i = (i-1) % RingSize;
macro succR i = (i+1) % RingSize;

wire Track ( this, prev, next ) =
  wire {this} ( {this}.value', {prev}.outval, {next}.inctl )
              ( {this}.value,  {next}.outctl, {prev}.inval )
;

for i = 0 to RingSize
        wire Track ( Ring{i}, Ring{predR(i)}, Ring{succR(i)})
;
```

2.4.7 Repeated Wiring

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

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

The repetition variable may be used within the wiring declaration (enclosed within braces), where it takes on each value in the iterator clause in turn. For example,

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

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

2.4.8 Initial Values

It is possible to specify initial values for wires. These may be provided either as part of the link specification for a wire or using an explicit `initial` declaration.

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

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

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

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

For example, we can provide an initialiser for the `value` input of the `Ring{Train1Pos}` box as shown below. Initialisers may be provided either on input or output wires, as convenient. It is, however, an error for more than one initialiser to be provided for any wire.

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

2.4.9 Expression Macros

Expression macros are used to construct simple compile-time macros that are resolved during construction of the static process network.

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

2.4.10 I/O Declarations

Interactions with the operating system and devices are specified in the declaration language.

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


B.4 Static Semantics: Expressions

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

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

$$\frac{(\mathrm{VE}\ \mathit{of}\ \mathrm{E})\ (\mathrm{id}) = \sigma}{\mathrm{E} \vdash \mathrm{id} \Rightarrow \sigma} \tag{23}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}\ \mathrm{exp} \notin \mathrm{Id} \tag{24}$$

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

$$\frac{\mathrm{E} \vdash \mathrm{id} \Rightarrow \forall\, \alpha_1 \ldots \alpha_n.\, \tau \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \tau_i}{\mathrm{E} \vdash \mathrm{id} \Rightarrow \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]} \tag{25}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{char} \Rightarrow \mathbf{Char}} \tag{26}$$

$$\frac{}{\mathrm{E} \vdash \mathrm{string} \Rightarrow \mathbf{String}} \tag{27}$$

$$\frac{\mathrm{E} \vdash \mathrm{con/var} \Rightarrow \tau_1 \to \ldots \to \tau_n \to \tau' \qquad \forall i.\ 1 < i \leq n,\ \ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \tau_i}{\mathrm{E} \vdash \mathrm{con/var}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \tau'}\ n > 0 \tag{28}$$

$$\frac{\mathrm{E} \vdash \forall i.\ 1 \leq i \leq n,\ \mathrm{E} \vdash \mathrm{exp}_i \Rightarrow \tau \Rightarrow \mathbf{List}\ \tau}{\mathrm{E} \vdash [\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ] \Rightarrow \mathbf{List}\ \tau}\ n \geq 1 \tag{29}$$

$$\frac{}{\mathrm{E} \vdash [\,] \Rightarrow \mathbf{List}\ \tau} \tag{30}$$

$$\frac{}{\mathrm{E} \vdash () \Rightarrow \mathbf{Tuple}_0} \tag{31}$$

B.3 Static Semantics: Programs and Wiring

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

$$\frac{\mathrm{E}_0 \oplus_{VE} \mathrm{IE} \vdash \mathrm{decls} \Rightarrow \mathrm{E} \qquad (((\mathrm{E}_0 \overset{\cdots}{\oplus} \mathrm{E}) \oplus_{VE} \mathrm{IE}) \oplus_{VE} \mathrm{SE}) \vdash \mathrm{boxes} \Rightarrow \mathrm{VE} \qquad \mathrm{VE} \vdash \mathrm{wires}}{\mathrm{IE} \vdash \textbf{program}\ \mathrm{decls\ boxes\ wires}} \tag{17}$$

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

$$\frac{\forall i,\ 1 < i \leq n,\ \ \vdash \mathrm{box}_i \Rightarrow \mathrm{VE}_i}{\vdash \mathrm{box}_1 \ \ldots\ \mathrm{box}_n \Rightarrow \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{18}$$

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

$$\frac{\mathrm{E} \vdash \mathrm{body} \Rightarrow \tau \rightarrow \tau' \qquad \mathrm{E} \vdash \mathrm{ins} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{outs} \Rightarrow \tau' \qquad \mathrm{E} \vdash \tau \rightarrow \tau' \Rightarrow \sigma}{\vdash \mathtt{box}\ \mathrm{boxid\ ins\ outs\ body} \Rightarrow \{\ \mathrm{boxid} \mapsto \sigma\ \}} \tag{19}$$

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

$$\frac{\forall i,\ 1 < i \leq n,\ \ \vdash \mathrm{wire}_i}{\vdash \mathrm{wire}_1 \ \ldots\ \mathrm{wire}_n} \tag{20}$$

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

$$\frac{\mathrm{E} \vdash \mathrm{sources} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathrm{dests} \Rightarrow \tau' \qquad \mathrm{E} \vdash \mathrm{boxid} \Rightarrow \tau \rightarrow \tau'}{\vdash \mathtt{wire}\ \mathrm{boxid\ sources\ dests}} \tag{21}$$

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

$$\frac{\mathrm{E} \vdash \mathrm{time} \Rightarrow \mathbf{Time} \qquad \mathrm{E} \vdash \mathrm{matches} \Rightarrow \tau \rightarrow \tau' \qquad \mathrm{E} \vdash \mathrm{handlers} \Rightarrow \tau'}{\mathrm{E}, \mathrm{vs} \vdash \mathtt{timeout}\ \mathrm{time\ matches}\ \mathbf{handle}\ \mathrm{handles} \Rightarrow \tau \rightarrow \tau'} \tag{22}$$


```
        | "port"      <iodes>
        | "interrupt" <iodes>
        | "memory"    <iodes>
        | "fifo"      <iodes>

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

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

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

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

Appendix A

Syntax

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

Programs and Modules

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

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

Declaration Language

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

<decl> :: =
          "import" <modid> <idlist>
        | "export" <idlist>
        | "exception" <exnid> "::" <exprtype>
        | "data" <typeid> <varids> "=" <constrs>
        | "type" <typeid> <varids> "=" <type>
        | "constant" <varid> "=" <cexpr>
        | "stream" <iodes>
        | "port" <iodes>
        | "memory" <iodes>
        | "interrupt" <iodes>
        | "fifo" <iodes>
        | <foreigndecl>
        | "operation" <boxid> "as" <string> "::" <exprtype>
```


$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}{\mathrm{E} \vdash \mathrm{var} = \mathrm{exp} \Rightarrow \{\, \mathrm{var} \mapsto \sigma \,\}, \{\,\}} \tag{8}$$

$$\frac{\mathrm{E} \vdash \mathrm{var} \Rightarrow \sigma \qquad \mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \mathrm{var} :: \mathrm{type} \Rightarrow \{\,\}, \{\,\}} \tag{9}$$

$$\frac{\mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \mathbf{Exn}\ \tau \Rightarrow \sigma}{\mathrm{E} \vdash \mathtt{exception}\ \mathrm{exnid}\ \mathrm{type} \Rightarrow \{\, \mathrm{exnid} \mapsto \sigma \,\}, \{\,\}} \tag{10}$$

$$\frac{(\mathrm{E}\ \vec{\oplus}_{VE}\ (\bigoplus_{i=1}^{n} \{\, \mathrm{var}_i \mapsto \alpha_i \,\}\,)) \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \mathtt{type}\ \mathrm{typeid}\ \mathrm{var}_1 \ldots \mathrm{var}_n = \mathrm{type} \Rightarrow \{\, \mathrm{typeid} \mapsto \sigma \,\}, \{\,\}} \tag{11}$$

$$\frac{\begin{array}{c}\sigma = \forall\, \alpha_1 \ldots \alpha_n.\ \chi\ \alpha_1 \ldots \alpha_n \qquad \mathrm{VE} = \{\, \mathrm{typeid} \mapsto \sigma \,\} \qquad \mathrm{TE} = \{\, \chi \,\} \\ ((\mathrm{E}\ \vec{\oplus}_{VE}\ (\bigoplus_{i=1}^{n} \{\, \mathrm{var}_i \mapsto \alpha_i \,\}\,)) \oplus \mathrm{VE}\,), \tau \vdash \mathrm{constrs} \Rightarrow \mathrm{VE}'\end{array}}{\mathrm{E} \vdash \mathtt{data}\ \mathrm{typeid}\ \mathrm{var}_1 \ldots \mathrm{var}_n = \mathrm{constrs} \Rightarrow (\mathrm{VE} \oplus \mathrm{VE}'), \mathrm{TE}} \tag{12}$$

$$\frac{\forall i.\ 1 \le i \le n,\ (\mathrm{IE}\ \mathit{of}\ \mathrm{E})\ (\mathrm{var}_i) = \sigma_i}{\mathrm{E} \vdash \mathtt{importmodid}\ \mathrm{var}_1 \ldots \mathrm{var}_n \Rightarrow \bigoplus_{i=1}^{n} \{\, \mathrm{var}_i \mapsto \sigma_i \,\}, \{\,\}} \tag{13}$$

$$\frac{}{\mathrm{E} \vdash \mathtt{export}\ \mathrm{var}_1 \ldots \mathrm{var}_n \Rightarrow \{\,\}, \{\,\}} \tag{14}$$

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

$$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E}, \tau \vdash \mathrm{constr}_i \Rightarrow \mathrm{VE}_i}{\mathrm{E}, \tau \vdash \mathrm{constr}_1 \mid \ldots \mid \mathrm{constr}_n \Rightarrow \bigoplus_{i=1}^{n} \mathrm{VE}_i} \tag{15}$$

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

$$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \vdash \mathrm{type}_i \Rightarrow \tau_i' \qquad \mathrm{E} \vdash \tau_1' \to \ldots \to \tau_n' \to \tau \Rightarrow \sigma}{\mathrm{E}, \tau \vdash \mathrm{conid}\ \mathrm{type}_1 \ldots \mathrm{type}_n \Rightarrow \{\, \mathrm{conid} \mapsto \sigma \,\}}\ n \ge 0 \tag{16}$$

B.2 Static Semantics: Declarations

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

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

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

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

$$\frac{\mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \texttt{foreign import}\ [\,\mathrm{s}\,]\ [\,\mathrm{c}\,]\ [\,\mathrm{str}\,]\ \mathrm{var} :: \mathrm{type} \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \},\ \{\ \}} \quad (2)$$

$$\frac{\mathrm{E} \vdash \mathrm{var} \Rightarrow \sigma \qquad \mathrm{E} \vdash \mathrm{type} \Rightarrow \tau \qquad \mathrm{E} \vdash \tau \Rightarrow \sigma}{\mathrm{E} \vdash \texttt{foreign export}\ [\,\mathrm{c}\,]\ [\,\mathrm{str}\,]\ \mathrm{var} :: \mathrm{type} \Rightarrow \{\ \},\ \{\ \}} \quad (3)$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow \sigma}{\mathrm{E} \vdash \texttt{constant}\ \mathrm{var} = \mathrm{exp} \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \},\ \{\ \}} \quad (4)$$

$$\frac{\begin{array}{c}(\mathrm{SE}\ \mathit{of}\ \mathrm{E})\ \mathrm{var}' = \forall\, \alpha_1\ \ldots\ \alpha_n.\ \tau \qquad [\mathrm{E} \vdash \mathrm{cexpr} \Rightarrow \tau] \\ \mathrm{E} \vdash \textbf{Port}\ \tau/\textbf{Stream}\ \tau/\textbf{Memory}\ \tau/\textbf{Fifo}\ \tau/\textbf{Interrupt}\ \tau \Rightarrow \sigma\end{array}}{\begin{array}{c}\mathrm{E} \vdash \texttt{port/stream/memory/fifo/interrupt}\ \mathrm{var}\ \texttt{from}\ \mathrm{var}'\ [\texttt{initial}\ \mathrm{cexpr}] \\ \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \},\ \{\ \}\end{array}} \quad (5)$$

$$\frac{\mathrm{E} \vdash \mathrm{matches} \Rightarrow \sigma \qquad \mathit{primrec}\ (\ \mathrm{var}\ \mathrm{matches}\ )}{\mathrm{E} \vdash \mathrm{var}\ \mathrm{matches} \Rightarrow \{\ \mathrm{var} \mapsto \sigma\ \},\ \{\ \}} \quad (6)$$

$$\frac{\mathrm{E} \vdash \mathrm{op}\ \mathrm{pat}_1\ \mathrm{pat}_2 = \mathrm{exp} \Rightarrow \mathrm{VE}, \mathrm{TE}}{\mathrm{E} \vdash \mathrm{pat}_1\ \mathrm{op}\ \mathrm{pat}_2 = \mathrm{exp} \Rightarrow \mathrm{VE}, \mathrm{TE}} \quad (7)$$


```
        | "expression" <expr>
        | <boxdecl>
        | <wiringdecl>
        | <fundecl>

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

<iodes> ::=
            <ioid> ( "from" | "to" ) <string>
               [ "timeout" <cexpr> [ "raise" <exnid> ] ]

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

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

<vardecl> ::=
            <varid> "::" <type>
        | <varid> "=" <expr>

<vardecls> ::=
            <vardecl1> ";" ... ";" <vardecln>       n >= 1

<foreigndecl> ::=
            "foreign" "import" <callconv> [ <safety> ] [ <string> ] <id> "::" <exprtype>
        | "foreign" "export" <callconv> [ <string> ] <id> "::" <exprtype>

<safety> ::=
            "safe" | "unsafe"

<callconv> ::=
            "ccall" | "stdcall" | "cplusplus" | "jvm" | "dotnet"
```

Types

```
<type> ::=
            <exprtype>
        | "time"                            time type

<exprtype> ::=
            <basetype>                      base type
        | "vector" <intconst1> "of" <type>  vector
        | "()"                              empty tuple
        | "(" <type> "," <types> ")"        tuple
        | "[" <types> "]"                   list
        | <typeid> <type1> ... <typen>      discr. union, n >= 0
```

```
        | <type> "->" <type>              function type
        | "view" <type>                   view as type
        | "(" <exprtype> ")"              grouping

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

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

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

Expression Language

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


<constraint> ::=                      time, heap (stack)
<cexpr> [ "," <cexpr> [ "(" <cexpr> ")" ] ]
<cexpr> ::= <expr>                                            constant expression
```


Appendix B

Static Semantics

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

B.1 Static Semantics: Notation

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

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

BasVal and *BasCon* are fully defined in Section B.8. The function *coerceable* is defined with reference to the table in Section 2.1.3.

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

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

```
<charconst>   ::= "'" <char> "'"
<stringconst> ::= """ <char> * """
<wordconst>   ::= "0x" <hexdigit> +
<timeconst>   ::= <intconst> <timedes>
<spaceconst>  ::= <intconst> <spacedes>
<timedes>     ::= "ps" | "ns" | "us" | "ms" | "s" | "min"
<spacedes>    ::= "B" | "KB" | "MB"
<char>        ::= "A" | ... | "Z" | " " | "\t" | "\n" | "\\" |
  "\" <digit> + | "\0x" <hexdigit> +
```

"a" ... "z" ?

<digit>
<hexdigit>


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

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

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

Constants

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

Patterns

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

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

Coordination language

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

<boxprelude> ::=
        "in"  <inoutlist>
        "out" <inoutlist>
        [ "handles" <exnidlist> ]
        [ "timeout" <cexpr> ]
```

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

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

Boxes

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

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

<boxmatches> ::=
        <matches>

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

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

Wiring MetaLanguage

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

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

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

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

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

Wiring

```
<wiredecl> ::=
          "wire" <wireid> <sources> <dests>
        | "wire" <wireid> <idlist> "=" <wireid> <sources> <dests>
        | "wire" <link1> "to" <link2>

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

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

<linkspec> ::=
          <id> "." <id>
        | <ioid>

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

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

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

Identifiers

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

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

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

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

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

<boxid>/<modid>/<exnid>/<varid>/<conid>/<typeid>  ::= <id>
<streamid>/<portid>/<intid>/<fifoid>/<memid>      ::= <id>

<ioid>      ::= <streamid> | <portid> | <intid> | <fifoid> | <memid>

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

Lexical Syntax

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

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

<intconst>    ::= <digit> +
<floatconst>  ::= <intconst> "." <intconst> [ "e" <intconst> ]
<boolconst>   ::= "true" | "false"
```