The Design of Hume: a High-Level Language for the Real-Time Embedded Systems Domain

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 chapter describes the design of Hume: a domain-specific language targeting real-time embedded 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 meta-programming features, whilst seeking to guarantee strong space/time behaviour and maintaining overall determinacy.

1 The Real-Time Embedded Systems Domain

Over the last decade, embedded systems have become a fundamental part of everyday society in the form of systems such as automotive engine control units, mobile telephones, PDAs, GPS receivers, washing machines, DVD players, or digital set-top boxes. In fact, today more than 98% of *all* processors are used in embedded systems [32]. The majority of these processors are relatively slow and primitive designs with small memory capabilities. In 2002, 75% of embedded processors used 8-bit or 16-bit architectures and a total of a few hundreds of bytes is not uncommon in current micro-controller systems.

The restricted capabilities of embedded hardware impose strong requirements on both the space and time behaviour of the corresponding firmware. These limited capabilities are, in turn, a reflection of the the cost sensitivity of typical embedded systems designs: with high production volumes, small differences in unit hardware cost lead to large variations in profit. At the same time software production costs must be kept under control, and time-to-market must be minimised. This is best achieved by employing appropriate levels of programming abstraction.

1.1 Domain-Specific Versus General-Purpose Language Design

Historically, much embedded systems software/firmware was written for specific hardware using native assembler. Rapid increases in software complexity and the need for productivity improvement means that there has been a transition to higher-level *general-purpose* languages such as C/C++, Ada or Java. Despite this, 80% of all embedded systems are delivered late [11], and massive amounts are spent on bug fixes: according to Klocwork, for example, Nortel spends on average $14,000 correcting each bug that is found once a system is deployed. Many of these faults are caused by poor programmer management of memory resources [29], exacerbated by programming at a relatively low level of abstraction. By adopting a *domain-specific* approach to language design rather than adapting an existing *general-purpose* language, it is possible to allow low-level system requirements to guide the design of the required high level language features rather than being required to use existing general-purpose designs. This is the approach we have taken in designing the Hume language, as described here.

Embedding a domain-specific language into a general purpose language such as Haskell [25] to give an *embedded domain-specific language* has a number of advantages: it is possible to build on existing high-quality implementations and to exploit the host language semantics, defining new features only where required. The primary disadvantage of the approach is that there may be a poor fit between the semantics of the host language and the embedded language. This is especially significant for the real-time domain: in order to ensure tight bounds in practice as well as in theory, all language constructs must have a direct and simple translation. Library-based approaches suffer from a similar problem. In order to avoid complex and unwanted language interactions, we have therefore designed Hume as a stand-alone language rather than embedding it into an existing design.

1.2 Language Properties

McDermid identifies a number of essential or desirable properties for a language that is aimed at real-time embedded systems [23].

– *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;
– *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;

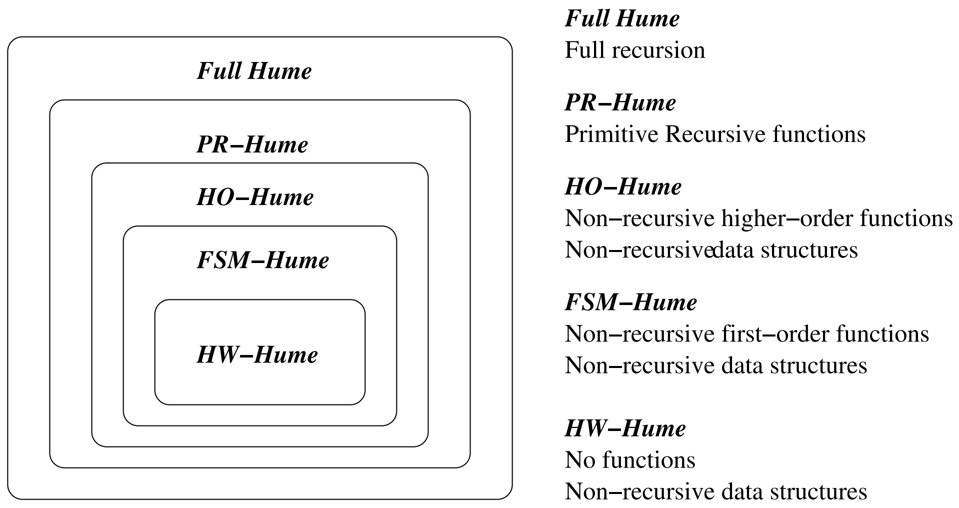
Full Hume
PR–Hume
HO–Hume
FSM–Hume
HW–Hume
Full Hume
Full recursion
PR–Hume
Primitive Recursive functions
HO–Hume
Non–recursive higher–order functions
Non–recursivedata structures
FSM–Hume
Non–recursive first–order functions
Non–recursive data structures
HW–Hume
No functions
Non–recursive data structures

Fig. 1. Hume Design Space

- *correctness* – the language must allow a high degree of confidence that constructed systems meet their formal requirements [1].

Moreover, the language design must incorporate constructs to allow the construction of embedded software, including:

- *exception handling* to deal with runtime error conditions;
- *periodic scheduling* to ensure that real-time constraints are met;
- *interrupts and polling* to deal with connections to external devices.

Finally, we can identify a number of high-level features that assist program construction and reduce overall software costs:

- *automatic memory management* eliminates errors arising from poor manual memory management;
- *strong typing* eliminates a large number of programming errors;
- *higher-order functions* abstract over common patterns of computation;
- *polymorphism* abstracts internal details of data structures;
- *recursion* allows a number of algorithms, especially involving data structures, to be expressed in a natural, and thus robust fashion.

1.3 The Hume Design Model

The Hume language design attempts to maintain the essential properties and features required by the embedded systems domain outlined above (especially for transparent time and space costing) whilst incorporating as high a level of program abstraction as possible. We aim to target applications ranging from simple micro-controllers to complex real-time systems such as Smart Phones. This ambitious goal requires us to incorporate both low-level notions such as interrupt handling, and high-level ones of data structure abstraction etc. Of course such systems will be programmed in widely differing ways, but the language design should accommodate these varying requirements.

We have designed Hume as a three-layer language [15]: an outer (static) declaration/metaprogramming layer, an intermediate coordination layer describing a static layout of dynamic processes and the associated devices, and an inner layer describing each process as a (dynamic) mapping from patterns to expressions. The inner layer is stateless and purely functional. Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework either directly or by altering a more general language (as with e.g. RTSj [7]), 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.

2 Boxes and Coordination

Figure 2 shows the (simplified) syntax of Hume. *Boxes* are the key structuring construct. They are abstractions of finite state machines mapping tuples of inputs to tuples of outputs. The mapping is defined in a functional style, relating input patterns to output expressions. For example, we can define a box that acts like a binary and-gate as:

```
box  band in ( b1, b2 :: bit ) out ( b :: bit)
match  (1,1) -> 1
|      (_,_) -> 0;
```

Although the body of a box is a single function, the *process* defined by a box will iterate indefinitely, repeatedly matching inputs and producing the corresponding outputs, in this case a single stream of bits representing a binary-and of the two input streams. 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 [22], as recently exploited by E-FRP, for example [33]. 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 [15]. 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 [14].

program ::=	$decl_1$; ... ; $decl_n$	$n \geq 1$
decl ::=	*box* | *function* | *datatype* | *exception* | *wire*	
	| *device* | *operation*	
function ::=	var *matches*	
datatype ::=	**data** id α_1 ... α_m = $constr_1$ | ... | $constr_n$	$n \geq 1$
constr ::=	con τ_1 ... τ_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 $expr_1$ ... $expr_n$	$n \geq 0$
	| ($expr_1$, ... , $expr_n$)	$n \geq 2$
	| **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 ... 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*]]	
operation ::=	**operation** *id* **as** *string* :: τ	

Fig. 2. Hume Syntax (Simplified)

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. Ports and streams connect to external devices, such as parallel ports, files etc. For example, we can wire two boxes `band` and `bor` into a static process network linked to the corresponding input and streams as follows:

```
box  band in ( b1, b2 :: bit ) out ( b :: bit) ...
box  bor  in ( b1, b2 :: bit ) out ( b :: bit) ...

stream input1 from "idev1"; stream input2 from "idev2";
stream input3 from "idev3"; stream output to "odev";

wire input1 to band.b1; wire input2 to band.b2;
wire band.b to bor.b1;  wire input3 to bor.b2;
wire bor.b to output;

initial band.b = 1;
```

Note the use of an initialiser to specify that the initial value of the wire connected to `band.b` is `1`.

2.2 Asynchronous Coordination Constructs

The two primary Hume constructs for asynchronous coordination are [15] to *ignore* certain inputs/outputs and to introduce *fair matching*. For example, a template for a fair merge operator can be defined as:

```
template merge ( mtype )
in  ( xs  :: mtype, ys :: mtype)
out ( xys :: mtype)
fair
  (x, *) -> x
| (*, y) -> y;
```

This matches two possible (polymorphic) input streams `xs` and `ys`, choosing fairly between them to produce the single merged output `xys`. Variables `x` and `y` match the corresponding input items in each of the two rules. The `*`-pattern indicates that the corresponding input position should be ignored, that is the pattern matches any input on the corresponding wire, without consuming it. Such a pattern must appear at the top level. Section 3.4 includes an example where an output is ignored.

3 Exceptions, Timing and Devices

3.1 Exceptions

Exceptions are raised in the expression layer and handled by the surrounding box. In order to ensure tight bounds on exception handling costs, 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 Java or Concurrent Haskell [26]. 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. We use Hume's `as` construct to coerce the result of the function `f` to a string containing 10 characters.

```
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
match  n  ->  f n
handle Div0 x -> "Divide by zero in: " ++ x;
```

3.2 Timing

Hume uses a hybrid static/dynamic approach to modelling time. Expressions and boxes are costed statically using a time analysis (as outlined in Section 4 for FSM-Hume programs). Since it is not possible to infer costs for arbitrarily complex full Hume programs, we provide a mechanism for the programmer to introduce time information where this has been derived from other sources, or to assert conditions on time usage that must be satisfied at runtime. 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 )
match  x -> complex_fn x within 20ns
handle Timeout -> 0;
```

3.3 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 (for interrupts, this must be input), 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. For example, we can define an operation to read a mouse periodically, returning true if the mouse button is down, as:

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

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

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

3.4 Interrupt Handling

This section illustrates interrupt handling in Hume using an example adapted from [10], which shows how to write a simple parallel port device driver for Linux kernel version 2.4. Some simplifying assumptions have been made. Like many other operating systems, Linux splits interrupt handlers into two halves: a *top-half* which runs entirely in kernel space, and which must execute in minimal time, and a *bottom-half* which may access user space memory, and which has more relaxed time constraints. For the parallel port, the top-half simply schedules the bottom-half for execution (using the `trig` wire), having first checked that the handler has been properly initialised. The IRQ and other device information are ignored.

```
box pp_int   -- Top half handler
in  ( (irq ::  Int, dev_id :: string, regs :: Regs), active :: boolean )
out ( trig :: boolean, ki :: string )
match
  ( _, false ) -> (*, "pp_int: spurious interrupt\n"),
| ( _, true )  -> ( true, * ) ;

{-# kernel box pp_int -}
```

The bottom-half handler receives a time record produced by the top-half trigger output and the action generated by the parallel port. Based on the internal buffer state (a non-circular buffer represented as a byte array plus head and tail indexes), the internal state is modified and a byte will be read from/written to the parallel port. In either case, a log message is generated.

```
box pp_bottomhalf   -- Bottom half handler
in    ( td :: Time, buffer :: Buffer, action :: Acts )
out   ( buffer' :: Buffer, b :: Byte, log :: String )
```

```
match ( td, (head, tail, buf), Read b ) ->
         if tail < MAX_BUF then
              ( (head, tail+1, update buf tail b), *, log_read td )
         else ( (head, tail, buf), *, "buffer overrun")
|     ( td, (head, tail, buf), Write ) ->
         if   head >= tail then  ( 0, 0, "", * )
         else ( (head+1, tail, buf), buf @ head, *, log_write td );
```

We require an operation to trigger the bottom-half handler. This uses the standard `gettimeofday` Linux function to produce a time record. We also need to log outputs from `pp_int` using the kernel print routine `printk`. This is also specified as an operation. Finally we need to provide the parallel port interrupt routine with information about the IRQ to be used, the name of the handler etc.

The implementation of operations uses a foreign function interface based on that for Haskell [9] to attach a function call to a pseudo-box. The pseudo-box takes one input, marshals it according to the type of the arguments to the external call, executes the external call, and then unmarshals and returns its result (if any). In this way, we use the box mechanism to interface to impure foreign functions, and thereby ensure that box rules are purely functional without resorting to type-based schemes such as Haskell's monadic I/O.

```
operation trigger as "gettimeofday" :: Boolean -> Time;

operation kernel_info as "printk" :: String -> ();

interrupt parport from "(7,pp_int,0)"; -- assumes base at 0x378

wire parport to pp_int.info; wire initialised to pp_int.active;
wire pp_int.trig to trigger.inp; wire pp_int.ki to kernel_info.inp;
```

We now define a box `pp_do_write` to write the output value and the necessary two control bytes to strobe the parallel port. The control bytes are sequenced internally using a state transition, and output is performed only if the parallel port is not busy. We also define abstract ports (`pp_action`, ...) to manage the interaction with actual parallel port.

```
box pp_do_write     -- Write to the parallel port
in  ( cr, stat :: Byte, bout :: Byte, cr2 :: Byte )
out ( bout', cr' :: Byte, cr'':: Byte  )
match
  ( *, SP_SR_BUSY, *, *, * ) -> ( *, *, * ) -- wait until non-busy
| ( *,  *, *,    *,     cr ) -> ( *,    cr & ~SP_CR_STROBE, * )
| ( cr, _, bout, false, *  ) -> ( bout, cr | SP_CR_STROBE,  cr );

port pp_action; port pp_stat; port pp_data; port pp_ctl_in; port pp_ctl_out;
```

Finally, wiring definitions link the bottom-half boxes with the parallel port.

```
wire trigger.outp to pp_bottomhalf.td;
wire pp_bottomhalf.buffer' to pp_bottomhalf.buffer;

wire pp_action to pp_bottomhalf.action;
wire pp_bottomhalf.b  to bottomhalf.buffer;
wire pp_bottomhalf.log to kernel_info;

wire pp_ctl_in to pp_do_write.cr; wire pp_stat to pp_do_write.stat;
wire pp_do_write.bout to pp_data;
wire pp_do_write.cr'  to pp_ctl_out;
wire pp_do_write.cr'' to pp_do_write.cr2;

wire pp_do_write.cr to pp_do_write.cr2;
```

4 Modelling Space Costs

A major goal of the Hume project is to provide good space and time cost models for Hume programs. In the long term, we intend to provide cost models for all levels of Hume up to at least PR-Hume by exploiting emerging theoretical results concerning cost models for recursive programs [16]. We illustrate the practicality of our general approach by describing 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.

4.1 Space Cost Rules

Figure 3 gives cost rules for a representative subset of FSM-Hume expressions, based on an operational interpretation of the `pHAM` implementation. The full set of cost rules is defined elsewhere [14]. We have shown that using this analysis, it is possible to construct programs that possess tightly bounded space behaviour *in practice* for FSM-Hume. 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 independent upper bounds on the stack and heap usage. The result is produced in the context of an environment, E, that maps function names to the heap and stack requirements associated with executing the body of the function, and which 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.

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

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

...

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

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

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

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

Fig. 3. Space cost axioms for expressions

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 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 declarations), 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.

5 Implementations of Hume

We have constructed two prototype implementations of Hume: a proof-of-concept reference interpreter written in the non-strict functional language, Haskell; and a basic abstract machine compiler [13], using the same Haskell front-end, and supported by a small runtime system written in C. We are in the process of producing a more sophisticated abstract machine implementation, and intend to produce fully compiled implementations in the near future. These implementations may be downloaded from `http://www.hume-lang.org`.

The prototype Hume abstract machine compiler (`phamc`) targets a novel abstract machine for Hume supporting concurrency, timeouts, asynchronicity, exceptions and other required constructs, the prototype Hume Abstract Machine (`pHAM`). The pHAM is primarily intended to provide a more realistic target for our work on cost modelling and analysis rather than as a production implementation. The compiler is defined using a formal translation scheme from Hume source to abstract machine code. We have also provided an informal operational semantics for Hume abstract machine instructions. Our long-term intention is to use formal methods to verify the correctness of the compiler with respect to the Hume language semantics, and to verify the soundness of the source-level cost models with respect to an actual implementation. The `phamc` compiler has been ported to two Unix platforms: Red Hat Linux and Mac OS/X. We have also provided an implementation for the real-time operating system RTLinux [3]. The C back-end should prove highly portable to non-Unix platforms.

We have compared the performance of the `pHAM` against that of the most widely used abstract machine implementation: the Java Virtual Machine. A number of new embedded applications (predominantly in the mobile phone sector) are seriously targeting the Java software platform, so this is a fair comparison against an equivalent general purpose approach. Performance results [13] show that the the performance of the `pHAM` implementation is consistently 9-12 times that of the standard `JVM` on various test-beds, including simple tests such as loop iteration. Compared with Sun's KVM for embedded systems [30], we thus estimate execution speed to be 12-20 times; and compared with the just-in-time KVM implementation [31], we estimate performance at 2-4 times. We conjecture that these performance gains arise from the use of a higher-level, more targeted abstract machine design. In the embedded systems domain, we do not need to include dynamic security checks, for example.

We have measured dynamic memory usage for a moderately complex control system [13] in the `pHAM` at less than 9KB of dynamic memory, and total memory usage, including static allocation, code, runtime system and operating system code at less than 62KB (more than half of this is required by the operating system – the `pHAM` implementation requires less than 32KB in total). These figures are approximately 50% of those for the corresponding KVM implementation, and suggest that the `pHAM` implementation should be suitable for fairly small-scale embedded control systems.

We are in the process of constructing a distributed implementation of Hume based on the `pHAM` implementation and mapping boxes to embedded system components. One outstanding technical issue is how to design the scheduling algorithm in such a way as to maintain consistency with the concurrent implementation that we have described here. We also intend to construct native code implementations and to target a wider range of hardware architectures. We do not foresee any serious technical difficulties with either of these activities.

6 Related Work

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [28]. 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. A number of functional languages have, however, looked at *soft* real-time issues (e.g. Erlang [2] or E-FRP [33], there has been work on using functional notations for hardware design (essentially at the HW-Hume level) (e.g. Hydra [24]), and there has been much recent theoretical interest both in the problems associated with costing functional languages (e.g. [28, 20, 21]) and in bounding space/time usage (e.g. [34, 19]).

In a wider framework, two extreme approaches to real-time language design are exemplified by SPARK Ada [4] and the real-time specification for Java (RTSJ) [7]. 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.

6.1 Synchronous Dataflow Languages

In synchronous dataflow languages, unlike Hume, every *action* (whether computation or communication) has a zero time duration. In practice this means that actions must complete before the arrival of the next event to be processed. Communication with the outside world occurs by reacting to external stimuli and by instantaneously emitting responses. Several languages have applied this model to real-time systems control. For example, Signal [5] and Lustre [12] are similar declarative notations, built around the notion of timed sequences of values. Esterel [6] is an imperative notation that can be translated into finite state machines or hardware circuits, and Statecharts [17] uses a visual notation, primarily for design. One obvious deficiency is the lack of expressiveness, notably the absence of recursion and higher-order combinators. Synchronous Kahn networks [8] incorporate higher-order functions and recursion, but lose strong guarantees of resource boundedness.

6.2 Static Analyses for Bounding Space or Time Usage

There has been much recent interest in applying static analysis to issues of bounded time and space, but none is capable of dealing with higher-order, polymorphic and generally recursive function definitions as found in full Hume. For example, region types [34] allow memory cells to be tagged with an allocation *region*, whose scope can be determined statically. When the region is no longer required, all memory associated with that region may be freed without invoking a garbage collector. This is analogous to the use of Hume boxes to scope memory allocations. Hofmann's linearly-typed functional programming language LFPL [18] uses linear types to determine resource usage patterns. First-order LFPL definitions can be computed in bounded space, even in the presence of general recursion. For arbitrary higher-order functions, however, an unbounded stack is required.

Building on earlier work on sized types [20, 28], we have developed an automatic analysis to *infer* the *upper bounds* on evaluation costs for a simple, but representative, functional language with parametric polymorphism, higher-order functions and recursion [35]. Our approach assigns finite costs to a non-trivial subset of primitive recursive definitions, and is *automatic* in producing cost equations without any user intervention, even in the form of type annotations. Obtaining closed-form solutions to the costs of recursive definitions currently requires the use of an external recurrence solver, however.

7 Conclusions and Further Work

This chapter has introduced Hume, a domain-specific language for resource-limited systems such as the real-time embedded systems domain. The language is novel in being built on a combination of finite state machine and λ-calculus concepts. It is also novel in aiming to provide a high level of programming abstraction whilst maintaining good formal properties, including bounded time and space behaviour and provably correct rule-based translation. 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. By taking a domain-specific approach to language design we have thus raised the level of programming abstraction without compromising the properties that are essential to the embedded systems application domain.

A number of important limitations remain to be addressed:

1. space and time cost models must be defined for additional Hume layers including higher-order functions and (primitive) recursion and these must be implemented as static analyses;
2. we need to provide machine-code implementations for a variety of architectures that are used in the real-time embedded systems domain, and to develop realistic demonstrator applications that will explore practical aspects of the Hume design and implementation;
3. more sophisticated scheduling algorithms could improve performance, however, these must be balanced with the need to maintain correctness;
4. no attempt is made to avoid deadlock situations through language constructs: a suitable model checker must be designed and implemented.

Of these, the most important limitation is the development and application of more sophisticated theoretical cost models. Our sized time type-and-effect system is already capable of inferring theoretical costs in terms of reduction steps for higher-order polymorphic definitions [28]. Adapting the system to infer heap, stack and time costs for Hume programs should be technically straightforward. Moreover, we have recently extended this theoretical system to cover primitive recursive definitions. Incorporating this analysis into Hume will go a long way towards our goal of achieving high level real-time programming.

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. J. Armstrong, S.R. Virding, and M.C. Williams, *Concurrent Programming in Erlang*, Prentice-Hall, 1993.
3. M. Barabanov, *A Linux-based Real-Time Operating System*, M.S. Thesis, Dept. of Comp. Sci., New Mexico Institute of Mining and Technology, June 1997.
4. J. Barnes, *High Integrity Ada: the Spark Approach*, Addison-Wesley, 1997.
5. A. Benveniste and P.L. Guernic, "Synchronous Programming with Events and Relations: the Signal Language and its Semantics", *Science of Computer Programming*, **16**, 1991, pp. 103–149.

6. G. Berry. “The Foundations of Esterel”, In *Proof, Language, and Interaction*. MIT Press, 2000.
7. G. Bollela et al. *The Real-Time Specification for Java*, Addison-Wesley, 2000.
8. P. Caspi and M. Pouzet. “Synchronous Kahn Networks”, *SIGPLAN Notices* 31(6):226–238, 1996.
9. M. Chakravarty (ed.), S.O. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S.L. Peyton Jones, A. Reid, M. Wallace and M. Weber, “The Haskell 98 Foreign Function Interface 1.0”, `http://www.cse.unsw.edu.au/~chak/haskell/ffi`, December, 2003.
10. J. Corbet and A. Rubini, “Linux Device Drivers”, 2nd Edition, O’Reilly, 2001.
11. The Ganssle Group. Perfecting the Art of Building Embedded Systems. `http://www.ganssle.com`, May 2003.
12. N. Halbwachs, D. Pilaud and F. Ouabdesselam, “Specificying, Programming and Verifying Real-Time Systems using a Synchronous Declarative Language”, in *Automatic Verification Methods for Finite State Systems*, J. Sifakis (ed.), Springer-Verlag, 1990, pp. 213–231.
13. K. Hammond. “An Abstract Machine Implementation for Embedded Systems Applications in Hume”, *Submitted to 2003 Workshop on Implementations of Functional Languages (IFL 2003)*, Edinburgh, 2003.
14. K. Hammond and G.J. Michaelson “Predictable Space Behaviour in FSM-Hume”, *Proc. 2002 Intl. Workshop on Impl. Functional Langs. (IFL ’02)*, Madrid, Spain, Springer-Verlag LNCS 2670, 2003.
15. K. Hammond and G.J. Michaelson, “Hume: a Domain-Specific Language for Real-Time Embedded Systems”, *Proc. Conf. on Generative Programming and Component Engineering (GPCE ’03)*, Springer-Verlag LNCS, 2003.
16. K. Hammond, H.-W. Loidl, A.J. Rebón Portillo and P. Vasconcelos, “A Type-and-Effect System for Determining Time and Space Bounds of Recursive Functional Programs”, *In Preparation*, 2003.
17. D. Harel, “Statecharts: a Visual Formalism for Complex Systems”, *Science of Computer Programming*, **8**, 1987, pp. 231–274.
18. M. Hofmann. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
19. M. Hofmann and S. Jost, “Static Prediction of Heap Space Usage for First-Order Functional Programs”, *Proc. POPL’03 — Symposium on Principles of Programming Languages*, New Orleans, LA, USA, January 2003. ACM Press.
20. 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.
21. 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)*, aris, France, pp. 70–81, 1999.
22. S.D. Johnson, *Synthesis of Digital Designs from Recursive Equations*, MIT Press, 1984, ISBN 0-262-10029-0.
23. 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.
24. J.T. O’Donnell, “The Hydra Hardware Description Language”, *This proc.*, 2003.
25. 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.

26. S.L. Peyton Jones, A.D. Gordon and S.O. Finne "Concurrent Haskell", *Proc. ACM Symp. on Princ. of Prog. Langs.*, St Petersburg Beach, Fl., Jan. 1996, pp. 295–308.
27. R. Pointon, "A Rate Analysis for Hume", *In preparation*, Heriot-Watt University, 2004.
28. A.J. Rebón Portillo, Kevin Hammond, H.-W. Loidl and P. Vasconcelos, "Automatic Size and Time Inference", *Proc. Intl. Workshop on Impl. of Functional Langs. (IFL 2002)*, Madrid, Spain, Sept. 2002, Springer-Verlag LNCS 2670, 2003.
29. M. Sakkinen. "The Darker Side of C++ Revisited", *Technical Report 1993-I-13*, `http://www.kcl.ac.uk/kis/support/cit//fortran/cpp/dark-cpl.ps`, 1993.
30. 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.
31. 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.
32. E. Schoitsch. "Embedded Systems – Introduction", *ERCIM News*, **52**:10–11, 2003.
33. W.Taha, "Event-Driven FRP", *Proc. ACM Symp. on Practical Applications of Declarative Languages (PADL '02)*, 2002.
34. M. Tofte and J.-P. Talpin, "Region-based Memory Management", *Information and Control*, **132**(2), 1997, pp. 109–176.
35. P. Vasconcelos and K. Hammond. "Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs", *Submitted to 2003 Workshop on Implementations of Functional Languages (IFL 2003)*, Edinburgh, 2003.