Predictable Space Behaviour in FSM-Hume

Kevin Hammond[1] and Greg Michaelson[2]

[1] School of Computer Science,
University of St Andrews, St Andrews, Scotland.
Email: kh@dcs.st-and.ac.uk. Tel: +44-1334-463241, Fax: +44-1334-463278

[2] Dept. of Mathematics and Computer Science,
Heriot-Watt University, Edinburgh, Scotland.
Email: greg@macs.hw.ac.uk. Tel: +44-131-451-3422, Fax: +44-131-451-3327

Abstract. The purpose of the Hume language design is to explore the expressibility/decidability spectrum in resource-constrained systems, such as real-time embedded or control systems. It is unusual in being based on a combination of λ-calculus and finite state machine notions, rather than the more usual propositional logic, or flat finite-state-machine models. It provides a number of high level features including polymorphic types, arbitrary but sized user-defined data structures and automatic memory management, whilst seeking to guarantee strong space/time behaviour and maintaining overall determinacy. A key issue is predictable space behaviour. This paper describes a simple model for calculating stack and heap costs in *FSM-Hume*, a limited subset of full Hume. This cost model is evaluated against an example taken from the research literature: a simple mine drainage control system. Empirical results suggest that our model is a good predictor of stack and heap usage, and that this can lead to good bounded memory utilisation.

1 Introduction

Hume is a functionally-based research language aimed at applications requiring bounded time and space behaviour, such as real-time embedded systems. It is possible to identify a number of overlapping subsets of the full Hume language, increasing in expressive power, but involving increasingly complicated cost models. The simplest is the language that is studied here, FSM-Hume, which is restricted to first-order non-recursive functions and non-recursive data structures, but which supports a form of implicit tail recursion across successive iterations of a process. Despite its restrictions, FSM-Hume is still capable of expressing a variety of problems in the embedded/real-time systems sphere. One such problem, a simple mine drainage control system, is described in this paper. The paper defines a simple bounded space usage cost model for FSM-Hume and evaluates it against this sample application. We demonstrate that it is possible to produce good cost models for FSM-Hume and that these can be used to give good bounded space usage in practice.

program ::=	$decl_1$; ... ; $decl_n$	$n \geq 1$
decl ::=	*box* | var *matches* | *datadecl* | *wiredecl*	
datadecl ::=	**data** id α_1 ... α_m = $constr_1$ | ... | $constr_n$	$n \geq 1$
constr ::=	con τ_1 ... τ_n	$n \geq 1$
wiredecl ::=	**wire** id *ins* *outs*	
box ::=	**box** *id* *ins* *outs* **fair/unfair** *matches*	
ins/outs ::=	$id_1, \ldots, id_n$	
matches ::=	$\langle$ match_1 , ... , match_n $\rangle$	$n \geq 1$
match ::=	$\langle$ pat_1 , ... , pat_n $\rangle$ $\rightarrow$ *expr*	
expr ::=	*int* | *float* | *char* | *bool* | *string* | var | *	
	| con exp_1 ... exp_n	$n \geq 0$
	| (exp_1 , ... , exp_n)	$n \geq 2$
	| **if** *cond* **then** exp_1 **else** exp_2	
	| **let** $\langle$ $vdecl_1$, ..., $vdecl_n$ $\rangle$ **in** *expr*	
vdecl ::=	*id* = exp	
pat ::=	*int* | *float* | *char* | *bool* | *string* | var | _ | * | _*	
	| con var_1 ... var_n	$n \geq 0$
	| (pat_1 , ... , pat_n)	$n \geq 2$

Fig. 1. Hume Abstract Syntax (Simplified)

2 Boxes and Coordination

In order to support concurrency, Hume requires both computation and coordination constructs. The fundamental unit of computation in Hume is the *box*, which defines a finite mapping from inputs to outputs. Boxes are *wired* into (static) networks of concurrent processes using wiring directives. Each box introduces one process. This section introduces such notions informally. A more formal treatment is in preparation.

Boxes are abstractions of finite state machines. An output-emitting Moore machine has transitions of the form:

$$(\textit{old state, input symbol}) \rightarrow \quad (\textit{new state, output symbol})$$

We generalise this to:

$$\textit{pattern} \rightarrow \textit{function}(\textit{pattern})$$

where *pattern* is based on arbitrarily nested constants, variables and data structures and *function* is an arbitrary recursive function over *pattern* written in the *expression language*. By controlling the types permissible in *pattern* and the constructs usable in *function*, the expressibility and hence formal properties of Hume may be altered. Where types are sized and constructs restricted to non-conditional operations, Hume has decidable time and space behaviour. As construct restrictions are relaxed to allow primitive and then general recursion, so expressibility increases, and decidable equivalence and then decidable termination are lost. A major objective of our research is to explore static analyses that tell the programmer when decidable properties are compromised in particular programs, rather than placing explicit restrictions on the forms of all programs.

The abstract syntax of Hume is shown in Figure 1. A single Hume box comprises a set of pattern-directed rules, rewriting a set of inputs to a set of outputs, plus appropriate exception handlers and type information. The left hand side pattern of each rule defines the situations in which that rule may be active. The right hand side of each rule is an expression specifying the results of the box when the rule is activated. A box becomes active when any of its rules may match the inputs that have been provided. Hume expressions are written using a strongly-typed purely functional notation with a strict semantics. The functional notation simplifies cost modelling and proof. The use of strict evaluation ensures that source and target code can be directly related, thus improving confidence in the correct operation of the compilation system. Strong typing improves correctness assurances, catching a large number of surface errors at relatively low overall programmer cost, as well as assisting the required static cost/space analyses. Each expression is deterministic, and has statically bounded time and space behaviour, achieved through a combination of static cost analysis and dynamic timeout constructs, where the timeout is explicitly bounded to a constant time. Since the expression language has no concept of external, imperative state, such considerations must be encapsulated entirely through explicit communication in the coordination language.

2.1 Wiring

Boxes are connected using wiring declarations to form a static process network. A wire provides a mapping between inputs and outputs. Each box input must be connected to precisely one output. An output may, in principle, be connected to multiple inputs, but is normally connected to a unique input. The usual form of a wiring declaration specifies the input/output mappings for a single box. This is technically redundant in that the opposite mapping must also be specified in the boxes to which a given box is wired. It has the advantage from a language perspective of concentrating wiring information for a single box close to the definition of that box. It also allows boxes to be connected to non-box objects (external ports/streams, such as the program's standard input or output). It is possible to specify the initial value that appears on a wire. This is typically used to seed computations, such as wires carrying explicit state parameters.

2.2 Box Example

The Hume code for a simple even parity checking box is shown below. The inputs to the box are a bit (either 0 or 1) and a boolean value indicating whether the system has detected even or odd parity so far. The output is a string indicating whether the result should be even or odd parity. This box defines a single cycle.

```
type bit = word 1;  type parity = boolean;

box even_parity
in  ( b  :: bit,    p :: parity )
out  (p' :: parity, show :: string)
unfair
  ( 0, true  ) -> ( true, "true" )
| ( 1, true  ) -> ( false, "false" )
| ( 0, false ) -> ( false, "false" )
| ( 1, false ) -> ( true, "false" );
```

The corresponding wiring specification connects the bit stream to the `input` source and the monitoring output to standard output. Note that the output `p'` is wired to the box input `p` as an explicit state parameter, initialised to `true`. The box will run continuously, outputting a log of the monitored parity.

```
stream input from "/dev/sensor";
stream output to "std_out";

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

2.3 Coordination

The basic box execution cycle is:

1. check input availability for all inputs and latch input values;
2. match inputs against rules in turn;
3. consume all inputs;
4. bind variables to input values and evaluate the RHS of the selected rule;
5. write outputs to the corresponding wires.

A key issue is how input and output values are managed. In the Hume model, there is a one-to-one correspondance between input and output wires, and these are single-buffered. In combination with the fixed size types that we require, this ensures that communications buffers are bounded size, whilst avoiding the synchronisation problems that can occur if no buffering is used. In particular, a box may write an output to one of its own inputs, so creating an explicit representation of state, as shown in the example above.

Values for available inputs are latched atomically, but not removed from the buffer (consumed) until a rule is matched. Consuming an input removes the lock on the wire buffer, resetting the availability flag.

Output writing is atomic: if any output cannot be written to its buffer because a previous value has not yet been consumed, the box blocks. This reduces concurrency by preventing boxes from proceeding if their inputs could be made available but the producer is blocked on some other output. However, it improves strong notions of causality: if a value has appeared as input on a wire the box that produced that input has certainly generated all of its outputs.

Once a cycle has completed and all outputs have been written to the corresponding wire buffers, the box can begin the next execution step. This improves concurrency, by avoiding unnecessary synchronisation. Individual boxes never terminate. Program termination occurs when no box is runnable.

2.4 Asynchronous Coordination Constructs

The two primary coordination constructs that are used to introduce asynchronous coordination are to *ignore* certain inputs/outputs and to introduce *fair matching*. It is necessary to alter the basic box execution cycle as follows (changes are italicised):

1. check input availability *against possible matches* and latch *available* input values;
2. match *available* inputs against rules in turn;
3. consume *those inputs that have been matched and which are not ignored in the selected rule*;
4. bind variables to input values and evaluate the RHS of the selected rule;
5. write *non-ignored* outputs to the corresponding wires;
6. *reorder match rules according to the fairness criteria.*

Note that: i) inputs are now consumed after rules have been selected rather than before; ii) only some inputs/outputs may be involved in a given box cycle, rather than all inputs/outputs being required; and iii) rules may be reordered if the box is engaged in fair matching. This new model in which inputs can be ignored in certain patterns or in certain output positions can be considered to be equivalent to non-strictness at the box level.

We use the accepted notion of *fairness* whereby each rule will be used equally often given a stream of inputs that match all rules [1]. *Channel fairness* [1] is not enforced, however: it is entirely possible, for example, for a programmer to write a sequence of rules that will treat the input from different sources unfairly. It is

for $i\ =\ 1$ **to** $nThreads$ **do**
  $runnable\ :=\ false$;
  for $j\ = 1$ **to** $thread[i].nRules$ **do**
    if $\neg\ runnable$ **then**
      $runnable\ :=\ true$;
      for $k\ = 1$ **to** $thread[i].nIns$ **do**
        $runnable\ \&=\ thread[i].required[j,k]\ \Rightarrow\ thread[i].ins[k].available$
      endfor
    endif
  endfor
  if $runnable$ **then** $schedule\ (thread[i])$ **endif**
endfor

Fig. 2. Hume Abstract Machine Thread Scheduling Algorithm

the programmer's responsibility to ensure that channel fairness is maintained, if required.

For example, a fair merge operator can be defined as:

```
box merge
in  ( xs  :: int 32, ys :: int 32)
out ( xys :: int 32)
fair
  (x, *) -> x
| (*, y) -> y
;
```

The *-pattern indicates that the corresponding input position should be ignored, that is the pattern matches any input, without consuming it. Such a pattern must appear at the top level. Note the difference between *-patterns and wildcard/variable patterns: in the latter cases, successful matching will mean that the corresponding input value (and all of that value) is removed from the input buffer. For convenience, we also introduce a hybrid pattern: _*. If matched, such patterns will consume the corresponding input value *if one is present*, but will ignore it otherwise. Note that this construct cannot introduce a race condition, since the availability status for each input is latched at the start of each box execution cycle rather than checked during each pattern match. Ignored values can also be used as dynamic outputs. In this case no output is produced on the corresponding wire, and consequently the box cannot be blocked on that output.

2.5 Thread Scheduling

The prototype Hume Abstract Machine implementation maintains a vector of threads (*thread*), one per box, each with its own *thread state record*, containing

constant	value (words)
$\mathcal{H}_{con}$	3
$\mathcal{H}_{tuple}$	2
$\mathcal{H}_{int32}$	2
$\mathcal{H}_{float32}$	2
...	...
$\mathcal{H}_{string}$	1

Fig. 3. Sizes of tags etc. in the prototype Hume Abstract Machine

state information and links to input/output wires. Each wire comprises a pair of a value (*value*) and a validity flag (*available*). used to ensure correct locking between input and output threads. The flag is atomically set to *true* when an output is written to the wire, and is reset to *false* when an input is consumed.

Threads are scheduled under the control of a built-in scheduler, which currently implements round-robin scheduling. A thread is deemed to be *runnable* if all the required inputs are available for any of its rules to be executed (Figure 2). A compiler-specified matrix is used to determine whether an input is needed: for some thread t, $thread[t].required[r, i]$ is true if input i is required to run rule r of that thread. Since wires are single-buffered, a thread will consequently block when writing to a wire which contains an output that has not yet been consumed. In order to ensure a consistent semantics, a single check is performed on all output wires immediately before any output is written. No output will be written until all the input on all output wires has been consumed. The check ignores ***** output positions.

3 A Space Cost Model for FSM-Hume

This section describes a simple cost model for space usage in FSM-Hume boxes. The model is defined with reference to the prototype Hume Abstract Machine [8], and provides a statically derivable upper bound on the space usage of FSM-Hume programs in terms of per-box stack and heap usage. The cost analysis is applied to the mine drainage example described in the previous section, and verified against the prototype Hume Abstract Machine Interpreter. A complete, formal description of the abstract machine and compiler can be found elsewhere [8]. The stack and heap requirements for the boxes and wires represent the only dynamically variable memory requirements: all other memory costs can be fixed at compile-time based on the number of wires, boxes, functions and the sizes of static strings. In the absence of recursion, we can provide precise static memory bounds on rule evaluation. Predicting the stack and heap requirements for an FSM-Hume program thus provides complete static information about system memory requirements.

$$\boxed{\mathrm{E} \overset{\text{box}}{\vdash} \text{box} \Rightarrow \text{Cost}, \text{Cost}}$$

$$(1)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{type}}{\vdash} \tau_i \Rightarrow h_i \qquad \mathrm{E} \overset{\text{body}}{\vdash} \text{body} \Rightarrow h, s}{\mathrm{E} \overset{\text{box}}{\vdash} \textbf{box}\ \text{id}\ \textbf{in}\ (\ \text{id}_1 : \tau_1, \ldots,\ \text{id}_n : \tau_n\ )\ \textbf{out}\ \text{outs body} \Rightarrow \sum_{i=1}^{n} h_i + h, s}$$

$$\boxed{\mathrm{E} \overset{\text{body}}{\vdash} \text{body} \Rightarrow \text{Cost}, \text{Cost}}$$

$$(2)\quad \frac{\begin{array}{c}\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{pat}}{\vdash} \text{ps}_i \Rightarrow sp_i \\ \forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_i \Rightarrow h_i, s_i\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{body}}{\vdash} (\ \textbf{fair} \mid \textbf{unfair}\ )\ \text{ps}_1 -> \text{exp}_1 \mid \ldots \mid \text{ps}_n -> \text{exp}_n \\ \Rightarrow \overset{n}{\underset{i=1}{max}}\ h_i,\ \overset{n}{\underset{i=1}{max}}\ (s_i + sp_i)\end{array}}$$

Fig. 4. Space cost axioms for boxes and box bodies

3.1 Dynamic Memory in the Hume Abstract Machine

The Hume Abstract Machine is loosely based on the design of the classical G-Machine [3], restricted to strict evaluation and with extensions to manage concurrency and asynchronicity. Each box has its own dynamic stack and heap. All arguments to function calls, return values and box inputs are held on the stack as (1-word) heap pointers. All available box inputs are copied from the corresponding wire into the box heap at the start of each cycle. All other heap allocation happens as a consequence of executing some right-hand-side expression.

In the prototype implementation, all heap cells are *boxed* [15] with tags distinguishing different kinds of objects. Furthermore, tuple structures require *size* fields, and data constructors also require a *constructor tag* field. All data objects in a structure are referenced by pointer. For simplicity each field is constrained to occupy one word of memory. There is one special representation: strings are represented as a tagged sequence of bytes. These values are summarised in Figure 3. Clearly, it would be easy to considerably reduce heap usage using a more compact representation such as that used by the state-of-the-art STG-Machine [15]. For now, we are, however, primarily concerned with bounding and predicting memory usage. Small changes to data representations can be easily incorporated into both models and implementations at a future date without affecting the fundamental results described here, except by reducing absolute costs of both model and implementation.

$$\boxed{\mathrm{E} \overset{\text{type}}{\vdash} \text{type} \Rightarrow \text{Cost}}$$

$$(3)\quad \frac{}{\mathrm{E} \overset{\text{type}}{\vdash} \textbf{int}\ 32 \Rightarrow \mathcal{H}_{int32}}$$

...

$$(4)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{type}}{\vdash} \tau_i \Rightarrow h_i}{\mathrm{E} \overset{\text{type}}{\vdash} (\ \tau_1\ ,\ \ldots\ ,\ \tau_n\ ) \Rightarrow \mathcal{H}_{con} + \sum_{i=1}^{n} h_i}$$

$$\boxed{\mathrm{E} \overset{\text{data}}{\vdash} \text{data} \Rightarrow \text{Cost}}$$

$$(5)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{constr}}{\vdash} constr_i \Rightarrow h_i}{\mathrm{E} \overset{\text{type}}{\vdash} \textbf{data}\ \text{id}\ \alpha_1\ \ldots\ \alpha_m = \text{constr}_1 \mid \ldots \mid \text{constr}_n \Rightarrow \max_{i=1}^{n} h_i}$$

$$\boxed{\mathrm{E} \overset{\text{constr}}{\vdash} \text{constr} \Rightarrow \text{Cost}}$$

$$(6)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{type}}{\vdash} \tau_i \Rightarrow h_i}{\mathrm{E} \overset{\text{constr}}{\vdash} \text{con}\ \tau_1\ \ldots\ \tau_n \Rightarrow \mathcal{H}_{con} + \sum_{i=1}^{n} h_i}$$

Fig. 5. Heap cost axioms for types

3.2 Space Cost Rules

Figures 4–8 specify a space cost model for FSM-Hume boxes and declarations, based on an operational interpretation of the Hume abstract machine implementation. Heap and stack costs are each integer values of type Cost, labelled h and s, respectively. Each rule produces a pair of such values representing an independent upper bound on the stack and heap usage. The result is produced in the context of an environment, E, that maps function names to the space (heap and stack) requirements associated with executing the body of the function. This environment is derived from the top-level program declarations plus standard prelude definitions. Rules for building the environment are omitted here, except for local declarations, but can be trivially constructed.

Rules 1 and 2 (Figure 4) cost boxes and box bodies, respectively. The cost of a box is derived from the space requirements for all box inputs plus the maximum cost of the individual rule matches in the box. The cost of each rule match is derived from the costs of the pattern and expression parts of the rule. Since the abstract machine copies all available inputs into box heap from the wire buffer

$$\boxed{\mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp} \Rightarrow \mathrm{Cost}, \mathrm{Cost}}$$

$$(7) \quad \frac{}{\mathrm{E} \overset{\text{space}}{\vdash} n \Rightarrow \mathcal{H}_{int32}, 1}$$

...

$$(8) \quad \frac{\mathrm{E}\ (\mathrm{var}) = \langle h, s \rangle \qquad \forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp}_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} \mathrm{var}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \sum_{i=1}^{n} h_i + h,\ \max_{i=1}^{n}\ (s_i + (i-1)) + s}$$

$$(9) \quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp}_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} \mathrm{con}\ \mathrm{exp}_1\ \ldots\ \mathrm{exp}_n \Rightarrow \sum_{i=1}^{n} h_i + n + \mathcal{H}_{con},\ \max_{i=1}^{n}\ (s_i + (i-1))}$$

$$(10) \quad \frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp}_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} (\ \mathrm{exp}_1,\ \ldots,\ \mathrm{exp}_n\ ) \Rightarrow \sum_{i=1}^{n} h_i + n + \mathcal{H}_{tuple},\ \max_{i=1}^{n}\ (s_i + (i-1))}$$

$$(11) \quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp}_1 \Rightarrow h_1, s_1 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp}_2 \Rightarrow h_2, s_2 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \mathrm{exp}_3 \Rightarrow h_3, s_3}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{if}\ \mathrm{exp}_1\ \texttt{then}\ \mathrm{exp}_2\ \texttt{else}\ \mathrm{exp}_3 \Rightarrow h_1 + max(h_2, h_3), max(s_1, s_2, s_3)}$$

$$(12) \quad \frac{\begin{array}{c} \mathrm{E} \overset{\text{decl}}{\vdash} \mathrm{decls} \Rightarrow h_d, s_d, s'_d, \mathrm{E'} \\ \mathrm{E'} \overset{\text{space}}{\vdash} \mathrm{exp} \Rightarrow h_e, s_e \end{array}}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{let}\ \mathrm{decls}\ \texttt{in}\ \mathrm{exp} \Rightarrow h_d + h_e, max(s_d, s'_d + s_e)}$$

Fig. 6. Space cost axioms for expressions

before they are matched, the maximum space usage for box inputs is the sum of the maximum space required for each input type. The space required for each output wire buffer is determined in the same way from the type of the output value. Figure 5 gives representative costs for integers (rule 3) and tuples (rule 4), and rules 5–6 provide costs for user-defined constructed datatypes.

Figure 6 gives cost rules for a representative subset of FSM-Hume expressions. The heap cost of a standard integer is given by $\mathcal{H}_{int32}$ (rule 7), with other scalar values costed similarly. The cost of a function application is the cost of evaluating the body of the function plus the cost of each argument (rule 8). Each evaluated argument is pushed on the stack before the function is applied, and

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

$$(13)\quad \frac{}{\mathrm{E} \overset{\text{pat}}{\vdash} n \Rightarrow 0}$$

...

$$(14)\quad \frac{}{\mathrm{E} \overset{\text{pat}}{\vdash} \text{var} \Rightarrow 1}$$

$$(15)\quad \frac{\forall i.\ 1 \leq i \leq n,\ \ \mathrm{E} \overset{\text{pat}}{\vdash} \text{pat}_i \Rightarrow s_i}{\mathrm{E} \overset{\text{pat}}{\vdash} \text{con pat}_1 \ \ldots \ \text{pat}_n \Rightarrow \sum_{i=1}^{n} s_i + n}$$

$$(16)\quad \frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E} \overset{\text{pat}}{\vdash} \text{pat}_i \Rightarrow s_i}{\mathrm{E} \overset{\text{pat}}{\vdash} (\ \text{pat}_1,\ \ldots,\ \text{pat}_n\ ) \Rightarrow \sum_{i=1}^{n} s_i + n}$$

Fig. 7. Stack cost axioms for patterns

this must be taken into account when calculating the maximum stack usage. The cost of building a new data constructor value such as a tuple (rule 10) or a user-defined constructed type (rule 9) is similar to a function application, except that pointers to the arguments must be stored in the newly created closure (one word per argument), and fixed costs $\mathcal{H}_{con}$ and $\mathcal{H}_{tuple}$ are added to represent the costs of tag and size fields. The heap usage of a conditional (rule 11) is the heap required by the condition part plus the maximum heap used by either branch. The maximum stack requirement is simply the maximum required by the condition and either branch. Case expressions (omitted) are costed analogously. The cost of a let-expression (rule 12) is the space required to evaluate the value definitions (including the stack required to store the result of each new value definition) plus the cost of the enclosed expression. The local declarations are used to derive a quadruple comprising total heap usage, maximum stack required to evaluate any value definition, a count of the value definitions in the declaration sequence (used to calculate the size of the stack frame for the local declaratons), and an environment mapping function names to heap and stack usage (rule 19 – Figure 8). The body of the let-expression is costed in the context of this extended environment.

Finally, patterns contribute to stack usage in two ways (Figure 7): firstly, the value attached to each variable is recorded in the stack frame (rule 14); and secondly, each nested data structure that is matched must be unpacked onto the

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

$$(17)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \text{exp} \Rightarrow h, s}{\mathrm{E} \overset{\text{decl}}{\vdash} \text{id} = \text{exp} \Rightarrow h, s, 1, \{\}}$$

$$(18)\quad \frac{\begin{array}{c} \forall i.\ 1 \leq i \leq n,\ \ \mathrm{E} \overset{\text{pat}}{\vdash} \text{pat}_i \Rightarrow sp_i \\ \forall i.\ 1 \leq i \leq n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_i \Rightarrow h_i, s_i \\ \mathrm{E'} \ = \ \{ \text{id} : \langle \overset{n}{\underset{i=1}{max}}\ h_i,\ \overset{n}{\underset{i=1}{max}}\ (s_i + sp_i) \} \rangle \end{array}}{\mathrm{E} \overset{\text{decl}}{\vdash} \text{id pat}_1 = \text{exp}_1 \mid \ldots \mid \text{id pat}_n = \text{exp}_n \Rightarrow 0, 0, 0, \mathrm{E'}}$$

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

$$(19)\quad \frac{\begin{array}{c} \forall i.\ 1 \leq i \leq n,\ \ \mathrm{E} \overset{\text{decl}}{\vdash} \text{decl}_i \Rightarrow h_i, s_i, s'_i, \mathrm{E}_i \\ \mathrm{E'} \ = \ \mathrm{E} \ \overset{\rightarrow}{\oplus} \ \bigcup_{i=1}^{n} \mathrm{E}_i \end{array}}{\mathrm{E} \overset{\text{decl}}{\vdash} \text{decl}_1 \ ; \ \ldots \ ; \ \text{decl}_n \Rightarrow \sum_{i=1}^{n} h_i,\ \overset{n}{\underset{i=1}{max}}\ h_i, \sum_{i=1}^{n} (s_i + s'_i + \sum_{j=1}^{i-1} s'_j), \mathrm{E'}}$$

Fig. 8. Cost axioms for function and value declarations

stack (requiring n words of stack) before its components can be matched by the abstract machine (rules 15 and 16).

4 The Mine Drainage Control System

As a working example we have chosen a simple control application with strong real-time requirements. This application has previously been studied in the context of a number of other languages, including Ada with real-time extensions [5]. It was originally constructed as a realistic exemplar for control applications and comprises 750 lines of Ada or about 250 lines of FSM-Hume, of which the functional core is about 100 lines. We have also constructed a Java version.

The problem is to construct software for a simplified pump control system, which is to be used to drain water from a mine shaft. The system runs on a single processor with memory-mapped I/O. The pump is used to remove water that is collected in a sump at the bottom of the shaft to the surface. To avoid damaging the pump, it should not be operated when the water level in the sump is

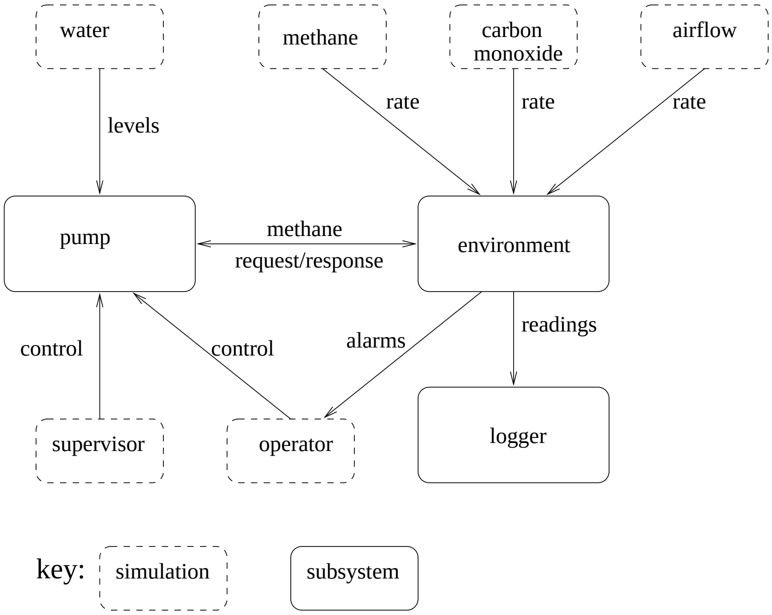

Fig. 9. Hume Processes for the Mine Drainage Control System

below a certain level. It must, however, be activated if the water rises above a certain level in order to avoid the risk of flooding. The main safety requirement is that the pump must not be operated when the concentration of methane gas in the atmosphere reaches a certain level. This is to avoid the risk of explosion. In order to ensure this, an environmental control station monitors information returned by sensors in the mine shaft, including the current methane level, the current level of carbon monoxide and the airflow. The pump is under the control of an operator who issues commands to turn the pump on and off subject to rules governing the pump operation. The operator may be overridden by their supervisor. All actions and periodic states of the sensors are logged.

Figure 9 shows the corresponding FSM-Hume process network. The primary processes are the pump controller, *pump*, the environment monitoring process, *environment* and the logging process *logger*. The *water*, *methane*, *carbon monoxide* and *airflow levels* are simulated, as are the *operator* and *supervisor*. Additional links between components to enable whole system monitoring through the logger are not shown but are present in the actual wiring. The application is fully described elsewhere [9].

4.1 Space Analysis of the Mine Drainage Example

The cost rules specified above have been extended to full FSM-Hume and implemented as a 200-line Haskell [14] module, which has then been integrated with

box	predicted heap	actual heap	excess	predicted stack	actual stack	excess
airflow	16	16	0	10	10	0
carbonmonoxide	16	16	0	10	10	0
environ	37	35	2	27	26	1
logger	144	104	40	25	25	0
methane	16	16	0	10	10	0
operator	38	29	9	23	23	0
pump	51	42	9	21	18	3
supervisor	29	29	0	20	20	0
water	54	54	0	20	20	0
(wires)	96	84	8	0	0	0
TOTAL	483	425	68	166	162	4

Fig. 10. Heap and stack usage in words for boxes in the mine drainage control system

the standard Hume parser and lexer and applied to the mine drainage control program. Figure 10 reports predicted and actual maximum stack and heap usage for 12,000,000 iterations of the box scheduler under the prototype Hume Abstract Machine Interpreter.

The results in Figure 10 show completely accurate cost predictions for the majority of boxes, with more serious variations for the heap usage of the *logger*, *operator* and *pump* boxes. These three boxes are all relatively complex boxes with many alternative choices. Moreover, since these boxes are asynchronous, they will frequently become active when only a few inputs are available. Since unavailable inputs do not contribute to heap usage, the cost estimate will therefore be noticeably larger than the actual usage. The logger function also makes extensive use of string append. Since the size of the result will vary dynamically in some cases, the heap usage will consequently have been overestimated in some cases. Overall, the space requirements of the boxes have been overestimated by 72 words or 11% of the total actual dynamic requirement. Since precise figures may be undecidable in general (for example, where a possible execution path is never taken in practice), this represents a good but not perfect static estimate of dynamic space usage.

To verify that FSM-Hume can yield constant space requirements in practice, the pump application has been run continuously on a 1GHz Pentium III processor under RTLinux, kernel version 2.4.4. RTLinux [4] is a realtime micro-kernel operating system, which supports multiple realtime threads and runs a modified Linux system as a separate non-realtime thread. Realtime threads communicate with non-realtime Unix tasks using realtime FIFOs which appear as normal Unix devices, and which are the only I/O mechanism available to realtime threads other than direct memory-mapped I/O. The system guarantees a $15\mu s$ worst-case thread context-switch time for realtime threads.

Our measurements [8] show that the total memory requirements of the pump application, including heap and stack overheads as calculated here, *RTLinux operating system code and data*, Hume runtime system code and data, and the abstract machine instructions amount to less than 62Kbytes. RTLinux itself accounts for 34.4Kbytes of this total. Since little use is made of RTLinux facilities, and there is scope for reducing the size of the Hume abstract machine, abstract machine instructions, and data representation, *we conclude that it should be possible to construct full Hume applications requiring much less than 32Kbytes of memory, including runtime system support, for bare hardware as found in typical embedded systems.* This storage requirement is well within the capabilities of common modern parts costing $30 or less.

5 Related Work

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [16]. Hume is thus, as far as we are aware, unique in being based on strong automatic cost models, and in being designed to allow straightforward space- and time-bounded implementation for hard real-time systems, those systems where tight real-time guarantees must be met. A number of functional languages have, however, looked at *soft* real-time issues [2, 19, 7, 13], and there has been considerable recent interest both in the problems associated with costing functional languages [16, 10, 6, 18] and in bounding space/time usage [11, 17]. All of these approaches other than our own require programmer interaction in the form of cost control annotations.

6 Conclusions and Further Work

This paper has introduced the FSM-Hume subset of Hume, a novel concurrent language aimed at resource-limited systems such as real-time embedded systems. We have defined and implemented a simple cost semantics for stack heap usage in FSM-Hume, which has been validated against an example from the real-time systems literature, a simple mine drainage control system and implemented using the prototype Hume Abstract Machine interpreter. Our empirical results demonstrate that it is possible to define a concurrent functionally-based language with bounded and predictable space properties. Moreover, such a language can be remarkably expressive: the example presented here is representative of a range of real applications. We anticipate that it will be possible to build a full Hume implementation for a bare-bones system, such as a typical embedded systems application using less than 32KBytes of memory, including all code and data requirements. We are currently working on extensions to our cost model to cover recursion and higher-order functions [16]. In the longer term, we intend to demonstrate the formal soundness of our cost model against the Hume Abstract Machine and compiler.

References

1. K.R. Apt and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, 2nd Edition, Springer Verlag, 1997.
2. J. Armstrong, S.R. Virding, and M.C. Williams, *Concurrent Programming in Erlang*, Prentice-Hall, 1993.
3. L. Augustsson, *Compiling Lazy Functional Languages, Part II*, PhD Thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, 1987.
4. M. Barabanov, *A Linux-based Real-Time Operating System*, M.S. Thesis, Dept. of Comp. Sci., New Mexico Institute of Mining and Technology, June 97.
5. A. Burns and A. Wellings, *Real-Time Systems and Programming Languages* (Third Edition), Addison Wesley Longman, 2001, Chapter 17, pp. 653–684.
6. R. Burstall, "Inductively Defined Functions in Functional Programming Languages", Dept. of Comp. Sci., Univ. of Edinburgh, ECS-LFCS-87-25, April, 1987.
7. D.H. Fijma and R.T. Udink, "A Case Study in Functional Real-Time Programming", Dept. of Computer Science, Univ. of Twente, The Netherlands, *Memoranda Informatica 91-62*, 1991.
8. K. Hammond and G.J. Michaelson "An Abstract Machine Implementation for Embedded Systems Applications in Hume", `http://www.hume-lang.org/papers/HAM.ps`, January 2003.
9. K. Hammond and G.J. Michaelson "The Mine Drainage Control System in Hume", `http://www.hume-lang.org/examples/pump`, January 2003.
10. R.J.M. Hughes, L. Pareto, and A. Sabry. "Proving the Correctness of Reactive Systems Using Sized Types", *Proc. POPL'96 — ACM Symp. on Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.
11. 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)*, Paris, France, pp. 70–81, 1999.
12. J. McDermid, "Engineering Safety-Critical Systems", I. Wand and R. Milner(eds), *Computing Tomorrow: Future Research Directions in Computer Science*, Cambridge University Press, 1996, pp. 217–245.
13. J.C. Peterson, P. Hudak and C. Elliot, "Lambda in Motion: Controlling Robots with Haskell", *First International Workshop. on Practical Aspects of Declarative Languages (PADL '99)*, San Antonio, Texas, January 1999, Springer-Verlag LNCS No. 1551, pp. .91–105.
14. 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, *Report on the Non-Strict Functional Language, Haskell (Haskell98)* Yale University, 1999.
15. S. L. Peyton Jones, "Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine", *J. Funct. Prog.*, **2**(2): 127–202, 1992.
16. A.J. Rebón Portillo, Kevin Hammond, H.-W. Loidl and P. Vasconcelos, "A Sized Time System for a Parallel Functional Language", *Proc. Intl. Workshop on Implementation of Functional Languages (IFL 2002)*, Madrid, Spain, Sept. 2002.
17. M. Tofte and J.-P. Talpin, "Region-based Memory Management", *Information and Control*, **132**(2), 1997, pp. 109–176.
18. D.A. Turner, "Elementary Strong Functional Programming", *Proc. Symp. on Funct. Prog. Langs. in Education — FPLE '95*, Springer-Verlag LNCS No. 1022, Dec. 1995.
19. M. Wallace and C. Runciman, "Extending a Functional Programming System for Embedded Applications", *Software: Practice & Experience*, **25**(1), January 1995.