Hume: a Domain-Specific Language for Real-Time Embedded Systems

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. This paper describes Hume: a novel domain-specific language whose purpose is to explore the expressibility/costability spectrum in resource-constrained systems, such as real-time embedded or control systems. Hume provides a number of high level features including higher-order functions, polymorphic types, arbitrary but sized user-defined data structures, asynchronous processes, lightweight exception handling, automatic memory management and domain-specific metaprogramming features, whilst seeking to guarantee strong space/time behaviour and maintaining overall determinacy.

1 Essential Language Properties in the Real-Time Embedded System Domain

Embedded systems are becoming increasingly prevalent in everyday life. From a computer science perspective such systems represent a software/hardware blend at a much lower level of abstraction than usual. They also introduce strict cost-driven requirements on system capabilities. Adapting general purpose languages for such a domain often leads to a poor fit between the language features and the implementation requirements. Domain-specific language designs such as Hume allow low-level system requirements to guide the design of the high level language features we desire.

We can identify a number of essential or desirable properties for a language that is aimed at real-time embedded systems [19].

- *determinacy* – the language should allow the construction of determinate systems, by which we mean that under identical environmental constraints, all executions of the system should be *observationally equivalent*;
- *bounded time/space* – the language must allow the construction of systems whose resource costs are statically bounded – so ensuring that *hard real-time* and *real-space* constraints can be met;

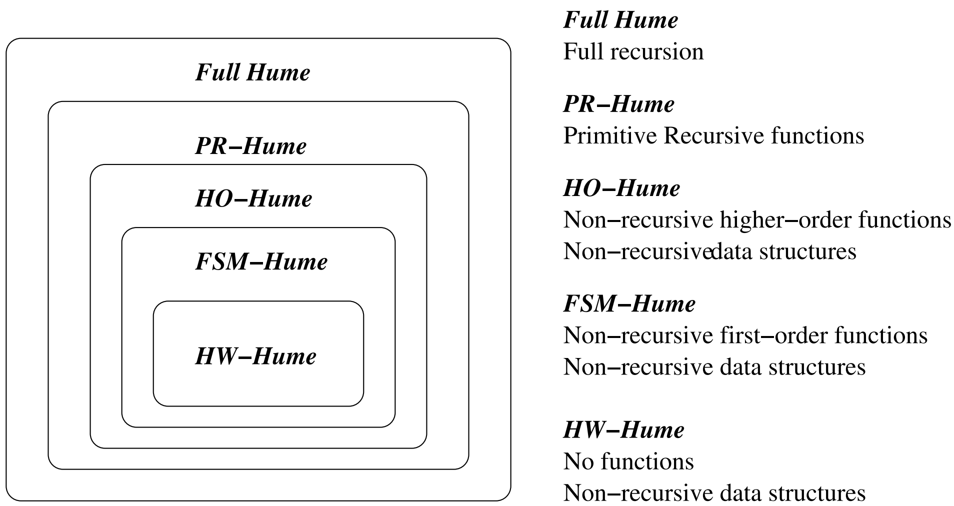

Fig. 1. Hume Design Space

- *asynchronicity* – the language must allow the construction of systems that are capable of responding to inputs as they are received without imposing total ordering on environmental or internal interactions;
- *concurrency* – the language must allow the construction of systems as communicating units of independent computation;
- *correctness* – the language must allow a high degree of confidence that constructed systems meet their formal requirements [1].

Our goal with the Hume design is to allow as high a level of program abstraction as possible whilst still maintaining these properties. For example, we provide exception handling, automatic memory management, higher-order functions, polymorphism and recursion in different levels of the language. In each case, the requirement for transparent time and space costing has been paramount in formulating the language design.

1.1 The Hume Design Model

We have designed Hume as a three-layer language: an outer (static) declaration/metaprogramming layer, an intermediate coordination layer describing a static layout of dynamic processes, and an inner layer describing each process as a dynamic mapping from patterns to expressions.

Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework either directly or by altering the language to a greater or lesser extent (as with e.g. RTSj [6]), our approach is to design Hume in such a way that we are certain that formal models and proofs can be constructed. We envisage a series of overlapping Hume language levels as shown in Figure 1, where each level adds expressibility to the expression semantics, but either loses some desirable property or increases the technical difficulty of providing formal correctness/cost models.

program ::=	$decl_1$; ... ; $decl_n$	$n \geq 1$
decl ::=	*box* | *function* | *datatype* | *exception* | *wire* | *device*	
function ::=	| var *matches*	
datatype ::=	**data** id $\alpha_1 \ldots \alpha_m$ = $constr_1$ | ... | $constr_n$	$n \geq 1$
constr ::=	con $\tau_1 \ldots \tau_n$	$n \geq 1$
exception ::=	**exception** id [:: τ]	
wire ::=	**wire** $link_1$ **to** $link_2$ [**initially** *cexpr*]	
link ::=	*connection* | *deviceid*	
connection ::=	*boxid* . *varid*	
box ::=	**box** *id* *ins* *outs* **fair/unfair** *matches* [**handle** *exnmatches*]	
ins/outs/ids ::=	(id_1 , ... , id_n)	
matches ::=	$match_1$ | ... | $match_n$	$n \geq 1$
match ::=	(pat_1 , ... , pat_n) $\rightarrow$ *expr*	
expr ::=	*int* | *float* | *char* | *bool* | *string* | var | *	
	| con $expr_1 \ldots expr_n$	$n \geq 0$
	| ($expr_1$, ... , $expr_n$)	$n \geq 2$
	| **if** *cond* **then** $expr_1$ **else** $expr_2$	
	| **let** $valdecl_1$; ... ; $valdecl_n$ **in** *expr*	
	| *expr* **within** (*time* | *space*)	
valdecl ::=	*id* = *expr*	
pat ::=	*int* | *float* | *char* | *bool* | *string* | var | _ | * | _*	
	| con $var_1 \ldots var_n$	$n \geq 0$
	| (pat_1 , ... , pat_n)	$n \geq 2$
device ::=	(**stream** | **port** | **fifo** | **memory** | **interrupt**) *devdesc*	
devdesc ::=	*id* (**from** | **to**) *string* [**within** *time* **raising** *id*]	

Fig. 2. Hume Syntax (Simplified)

Figure 2 shows the syntax of Hume. We use a rule-based design, with a functional expression notation embedded in an asynchronous process model. This simplifies both correctness proofs and the construction of cost models at the expression level. Process abstractions ("boxes") specify an asynchronous and stateless mapping of inputs to outputs. These abstractions can be seen as stateless objects with a rigid communication structure, which both assists process/communication costing and simplifies the construction of deadlock/termination proofs. Boxes are wired explicitly into a static process network (again simplifying both correctness and costing) using a single-buffer approach. Single-buffering allows tight controls over buffer sizes based on the types of values that are communicated, gives a simple easily implemented semantics, and can be extended to multiple buffering stages by simply introducing additional intermediary boxes.

2 Boxes and Coordination

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

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

We generalise this to:

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

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 that are permissible in *pattern* and the constructs that are usable in *function*, the expressibility and hence formal properties of Hume may be altered.

Although the body of a box is a single function, the process defined by a box will iterate indefinitely. Since a box is stateless, information that is preserved between box iterations must be passed explicitly between those iterations through some *wire* (Section 2.1). This roughly corresponds to tail recursion over a stream in a functional language, as used by E-FRP, for example [?]. In the Hume context, this design allows a box to be implemented as an uninterruptible thread, taking its inputs, computing some result values and producing its outputs. Moreover, if a bound on dynamic memory usage can be predetermined, a box can execute with a fixed size stack and heap without requiring garbage collection.

2.1 Wiring

Boxes are connected using wiring declarations to form a static process network. A wire provides a mapping between an output link and an input link, each of which may be a named box input/output, a port, or a stream. For example, `wire x.o to y.i` connects the output `x.o` to the input `x.i`. Initial values may be specified. For example, `initial x.o = 1` specifies that the initial value of the wire connected to `x.o` is `1`.

2.2 Box and Wiring 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.

```
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 input            to even_parity.b;
wire even_parity.p'   to even_parity.p    initially true;
wire even_parity.show to 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 [2]. *Channel fairness* [2] 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 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 that might affect the meaning of a box, since the availability status for each input is latched at the start of each box execution cycle rather than checked during each individual 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.

For example, a box can be written that waits for a boolean trigger, discarding any input values that arrive before the trigger is fired. It is unimportant whether the rules are matched fairly or unfairly.

```
box sink
in  ( trigger :: boolean, i  :: int 16 )
out ( i' :: int 16 )
unfair
   ( false, _* )  ->  *
|  ( true,  x  )  ->  x;
```

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 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 3). 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 The MetaProgramming Layer

The Hume metalanguage (Figure 4) is designed to simplify the construction of boxes and wiring by creating abstractions that can be instantiated at compile-time. The metalanguage supports box templates which may be instantiated to one or more boxes; defined constants used as identifiers both in expressions and in the metalanguage; macro expressions and parameterised macros which may be used in the metalanguage, in declarations, and in expressions; and abstract wiring/initial declarations. The syntax of the metalanguage is shown in Figure 4. *cexpr* is used to indicate a statically computable expression that is evaluated in the metaprogramming layer.

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

Fig. 3. Hume Abstract Machine Thread Scheduling Algorithm

meta ::=	**constant** *id* = *cexpr*	
	| **template** *templateid prelude body*	
	| **instantiate** *templateid* **as** *boxid* [* *intconst*]	
	| **replicate** *boxid* **as** *boxid* [* *intconst*]	
	| **macro** *mid ids* = *cexpr*	
	| *wiredecl*	
	| **for** *id* = *cexpr* **to** *cexpr* [**except** *excepts*] *meta*	
excepts ::=	($cexpr_1$, … , $cexpr_n$)	$n \geq 1$
wire ::=	**wire** *boxid sources dests*	
	| **wire** *wmacid* (id_1 , … , id_n) = *wireid sources dests*	
	| **wire** *wmacid args*	
sources/dests/args ::=	($linkspec_1$, … , $linkspec_n$)	$n \geq 0$
linkspec ::=	*link* [**intially** *cexpr*]	

Fig. 4. Hume MetaProgramming Constructs

3.1 Box Templates

A box template can be defined to give the structure of a box, which is then instantiated to produce one or more boxes. Such templates are polymorphic. For convenience, we also allow boxes to be replicated, but in this case the replicated box has the same monomporphic type as the original. The box/template may optionally be replicated/instantiated a number of times. For example, `instantiate t as b * 4` will introduce boxes `b1`, `b2`, `b3` and `b4`.

In the second example, we generate a `Track` template which is then instantiated to give 16 `Ring` boxes, named `Ring0` to `Ring15`, and 4 `ByPass` boxes, named `ByPass1` to `ByPass4`. This example forms part of an example model railway track layout defined by Roscoe to illustrate deadlock issues in CSP [?]. The layout comprises two loops of track with a common section. The `ByPass` creates a short circuit of the track by forking from the `Ring` at some point (`ForkPos`) and joining it at a later point (`JoinPos`). The `Fork` and `Join` boxes are also template instantiations. Note the reuse of the same input/output names in the different templates. This is used to assist abstraction.

```
template Track
in  ( value  :: State, inp     :: Channel, outctl :: Ctl       )
out ( value' :: State, inctl  :: Ctl ,      outp    :: Channel )
...

constant RingSize =   16;
```

```
instantiate Track as Ring0;
instantiate Track as Ring*(RingSize-1);

instantiate Track as Bypass*4;

template SyncMerge
in  ( value  :: State, in1,    in2    :: Channel, outctl :: Ctl       )
out ( value' :: State, in1ctl, in2ctl :: Ctl,     outp   :: Channel   )
...

template SyncSplit
in  ( value  :: State, inp    :: Channel,  out1ctl, out2ctl :: Ctl      )
out ( value' :: State, inctl :: Ctl ,      out1,    out2    :: Channel  )
...

instantiate SyncMerge as Join;
instantiate SyncSplit as Fork;
```

3.2 Macro Expressions and Definitions

Macro expressions are quoted using "{"...""}", and have integer types. They may appear as basic expressions, be used to qualify identifiers in any layer, or used in positions requiring an integer constant. Basic arithetic, conditional and macro call operations are supported. Parameterised macro definitions are also supported: these can be used in any macro position.

```
macro predR pos = (pos - 1)  mod  RingSize;
macro succR pos = (pos + 1)  mod  RingSize;

constant BeforeFork  = { predR(ForkPos) };
constant AfterFork   = { succR(ForkPos) };
constant BeforeJoin  = { predR(JoinPos) };
constant AfterJoin   = { succR(JoinPos) };
```

3.3 Wiring Macros

Wiring macros are introduced by associating a wiring definition with a name and set of parameter names. These parameters textually substitute the corresponding concrete links. For example,

```
wire Track ( this, prev, next ) =
    wire {this} ( {this}.value', {prev}.outp,   {next}.inctl )
                ( {this}.value,  {prev}.outctl, {next}.inp   );
```

declares a macro to wire the box called `this` to the `prev` and `next` boxes. The full input/output names are derived textually from the arguments. So for example,

```
wire Track ( Ring{0}, Ring{RingSize-1}, Ring{1} );
```

is equivalent to:

```
wire Ring{0}.value'        to Ring{0}.value;
wire Ring{RingSize-1}.outp to Ring{0}.inp;
wire Ring{1}.inctl         to Ring{0}.outctl;
wire Ring{0}.value'        to Ring{0}.value;
wire Ring{0}.inctl         to Ring{RingSize-1}.outctl;
wire Ring{0}.outp          to Ring{RingSize-1}.inp;
```

Since the `Fork` and `Join` boxes reuse the same names for equivalent inputs/outputs, the `Track` wiring macro can also be used to wire general `Ring` boxes to these special boxes. For example,

```
wire Track ( Ring{BeforeFork}, Ring{predR(BeforeFork)}, Fork );
```

3.4 Repetition

In order to further improve abstraction, meta-level declarations can be repeated under the control of a variable (optionally omitting certain values). The repetition variable may be used within the wiring declation (enclosed within braces), where it takes on each iterated value. For example,

```
for i =  0 to RingSize-1 except (ForkPos, JoinPos)
   instantiate Track as Ring{i};

for i = 1 to RingSize-2
  except ( ForkPos, JoinPos, BeforeFork, AfterFork, BeforeJoin, AfterJoin )
  wire Track ( Ring{i}, Ring{predR(i)}, Ring{succR(i)} );
```

generates `RingSize` boxes from the `Track` template, and wires them in a ring, leaving holes for the `Fork` and `Join` boxes. Note the use of the loop variable `i` within the macros. Loops may be nested as required.

4 Exceptions, Timing and Devices

4.1 Exceptions

Exceptions are declared by `exception <id> <type>` within declarations, raised by `raise <id> <expr>` within expressions and handled by boxes. In order to ensure tight bounds on exception handling costs and to allow a simple implementation, exception handlers are not permitted within expressions. Consequently, we avoid the hard-to-cost chain of dynamic exception handlers that can arise when using non-real-time languages such as standard Java or Concurrent Haskell.

The following (trivial) example shows how an exception that is raised in a function is handled by the calling box. Since each Hume process instantiates one box, the exception handler is fixed at the start of each process and can therefore be called directly. A static analysis is used to ensure that all exceptions that could be raised are handled by each box.

```
exception Div0 :: string 10

f n = if n == 0 then raise Div0 "f" else (n / 0) as string 10;

box example in  (c :: char) out (v :: string 29) handles Div0
unfair n  ->  f n
handle Div0 x -> "Divide by zero in: " ++ x;
```

4.2 Timing

Hume uses a hybrid static/dynamic approach to modelling time. Expressions and boxes are costed statically using the defined time analysis. This may be inadequate to ensure the required time Dynamic time requirements can be introduced at two levels: inside expressions, the `within` construct is used to limit the time that can be taken by an expression. Where an expression would exceed this time, a `timeout` exception is raised. A fixed time exception result is then used instead. In this way we can provide a hard guarantee on execution time. At the coordination level, timeouts can be specified on both input and output wires, and on devices. Such timeouts are also handled through the exception handler mechanism at the box level. This allows hard real-time timing constraints to be expressed, such as a requirement to read a given input within a stated time of it being produced.

```
box b  in ( v :: int 32 ) out ( v' :: int 32 )
unfair x -> complex_fn x within 20ns
handle Timeout -> 0;
```

4.3 Device Declarations

Five kinds of device are supported: buffered streams (files), unbuffered fifo streams, ports, memory-mapped devices, and interrupts. Each device has a directionality (which can only be input for interrupts), an operating system designator, and an optional time specification. The latter can be used to enable periodic scheduling or to ensure that critical events are not missed. Devices may be wired to box inputs or outputs using normal wiring declarations.

For example, an operation that reads a mouse periodically, returning true if the mouse button is down could be specified as:

```
exception Timeout_mouseport;
port mouseport from "/dev/mouse" within 1ms raising Timeout_mouseport;

box readmouse in ( mousein :: bool ) out ( clicked :: bool )
unfair m -> m;
handle Timeout_mouseport -> false;

wire mouseport to readmouse.mousein;
wire readmouse.clicked to mouseuser.mouse;
```

5 Modelling Space Costs

A major goal of the Hume project is to provide good cost models for Hume programs. We have defined a simple space cost model for FSM-Hume that predicts upper bound stack and space usage with respect to the prototype Hume Abstract Machine (pHAM) [13]. 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.

5.1 Space Cost Rules

Figure 5 gives cost rules for a representative subset of FSM-Hume expressions, based on an operational interpretation of the pHAM 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.

The heap cost of a standard integer is given by $\mathcal{H}_{int32}$ (rule 1), 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 2). Each evaluated argument is pushed on the stack before the function is applied, and this must be taken into account when calculating the maximum stack usage. The cost of building a new data constructor value such as a user-defined constructed type (rule 3) 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 $\mathcal{H}_{con}$ are added to represent the costs of tag and size fields. The heap

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

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

...

$$(2)\quad \frac{\mathrm{E}\ (\text{var}) \ = \ \langle h, s\rangle \qquad \forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} \text{var exp}_1 \ \ldots\ \text{exp}_n \Rightarrow \sum_{i=1}^{n} h_i + h,\ \overset{n}{\underset{i=1}{max}}\ (s_i + (i-1)) + s}$$

$$(3)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} \text{con exp}_1 \ \ldots\ \text{exp}_n \Rightarrow \sum_{i=1}^{n} h_i + n + \mathcal{H}_{con},\ \overset{n}{\underset{i=1}{max}}\ (s_i + (i-1))}$$

$$(4)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_1 \Rightarrow h_1, s_1 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_2 \Rightarrow h_2, s_2 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \text{exp}_3 \Rightarrow h_3, s_3}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{if}\ \text{exp}_1\ \texttt{then}\ \text{exp}_2\ \texttt{else}\ \text{exp}_3 \Rightarrow h_1 + max(h_2, h_3), max(s_1, s_2, s_3)}$$

$$(5)\quad \frac{\mathrm{E} \overset{\text{decl}}{\vdash} \text{decls} \Rightarrow h_d, s_d, s'_d, \mathrm{E'} \qquad \mathrm{E'} \overset{\text{space}}{\vdash} \text{exp} \Rightarrow h_e, s_e}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{let}\ \text{decls}\ \texttt{in}\ \text{exp} \Rightarrow h_d + h_e, max(s_d, s'_d + s_e)}$$

Fig. 5. Space cost axioms for expressions

usage of a conditional (rule 4) 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 5) 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. The body of the let-expression is costed in the context of this extended environment.

6 The Mine Drainage Control System

In order to illustrate cost modelling and the use of real-time features in Hume, we will use a simple control application with strong real-time requirements, a simplified pump control system, which is to be used to drain water from a mine shaft. This application has previously been studied in the context of a number of other languages, including Ada with real-time extensions [8]. 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.

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

6.1 Space Analysis of the Mine Drainage Example

The cost rules specified in Section 5 have been extended to cover all of FSM-Hume [13] and implemented as a 200-line Haskell [21] module which is integrated into the `pHAM` compiler. Figure 6 reports predicted and actual maximum stack and heap usage for each box for 12,000,000 iterations of the box scheduler under the prototype Hume Abstract Machine Interpreter.

The results in Figure 6 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. All three boxes are relatively complex with many alternative choices (for which the worst case must be assumed even if it is not used). Moreover, since these boxes are asynchronous, they will frequently become active when only a few inputs are available. For correctness, it is necessary to reserve space for the unavailable inputs, since they could be present in other dynamic contexts. Finally, the *logger* function makes extensive use of string append. Since the size of the result will vary dynamically in some cases and a conservative estimate has been used, the heap usage will consequently have been overestimated slightly. 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, we conclude that this represents a good static estimate of dynamic space usage.

In order to verify that the `pHAM` 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 microkernel 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 15μs worst-case thread context-switch time for realtime threads.

Our measurements 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 62KB. RTLinux itself accounts for 34.4KB of this total. Since little use is made of RTLinux facilities, and there is scope for reducing the sizes of the prototype abstract machine runtime code, 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. The total dynamic memory usage for the pump application amounts to approximately 9KB on RTLinux, comprising 2KB for C stack and runtime initialisation, 4KB for process data structures (including runtime statistics monitoring accounting for about 2KB of this figure), and 3KB for application stack and heap space as costed above.

6.2 Timing Behaviour under RTLinux

In order to confirm that our system can also meet real-time requirements, we have run the mine drainage control system continuously for a period of about 6 minutes under RTLinux on the same 1GHz Pentium III processor (effectively locking out all Linux processes during this period). At this point, the simulation has run to completion. Clock timings have been taken using the RTLinux system clock, which is accurate to the nanosecond level. The primary real-time constraint on the mine drainage control system is that the methane alarm request generated by the `environ` box is serviced within 3ms. In fact, we have measured this delay to be approx. 150μs (20 times faster than required). Moreover, over the 6 minute time period, the *maximum delay* in servicing *any* wire is approximately 2.2ms (for the simulated operator input, which is often locked out

	complete program		pump process	
	(loc)	(object code – bytes)	(loc)	(object code – bytes)
Ada	?	?	206	12045
Java	867	18136	234	7360
Hume	251	7991	51	2003

Fig. 7. Source and object code sizes for the mine drainage control system (Linux)

by more important and therefore higher priority inputs). The maximum delay on any other wire is less than 1.1ms (`environ.meth_reply` in the `pump` box). Clearly, it is necessary to adjust for the lower speed processors found in typical embedded systems, but these figures demonstrate that it is possible to meet hard real-time constraints *in practice* using the Hume abstract machine implementation given suitable hardware. Moreover, in order to improve confidence in its correctness, the implementation deliberately follows the formal translation rules precisely, without optimisation. We estimate that a factor of 2-3 performance improvement can be gained through bytecode optimisations, for example, and that significantly better performance can be achieved using threaded interpretation.

6.3 Comparison with Ada and Java implementations

The Ada implementation is unfortunately restricted to the pump process, and we have therefore been unable to execute it or to provide code size comparisons with the complete Hume application. In order to provide a good basis for comparison, we have therefore constructed a Java implementation for the full system. We have used the Hume implementation to guide the Java design including the correct use of threads, but have used normal Java features including iterative loop constructs, assignment etc. as would be expected. This should not fundamentally change the language basis of comparison [7] but allows us to conduct a direct comparison of the quality of the Hume bytecode against that of the equivalent Java. The advantage of a bytecode representation over machine code is compactness, an attractive property for embedded systems. The disadvantage is that there is usually a high performance cost.

Figure 7 shows source and object code sizes for the three languages. We used the `gnat` Ada compiler, and Sun's `javac` Java compiler on Linux, choosing options to minimise object code size. The Hume bytecode compiler is a small refinement of the RTLinux version, and the code size is therefore smaller than reported earlier (8KB versus 20KB). The Hume program is both considerably shorter, and produces smaller object code than either the Java or Ada programs. This reflects the use of high-level concurrency features in Hume, and the higher-level (and thus more compact) bytecode format.

It might be expected that use of a lower-level abstract machine code as in the `JVM` would yield performance advantages over the higher-level code used in the pHAM. However, we have found that the performance of the pHAM is consistently 9-12 times that of the standard JVM on various testbeds, including simple tests such as loop iteration. This is exacerbated by conventional embedded implementations of Java such as Sun's KVM, whose performance is only 30%-80% that of the standard machine [23]. Partial explanations for this difference are that fewer abstract machine instructions are executed by the pHAM, there is no runtime type checking in Hume, the JVM performs dynamic consistency checking, garbage collection is trivial in the pHAM, and that Java uses dynamic method dispatch rather than static function call. The conventional answer to performance problems in Java is to use more sophisticated compilation techniques such as just-in-time compilation. However, such techniques also carry a fixed runtime space penalty, which may render them unsuitable for use in embedded systems. For example, Shaylor has implemented a small-footprint just-in-time compiler based on the KVM [24], which speeds execution by a factor of 5.7-10.7 over the KVM. This is still rather slower than the interpreted pHAM implementation. Moreover, the code footprint for this RTS is 60K bytes on the ARM, compared with less than 30K bytes for the pHAM on RTLinux. We conclude that the basic implementation of the pHAM is highly competitive with current Java virtual machine implementations for embedded systems.

7 Related Work

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [22]. 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 [3, 28, 11, **?**], there has been work on using functional notations for hardware design (essentially at the HW-Hume level) [14, 9, 20], and there has been much recent theoretical interest both in the problems associated with costing functional languages [22, 16, 10, 26] and in bounding space/time usage [17, 25, **?**], including work on predicting heap and stack memory usage [27]. There is, however, a considerable gap between this work and current implementation practice as exemplified by e.g. the Glasgow Haskell Compiler (GHC) or Standard ML of New Jersey, where space and heap consumption is typically measured in tens of megabytes, and is not guaranteed to be bounded. Even the Gofer implementation that is recognised for its good memory usage requires 640KB to run effectively, an order of magnitude greater than the pHAM.

In a wider framework, two extreme approaches to real-time language design are exemplified by SPARK Ada [5] and the real-time specification for Java (RTSJ) [6]. The former epitomises the idea of language design by elimination of unwanted behaviour from a general-purpose language, including concurrency. The remaining behaviour is guaranteed by strong formal models. In contrast, the latter provides specialised runtime and library support for real-time systems work, but makes no absolute performance guarantees. Thus, SPARK Ada provides a minimal, highly controlled environment for real-time programming emphasising *correctness by construction* [1], whilst Real-Time Java provides a much more expressible, but less controlled environment, without formal guarantees. Our objective with the Hume design is to maintain correctness whilst providing high levels of expressibility.

8 Conclusions and Further Work

This paper has introduced the novel Hume language, a concurrent functional language aimed at resource-limited systems such as the real-time embedded systems domain. We achieve the combination of a high level of programming abstraction with strong properties included bounded time and space behaviour through synthesising recent advances in theoretical computer science into a coherent pragmatic framework. Our empirical results suggest that hard real-time functional programming may be highly practical, provided that good design and implementation practice is adopted.

As shown here, the use of box templates and wiring macros has proved to be a good abstraction mechanism. We are now investigating structured forms of template (a "box calculus") to capture higher levels of process and wiring abstraction. Finally, we are in the process of extending our cost models to cover higher-order polymorphic functions and primitive recursion (i.e. HO-Hume and PR-Hume). Once complete, these cost models will be adapted to also cover concrete time costs for specific architectures.

References

1. P. Amey, "Correctness by Construction: Better can also be Cheaper", *CrossTalk: the Journal of Defense Software Engineering*, March 2002, pp. 24–28.
2. K.R. Apt and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, 2nd Edition, Springer Verlag, 1997.
3. J. Armstrong, S.R. Virding, and M.C. Williams, *Concurrent Programming in Erlang*, Prentice-Hall, 1993.
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. J. Barnes, *High Integrity Ada: the Spark Approach*, Addison-Wesley, 1997.
6. G. Bollela et al. *The Real-Time Specification for Java*, Addison-Wesley, 2000.
7. B. Brosgol and B. Dobbing, "Real-Time Convergence of Ada and Java", *Proc. ACM SIGAda Annual Intl. Conf.*, Bloomington MN, Sept-Oct 2001, pp. 11–26.
8. A. Burns and A. Wellings, *Real-Time Systems and Programming Languages* (Third Edition), Addison Wesley Longman, 2001, Chapter 17, pp. 653–684.
9. K. Claessen and M. Sheeran, "A Tutorial on Lava: a Hardware Description and Verification System", Tech. Rept., Chalmers Univ. of Tech., Sweden, Aug. 2000.

10. R. Burstall, "Inductively Defined Functions in Functional Programming Languages", Dept. of Comp. Sci., Univ. of Edinburgh, ECS-LFCS-87-25, April, 1987.
11. 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.
12. K. Hammond and G.J. Michaelson "The Mine Drainage Control System in Hume", `http://www.hume-lang.org/examples/pump`, January 2003.
13. K. Hammond and G.J. Michaelson "Predictable Space Behaviour in FSM-Hume", To appear in *Proc. 2002 Intl. Workshop on Impl. Functional Langs. (IFL '02)*, Madrid, Spain, Springer-Verlag LNCS 2670, 2003.
14. J. Hawkins and A. Abdallah "An Overview of Systematic Development of Parallel Systems for Reconfigurable Hardware", *Proc. EuroPar 2002*, Paderborn, Germany, Springer-Verlag LNCS 2400, 2002, pp. 615–619.
15. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. "Giotto: A Time-triggered Language for Embedded Programming". *Proc. First International Workshop on Embedded Software*, Springer-Verlag LNCS 2211, 2001, pp. 166-184.
16. 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.
17. 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.
18. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Second Edition, Addison-Wesley, April 1999.
19. 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.
20. J.L. Matthews, J. Launchbury and B. Cook, "Microprocessor Specification in Hawk", *Proc. Intl. Conf. on Computer Science*, Chicago, 1998.
21. 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.
22. A.J. Rebón Portillo, Kevin Hammond, H.-W. Loidl and P. Vasconcelos, "Automatic Size and Time Inference", To appear in *Proc. Intl. Workshop on Impl. of Functional Langs. (IFL 2002)*, Madrid, Spain, Sept. 2002, Springer-Verlag LNCS 2670, 2003.
23. T. Sayeed, N. Shaylor and A. Taivalsaari, "Connected, Limited Device Configuration (CLDC) for the J2ME Platform and the K Virtual Machine (KVM)", *Proc. JavaOne – Sun's Worldwide 2000 Java Developers Conf.*, San Francisco, June 2000.
24. N. Shaylor, "A Just-In-Time Compiler for Memory Constrained Low-Power Devices", *Proc. 2nd Usenix Symposium on Java Virtual Machine Research and Technlog (JVM '02)*, San Francisco, August 2002.
25. M. Tofte and J.-P. Talpin, "Region-based Memory Management", *Information and Control*, **132**(2), 1997, pp. 109–176.
26. D.A. Turner, "Elementary Strong Functional Programming", *Proc. Symp. on Funct. Prog. Langs. in Education*, Springer-Verlag LNCS No. 1022, Dec. 1995.
27. L. Unnikrishnan, S.D. Stoller and Y.A. Liu, "Automatic Accurate Stack Sapce and Heap-Space Analysis for High-Level Languages", *Technical Report TR-538*, Dept. of Comp. Sci., Indiana University, 2000.
28. M. Wallace and C. Runciman, "Extending a Functional Programming System for Embedded Applications", *Software: Practice & Experience*, **25**(1), January 1995.