Chapter 1

Hume: a Functionally-Inspired Language for Safety-Critical Systems

Kevin Hammond[1] and Greg Michaelson[2]

Abstract: Hume is a novel functionally-based language that targets safety-critical applications. The language is designed to support rigorous cost and space analyses, whilst providing a high level of abstraction including polymorphic type inference, automatic memory management, higher-order functions, exception-handling and a good range of primitive types.

1.1 INTRODUCTION

This paper describes the Hume programming language. *Hume* (Higher-order Unified Meta-Environment) is a strongly typed, mostly-functional language with an integrated tool set for developing, proving and assessing concurrent, safety-critical systems. Hume aims to extend the frontiers of safety-critical language design, introducing new levels of abstraction and provability. A full description of Hume including a formal dynamic semantics and a partially completed static semantics can be found at `http://www-fp.dcs.st-and.ac.uk/hume`. We anticipate publishing this material as a formal language definition in the near future.

[1]School of Computer Science, University of St Andrews, North Haugh, St Andrews, KY16 9SS, Scotland; Email: `kh@dcs.st-and.ac.uk`

[2]Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH? ???, Scotland; Email: `greg@cee.hw.ac.uk`

1.1.1 Background

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 we have followed in constructing Hume.

1.2 MOTIVATION AND OBJECTIVES

Since the focus of the Hume design is on safety critical applications, 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 complete 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. Hume reflects these considerations in:

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

1.2.1 Important Design Characteristics

Any system that is aimed at safety critical applications must meet a number of stringent criteria if it is to be used with confidence. Hume has been designed to support Leveson's guidelines for *Safeware* — software intended for safety critical applications [11], as well as similar guidelines provided by other authors [2].

In general, safety critical 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 Hume 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. This first version of the language design is deliberately rather spartan, allowing experimentation with essential features but omitting some desirable syntax or other language features, such as type classes. 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 safety.

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 safety critical 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 [3].

1.3 THE HUME LANGUAGE

This section introduces the Hume language informally. A full description, from which this section is abstracted, can be found in the Hume language definition [13].

1.3.1 Structure of the Language

In common with other coordination language approaches such as Linda [4], 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 fairly conventional *expression language* which is used to define values and (potentially higher-order) functions using purely functional expressions. Finally, the middle layer is a *coordination language* that links functions into possibly concurrent processes.

Exceptions transcend the layers. They are declared by `exception <id> <type>` within declarations, raised by `raise <id> <expr>` within expressions and handled by boxes.

1.3.2 The Hume Expression Language

The Hume *expression language* is a purely functional language with a strict semantics. It is intended to be used for the description of single, one-shot, non-reentrant processes. It is deterministic, and has statically bounded time and space behaviour. In order to achieve this, expressions that are not the target of dynamic timeouts must be restricted to statically-checkable primitive recursive forms. This allows the construction of verifiable static cost analyses. Dynamically bounded time and space expressions (`within`-expressions) specify that evaluation of the associated expression must complete within the specified constant time or space. Failure to do so causes the `Timeout` or `HeapOverflow` exception to be raised.

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

1.3.3 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.

1.3.4 The Hume Declaration Language

The declaration language introduces types and values that scope over either or both the coordination and expression languages. The coordination language is embedded in the declaration language through box and wiring declarations while the expression language is embedded through function and value declarations. While it is possible to define recursive and mutually recursive functions, it is not possible to do the same for simple values.

1.4 EXAMPLE PROGRAMS

1.4.1 Parity Checking

Our first example is a simple even parity checker whose input is wired to a communications port (`comm1`), and whose output is wired to its second input. The initial parity value is defined to be `true`.

```
box even_parity
in  (b::word 1,p::bool)
out (p'::bool)

match
  (0,true)  -> true
| (1,true)  -> false
| (0,false) -> false
```

```
| (1,false) -> true;

wire even_parity (comm1, even_parity.p' initially true)
                 (even_parity.p);
```

1.4.2 Checksum

The second example calculates the checksum of 256 input numbers. It returns a string which is either "OK" or an error message. This example illustrates (somewhat artificially) the use of exceptions and handlers.

```
type short = int 16; type long = int 32;

box checksum
in  (n::short, sum::long, count::short)
out (sum'::long, count'::short, message::string)
handles CheckSumError

match
  (n,sum,256) ->   (0,0,if n=sum then "OK"
                         else raise CheckSumError ())
| (n,sum,count) -> (sum+n,count+1,"")

handle
  CheckSumError _ ->
    (0,0,"Checksum error: sum = " +tostring(sum)+
         " checksum ="+tostring(n));

wire checksum (comm1, checksum.sum' initially 0,
               checksum.count' initially 0)
              (checksum.sum, checksum.count, stdout);
```

1.5 COSTING HUME PROGRAMS

Figures 1.1– 1.6 give rules that can be used to statically derive a bounded time cost for first-order Hume programs (we are independently working on time systems for higher-order programs [12]). The rules are a simple big-step operational semantics, with extensions to timeouts and exceptions. We have deliberately used a very simple cost semantics here rather than the more general form that we are developing as part of our work in parallel computation [12]. The latter is intended for arbitrary recursion and requires the use of a constraint solving engine, whereas Hume is restricted to primitive recursive forms. It would be straightforward to alter these rules to cover space rather than time cost, as required.

The cost rules specified here derive notation from the Hume static and dynamic semantics, which is in turn derived from that used for Haskell and Standard ML.

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

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

$$\boxed{\mathrm{E} \vdash \mathrm{decl} \Rightarrow \mathrm{E}, \mathrm{Cost}}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow c}{\mathrm{E} \vdash \mathrm{var}\ =\ \mathrm{exp} \Rightarrow \{\ \}, c}$$

$$\frac{\mathrm{E} \overset{\rightarrow}{\oplus} \mathrm{var} \mapsto \infty \vdash \mathrm{matches} \Rightarrow c, c'}{\mathrm{E} \vdash \mathrm{var\ matches} \Rightarrow \{\ \mathrm{var} \mapsto c + c'\ \}, 0}$$

FIGURE 1.1. Cost axioms for declarations

Environments $\{\mathrm{var}_1 \mapsto c_1, \ldots, \mathrm{var}_n \mapsto c_n\}$ map variables to cost expressions. The notation E $\oplus$ E' is the result of combining the bindings in E with those in E' where the domains of E and E' are disjoint; while E $\overset{\rightarrow}{\oplus}$ E' represents the combination of the two environments, but replacing bindings for variables in E with those from E' where there is overlap.

$\mathrm{E}, \mathrm{E'} \in$	Env =	$\{\mathrm{var} \mapsto \mathrm{Cost}\}$	Cost Environments
$c, t \in$	Cost =	$0, 1, \ldots, \infty$	Cost Values
$\sigma, \sigma' \in$	PolyType =	$\forall$ var . MonoType	Polymorphic Types
$\tau \in$	MonoType =	TyCon MonoType* + TyVar	Monomorphic Types
$\chi \in$	TyCon =	$\chi_{\mathrm{Int}_n}, \chi_{\mathrm{Bool}}, \cdots$	Type Constructors
$\alpha \in$	TyVar =	$\alpha, \beta, \ldots$	Type Variables

Arithmetic operations $+$, *max* etc. are defined on cost expressions in the obvious way as for normal integers, with the addition of an infinite cost value ∞, which is treated in the obvious way. Type constructors include an indication of their size. So $\chi_{\mathrm{Int}_{16}}$ is a 16-bit fixed-precision integer etc.

The cost rules given here rely on the auxiliary function *value* : exp $\rightarrow$ Cost which delivers the semantic value of an expression. In its sole use (for timeouts) this must be a finite natural number corresponding to a cost value.

Figure 1.1 gives rules for calculating the costs of declaration sequences. A declaration yields an environment mapping variables to costs, and a separate cost.

$$\boxed{\text{E} \vdash \text{body} \Rightarrow \text{Cost}}$$

$$\frac{\text{E} \vdash \text{matches} \Rightarrow c, c' \qquad \text{E} \vdash \text{handlers} \Rightarrow c''}{\text{E} \vdash \texttt{match}\ \text{matches}\ \texttt{handle}\ \text{handlers} \Rightarrow c + c' + c''} \quad c \neq \infty \wedge c' \neq \infty$$

$$\frac{\text{E} \vdash \text{time} \Rightarrow t \qquad \text{E} \vdash \text{matches} \Rightarrow c, c' \qquad \text{E} \vdash \text{handlers} \Rightarrow c''}{\text{E} \vdash \texttt{match}\ \text{matches}\ \texttt{timeout}\ \text{time}\ \texttt{handle}\ \text{handles} \Rightarrow min(t, c + c') + c''}$$

FIGURE 1.2. Cost axioms for boxes

Only two forms of declaration are interesting. Variable declarations are evaluated at the start of a declaration block. They therefore have a fixed dynamic cost. The cost of a function (of the form *var matches*) is defined as the cost of matching all patterns in the set of matches, plus the cost of the most expensive expression on the right-hand-sides of the matches. This therefore computes an upper bound on the cost of evaluating a function body. Recursive definitions are assigned infinite cost. The cost is bound to the function name in the environment so that the cost of each and every function call is correctly calculated. The result of a declaration sequence is the union of the environment recording function costs

Figure 1.2 gives the cost of a single box iteration. This is defined similarly to the cost of a function definition, but including the upper bound of the exception handling cost. There are two cases. If no `timeout` clause is given, then the matches must be finite. Otherwise the cost is the lesser of the timeout and the actual execution cost plus the handler cost.

Figures 1.3–1.4 give cost rules for expressions. The cost of a function application is the cost of evaluating the body of the function plus the cost of each argument. The cost of building a new constructor such as a list or tuple is the cost of evaluating the arguments to that constructor plus one for each argument (representing the cost of building each heap cell). Note that the cost of a non-null list $[\text{exp}_1, \ldots, \text{exp}_n]$ includes the cost of the final [], and is therefore greater than the equivalently sized tuple $(\text{exp}_1, \ldots, \text{exp}_n)$ or vector $< \text{exp}_1, \ldots, \text{exp}_n >$. The cost of a coercing an expression to a given type using an *as*-expression depends on the original type of the expression and the one that it is coerced to. There is, of course, no cost for a simple type cast. The cost of raising an exception is the cost of the enclosed expression plus one (representing the cost of actually throwing that exception). Finally the cost of an expression enclosed within a timeout is the minimum of the expression cost and the specified timeout.

Figure 1.5 gives costs for pattern matches. Two values are returned from a match sequence. The first is the total cost of matching all the patterns in the match sequence. This places an upper bound on the match cost. The second is the cost of evaluating the most expensive right-hand-side. This places an upper

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

$$\frac{}{E \vdash \text{id} \Rightarrow 1}$$

$$\frac{E(\text{var}) \ = \ c \qquad \forall i.\ 1 < i \leq n,\ \ E \vdash \text{exp}_i \Rightarrow c_i}{E \vdash \text{var}\ \text{exp}_1\ \ldots\ \text{exp}_n \Rightarrow \sum_{i=1}^{n} c_i + c}\ \ n \ > \ 0$$

$$\frac{}{E \vdash \text{literal} \Rightarrow 1}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \ E \vdash \text{exp}_i \Rightarrow c_i}{E \vdash \text{con}\ \text{exp}_1\ \ldots\ \text{exp}_n \Rightarrow \sum_{i=1}^{n} c_i + 1}$$

$$\frac{}{E \vdash [\,] \Rightarrow 1}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ E \vdash \text{exp}_i \Rightarrow c_i}{E \vdash [\ \text{exp}_1,\ \ldots,\ \text{exp}_n\ ] \Rightarrow \sum_{i=1}^{n} c_i + (n+1)}$$

$$\frac{}{E \vdash () \Rightarrow 1}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ E \vdash \text{exp}_i \Rightarrow c_i}{E \vdash (\ \text{exp}_1,\ \ldots,\ \text{exp}_n\ ) \Rightarrow \sum_{i=1}^{n} c_i + n}$$

$$\frac{}{E \vdash <\ \ > \Rightarrow 1}$$

$$\frac{\forall i.\ 1 \leq i \leq n,\ E \vdash \text{exp}_i \Rightarrow c_i + (n+1)}{E \vdash <\ \text{exp}_1,\ \ldots,\ \text{exp}_n\ \ > \Rightarrow \sum_{i=1}^{n} c_i + n}$$

FIGURE 1.3. Cost axioms for expressions (variables, applications and constructors)

bound on the expression cost. The cost of matching multiple patterns in a single match is the sum of matching each pattern.

Wildcard patterns (_) cost nothing to match, whereas normal variables have a match cost (here specified as one). The cost of matching a constructor pattern is the cost of matching all subsidiary patterns plus one for the constructor itself. The costs of matching lists, tuples and vectors are defined similarly, noting that it is always necessary to include the cost of matching the `[]` at the end of a list, but

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

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow c \qquad \mathrm{E} \vdash \mathrm{match} \Rightarrow c'}{\mathrm{E} \vdash \texttt{case}\ \mathrm{exp}\ \texttt{of}\ \mathrm{match} \Rightarrow c + c'}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp}_1 \Rightarrow c_1 \qquad \mathrm{E} \vdash \mathrm{exp}_2 \Rightarrow c_2 \qquad \mathrm{E} \vdash \mathrm{exp}_3 \Rightarrow c_3}{\mathrm{E} \vdash \texttt{if}\ \mathrm{exp}_1\ \texttt{then}\ \mathrm{exp}_2\ \texttt{else}\ \mathrm{exp}_3 \Rightarrow c_1 + max(c_2, c_3)}$$

$$\frac{\mathrm{E} \vdash \mathrm{decls} \Rightarrow \mathrm{E'}, c \qquad \mathrm{E} \overset{\rightarrow}{\oplus} \mathrm{E'} \vdash \mathrm{exp} \Rightarrow c'}{\mathrm{E} \vdash \texttt{let}\ \mathrm{decls}\ \texttt{in}\ \mathrm{exp} \Rightarrow c + c'}$$

$$\frac{\mathrm{E} \vdash \mathrm{exp} \Rightarrow c}{\mathrm{E} \vdash \texttt{raise}\ \mathrm{exnid}\ \mathrm{exp} \Rightarrow c + 1}$$

$$\frac{value(\mathrm{exp}_2) = t \qquad \mathrm{E} \vdash \mathrm{exp}_1 \Rightarrow c}{\mathrm{E} \vdash \mathrm{exp}_1\ \texttt{within}\ \mathrm{exp}_2 \Rightarrow min(c, t)}$$

FIGURE 1.4. Cost axioms for expressions (cases, conditionals and exceptions)

that empty vectors and tuples can be deduced by type.

Figure 1.6 gives the cost of matching exceptions. Since only one handler can ever be matched, and unlike function definitions or boxes, matches are not made in sequence, the cost of a handler sequence is the greatest cost of any exception handler. The cost of an individual handler is the cost of the match plus 1 for returning the (compile-time-constant) expression that forms its right-hand-side.

1.6 RELATED WORK

During the course of the 20th century, much fundamental work was done on primitive recursion. Crudely, primitive recursion differs from Turing Complete general recursion in forbidding unbounded minimisation. However, there are further subtleties in that nested recursion may give otherwise primitive recursive programs greater complexity than true primitive recursion. For example, Ackermann showed that the function named for him has a faster growth rate than primitive recursion [1]. On the other hand, there are nested recursive programs which have primitive recursive complexity. Complexity analysis of primitive recursive programs is also undecidable and complexity analysis heuristics often lead to further recursive forms which are not straightforward to characterise succinctly.

Turner's *elementary strong functional programming* [18] has explored issues of guaranteed termination in a purely functional programming language. Turner's

$$\boxed{\text{E} \vdash \text{Matches} \Rightarrow \text{Cost}, \text{Cost}}$$

$$\frac{\text{E} \vdash \{\text{ match }\} \Rightarrow c, c' \qquad \text{E} \vdash \{\text{ matches }\} \Rightarrow c'', c'''}{\text{E} \vdash \{\text{ match} \mid \text{matches }\} \Rightarrow c + c'', max(c', c''')}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \text{E} \vdash \text{pat}_i \Rightarrow c_i \qquad \text{E} \vdash \text{exp} \Rightarrow c}{\text{E} \vdash \{\text{ pat}_1\ \ldots\ \text{pat}_n\ \rightarrow\ \text{exp }\} \Rightarrow \sum_{i=1}^{n} c_i, c}$$

$$\boxed{\text{E} \vdash \text{pat} \Rightarrow \text{Cost}}$$

$$\frac{}{\text{E} \vdash \_ \Rightarrow 0} \qquad \frac{}{\text{E} \vdash \text{var} \Rightarrow 1}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \text{E} \vdash \text{pat}_i \Rightarrow c_i}{\text{E} \vdash \text{con pat}_1\ \ldots\ \text{pat}_n \Rightarrow \sum_{i=1}^{n} c_i + 1}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \text{E} \vdash \text{pat}_i \Rightarrow c_i}{\text{E} \vdash (\text{ pat}_1,\ \ldots,\ \text{pat}_n\ ) \Rightarrow \sum_{i=1}^{n} c_i}\ n \geq 2 \qquad \frac{}{\text{E} \vdash () \Rightarrow 0}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \text{E} \vdash \text{pat}_i \Rightarrow c_i}{\text{E} \vdash <\ \text{pat}_1,\ \ldots,\ \text{pat}_n\ > \Rightarrow \sum_{i=1}^{n} c_i}\ n > 0 \qquad \frac{}{\text{E} \vdash <> \Rightarrow 0}$$

$$\frac{\forall i.\ 1 < i \leq n,\ \text{E} \vdash \text{pat}_i \Rightarrow c_i}{\text{E} \vdash [\ \text{pat}_1,\ \ldots,\ \text{pat}_n\ ] \Rightarrow \sum_{i=1}^{n} c_i + 1}\ n > 0 \qquad \frac{}{\text{E} \vdash [] \Rightarrow 1}$$

FIGURE 1.5. Cost axioms for pattern matches

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 Hume, it is necessary to identify functions that may be more generally recursive.

Ada is widely used for embedded systems, and many tools have been constructed to assist the understanding of space and time behaviour [2]. 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.

Finally, 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 [17]

$$\boxed{\text{E} \vdash \text{Handlers} \Rightarrow \text{Cost}}$$

$$\frac{\text{E} \vdash \text{pat} \Rightarrow c \qquad \text{E} \vdash \text{exp} \Rightarrow c'}{\text{E} \vdash \{\ \text{exnid pat} \ \rightarrow\ \text{exp}\ \} \Rightarrow c+1} \quad c' \neq \infty$$

$$\frac{\text{E} \vdash \{\ \text{handler}\ \} \Rightarrow c \qquad \text{E} \vdash \{\ \text{handlers}\ \} \Rightarrow c'}{\text{E} \vdash \{\ \text{handler} \mid \text{handlers}\ \} \Rightarrow \mathit{max}(c, c')}$$

FIGURE 1.6. Cost axioms for exception handlers

and RTJava [9], for soft real-time applications. Like Hume, both languages support dynamic memory allocation with automatic garbage collection and provide strong exception handling mechanism. The primary differences from Hume are the incorporation of arbitrary recursion in all cases, 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 is paramount. 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.

1.7 CONCLUSION

This paper has introduced the Hume language, a novel functionally-inspired language that is geared towards safety critical applications. A key feature of the design is the incorporation of a purely functional expression language that provides strong guarantees of functional correctness with equally strong guarantees of time and space behaviour. To our knowledge, this is the first fully-featured functional language that supports such guarantees.

To illustrate the key features of our language design, we have produced a simple bounded first-order time cost analysis. The analysis makes a number of simplifying assumptions, such as costing all heap allocations, exception raises and variable bindings identically (as 1 in this set of definitions). It would, however, be straightforward to parameterise the analysis on more realistic costs, or even to modify it so as to calculate space rather than time costs.

Clearly, Hume is still at an early stage of development. None of our language tools has yet been constructed and important design details are still to be resolved. We feel, however, that the properties designed into the language should yield long-term benefits in the safety-critical arena.

REFERENCES

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

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

[3] C. Clack, "The Demense Model for Bounded Automatic Memory Management", Unpublished Report, University College, London, 2000.

[4] D. Gelernter, and N. Carriero, "Coordination Languages and Their Significance", *CACM*, **32**(2), February, 1992, pp. 97–107.

[5] A.D. Gordon, S. Marlow and S.L. Peyton Jones, "A Semantics for Imprecise Exceptions", *Proc. Symposium on Principles of Programming Languages (POPL 2000)*, January 2000.

[6] M.G. Hinchey and J.P. Bowen, *Applications of Formal Methods*, Prentice-Hall, 1995.

[7] 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.

[8] 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.

[9] D. Jensen, "'Real-time Java'. What could and what should it mean?" URL: `http://www.sdct.itl.nist.gov/~carnahan/real-time/vocab/rt-java_glossary.htm`, May 1998.

[10] H. Jifeng, *Provably Correct Systems: Modelling of Communication Languages and Design of Optimised Compilers*, McGraw-Hill, 1995.

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

[12] H.-W. Loidl, A.J. Rebón Portillo and Kevin Hammond, "A Sized Time System for a Parallel Functional Language (Revised)", *In Preparation*, St Andrews, July 2000.

[13] G.A. Michaelson and K.Hammond, "The Hume Language Definition and Report, Version 0.0", University of St Andrews and Heriot-Watt University, 2000.

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

[15] *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, J.C. Peterson, A. Reid, and P.L. Wadler, Yale University, 1999.

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

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

[18] 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.