Programming Reactive Systems in Hume (Extended Abstract)

Kevin Hammond[1], Greg J. Michaelson[2], and Meng Sun[1]

[1] School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SS. kh@dcs.st-and.ac.uk
[2] School of Mathematics and Computer Science, Heriot-Watt University, Riccarton, Edinburgh, Scotland. greg@macs.hw.ac.uk

Abstract

Hume is a domain-specific programming language based on a combination of finite state machine and functional programming constructs. Hume is unusual in being structured as a series of levels, each of which exposes different machine properties, and from which specific cost models can be derived. In this paper we will explore how Hume can be used to program reactive systems using a number of examples. We demonstrate that bounded programs can be constructed using the cost modelling technology we have previously developed.

1 INTRODUCTION

Hume is a domain-specific programming language targetting the real-time embedded systems domain. It is unusual in incorporating ideas from both *finite state automata* and *functional programming*. Hume combines a very high level of expressivity (including constructs such as automatic memory management, implicit concurrency and recursion) with very good low-level capability (including real-time constructs, access to ports, interrupts etc., and foreign-function interfacing).

While Hume is primarily aimed at real-time embedded systems (in a fairly broad sense, covering small-scale control systems up to mobile phones or similar devices), the language design is sufficiently general that it could, in principle, be used as the basis for hardware/software co-design or even for general-purpose programming.

In this paper, we will introduce a number of new examples showing how *reactive* programs can be written in Hume. The programs have been costed using previously-described stack-and-heap space costing technology [6].

1.1 Overview of the Hume Design

Notationally, Hume combines finite-state automata for programming processes (the *coordination* layer), with (purely) functional programming for programming expressions (the *expression layer*). The coordination layer supports *coinductive* programming on potentially infinite streams and other structures, while the expression layer supports finite *inductive* programming on data and program structures. Finiteness is guaranteed by supplementing careful language construction with static program analysis.

1.2 Finite State Automata

Finite state automata provide a basis for constructing simple state-changing systems. They may also be used to give a natural model of concurrency. Finite state automata comprise a set of linked states, with transitions showing the changes from one state to another based on the inputs that are received. Because pure finite-state-automata are so simple, there is a natural fit between finite-state-automata and hardware, and it is easy to show that such automata have bounded time and space costs. Some low-level programming langauges, such as Esterel [4] also use an essentially pure finite-state approach, and mechanistic systems such as lexers and parsers are also commonly automaton-based. The primary deficiencies of finite state approaches are:

- there may be a huge number of states for even fairly simple software programs;
- it may be necessary to decompose programming problems into very low-level abstractions;
- there are theoretical limitations on the classes of problem that can be solved using automata.

While each state may be simple in itself, the explosion in the number of states means even small, simple programs can be too complex to understand easily. Moreover, it can be cumbersome to write simple functions and other operations as automata. Hume attempts to systematically address these objections as follows:

- finite-state automata are used to structure concurrency only – computations are written using conventional programming notations;
- high-level programming notations are used to collapse complex sets into a few manageable automata;
- by combining high-level programming notations with automata the class of problem that can be solved is extended to cover all *computable* problems.

In Hume, programs are formed from concurrent *boxes*, which respond to inputs and produce outputs on one or more *wires*. Computations within boxes are described using normal high-level programming notations rather than as automata. While there is a broad analogy with the use of high-level *objects* as concurrent agents, and object-based designs can thus be exploited at the high-level, the analogy should not be stretched too far – boxes are much more structured than objects, in particular in restricting communication patterns, in relating inputs directly to outputs, and in providing a static rather than dynamic process network. This discipline allows us to automate testing, to demonstrate deadlock-freedom using an automatic analysis, and to enforce strong bounds on program costs.

1.3 Functional Programming

Purely functional programming provides a good basis for constructing software with excellent formal properties. Because functional programs are both *declarative* and deterministic, they are much easier to reason about using mathematically-derived techniques than either object-based or imperative approaches [8]. In fact, many advanced compiler optimisations work on an internal representation that is purely functional, and compilers can go to great lengths to isolate parts that are not purely functional, so that they can take advantage of these techniques. However, to obtain these advantages:

1. programmers must be trained to exploit high-level functional abstractions, which some programmers find difficult;
2. there may be a poor match between program and machine implementation, making it difficult to construct software that must access low-level features; and
3. performance can be significantly worse than the best imperative implementations (though performance may be better than say C++ or even Fortran in some cases [9]).

Hume attempts to systematically address these objections in the following ways:

1. finite-state automata are a natural way to decompose concurrent programs, and provide analogies to the higher levels of object-based program decomposition, without the strictures of overly low-level objects;
2. state changes are made explicit through finite-state automata, and explicit operating system interactions;
3. it is possible to provide a straightforward translation from Hume source to the corresponding machine code, whether direct or through an intermediate abstract machine; and
4. Hume has been designed to allow the best compiler optimisations to be exploited – hindrances to compiler optimisation have been designed out as far as possible – time performance is roughly 10 times that for Sun's embedded KVM, and dynamic space usage is both guaranteed to be bounded and a fraction of that required by Java or C++.

The combination of finite state automata with functional programming therefore gives a powerful programming basis without sacrificing crucial low-level capabilities.

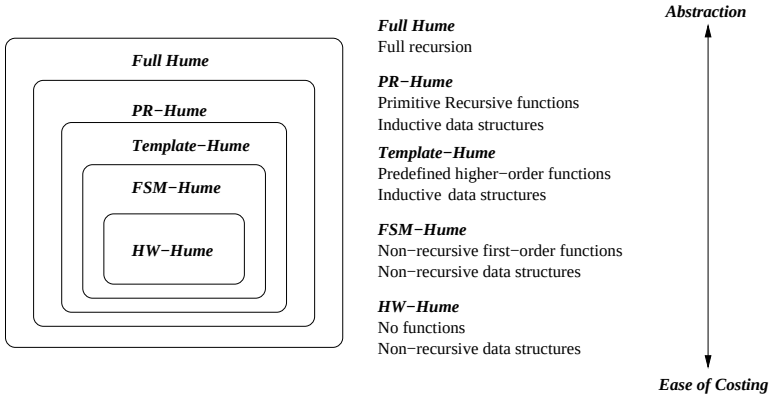

FIGURE 1. Hume Levels

1.4 Hume Levels

Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework either directly or by altering/emasculating the language to a greater or lesser extent (as with e.g. RTSj [3] or SPARK Ada [2]), our approach is to design Hume in such a way that we are certain that formal cost models and proofs can be constructed. We identify the series of overlapping Hume language levels shown in Figure 1, where each level adds expressibility to the expression semantics, but either loses some desirable behavioural property or increases the technical difficulty of providing formal correctness/cost models.

HW-Hume: a hardware description language — capable of describing both synchronous and asynchronous hardware circuits, with arithmetic and other primitive operations, but with no functions and only non-recursive data structures;

FSM-Hume: a hardware/software language — HW-Hume plus first-order functions, conditionals and local definitions;

Template-Hume: a language for template-based programmimng — FSM-Hume plus predefined higher-order polymorphic functions, but no recursion;

PR-Hume: a fairly powerful language — HO-Hume plus primitive recursion in both functions and data structures;

Full-Hume: a Turing-complete language — PR-Hume plus unrestricted recursion in both functions and data structures.

This paper focuses on programming in the FSM-Hume level.

1.5 Boxes and Wires

A Hume program consists of a number of autonomous but interconnected *boxes*, representing individual FSMs linked by single-buffered *wires*, representing the links between those FMSs. Each box's behaviour is described by a sequence of *match rules* associating input patterns with output tuples.

For example, we can define or- and and-gates as:

```
box or
in  (b1,b2::Bit)
out (res::Bit)
match
  (0,0) -> 0
| (_,_) -> 1;

box and
in  (b1,b2::Bit)
out (res::Bit)
match
  (1,1) -> 1
| (_,_) -> 0;
```

When a program runs, all boxes are checked to determine whether they may be scheduled. Each runnable box is then executed in turn. The inputs for each box are matched against each of its patterns, and the first matching rule selected for execution. The right-hand-side of the rule is evaluated and the resulting values placed on the outputs. If one or more outputs from a previous step were not consumed then the box *blocks* until the output is consumed. Otherwise the box completes and may be rescheduled in the next scheduling cycle. This continues repeatedly.

The corresponding wiring specification connects the three system inputs to the two inputs of the or-gate and one of the and-gate; the output of the or-gate to the other input of the and-gate; and the output of the and-gate to the system output.

```
wire input1  to or.b1;
wire input2  to or.b2;
wire or.res  to and.b1;
wire input3  to and.b2;
wire and.res to output;
```

2 HUME PROGRAMMING EXAMPLES

This section will describe a number of reactive programs written using the FSM-Hume level. In the extended abstract we consider only one simple example.

The drinks vending machine is a classic reactive systems programming example with variations implemented in a number of systems including CSP and

CCS. The Hume implementation uses a number of enumerated datatype definitions. `Coins` can be either `Nickel`s or `Dime`s, the `Buttons` pressed by the user can either request a coffee (`BCoffee`), a tea (`BTea`) or a refund (`BCancel`), and the machine will vend `Drinks` which are either `Coffee` or `Tea`

```
data Coins =  Nickel | Dime;
data Drinks = Coffee | Tea;
data Buttons = BCoffee | BTea | BCancel;
```

The system comprises three concurrent boxes (representing finite automata) wired together into a concurrent system of three static processes: `inp` parses user input and converts it into values that are passed to the `coffee` box; this acts on the input, passing results to the `outp` box, which converts the results into output strings.

The heart of the system is the `coffee` box. This has three possible inputs: a coin placed into the slot by the customer, a button pressed by the customer and a value representing the value of coins that have been paid into the machine so far (the cash level). It has three possible outputs: a dispensed drink, a revised cash level and a refund. The cash level is maintained by mapping the output value from one box iteration to the input value in the next iteration.

If a nickel or a dime is paid into the machine through the coin slot (rules 1 and 2) then the current value (`v`) is incremented by the value of the coin (5 or 10 cents, respectively). These rules use the `*` pattern to match an arbitrary button input without consuming it (i.e. without removing it from the corresponding input stream buffer), and the ignored value (also `*`) indicating that no value is to be output on the drink and refund wires.

If a coffee or tea is requested (rules 3 and 4), then the vend function is called to dispense the corresponding drink. This checks the current cash level aganst the price of the requested drink and if enough cash has been paid into the machine dispenses the drink and decrements the current cash level . Otherwise, no drink is dispensed (represented by an `*`) and the current cash value remains unchanged.

Finally, rule 5 refunds all money that has been paid into the machine.

```
-- coffee vending box

box coffee
in  ( coin :: Coins,   button :: Buttons, value :: Int  )
out ( drink :: Drinks, value' :: Int,     refund :: Int )
match
    ( Nickel, *,       v )  ->  ( *, v + 5, * )    -- (1)
  | ( Dime,   *,       v )  ->  ( *, v + 10, * )   -- (2)
  | ( *,      BCoffee, v )  ->  vend Coffee 10 v   -- (3)
  | ( *,      BTea,    v )  ->  vend Tea 5 v       -- (4)
  | ( *,      BCancel, v )  ->  ( *, 0, v )        -- (5)
  ;
```

```
vend drink cost v = if v >= cost then ( drink, v-cost, * )
                                 else ( *,     v,      * );
```

This box definition clearly shows how programs are separated into coordination and expression layers: all reactivity (coinduction) is handled at the box (automaton) level; patterns transcend the layers: with asynchronous pattern matches used to select a rule that matches the available inputs; purely functional (inductive) expressions are used to produce results. Finally polymorphic `*` values indicate outputs that should be ignored.

The remaining boxes have straightforward definitions. The `inp` box converts text inputs into either coin inputs or button presses. Unrecognised inputs are ignored. Finally the `outp` box matches either a drink or refund action and

```
-- input handling box

box inp
in ( c :: char )
out ( coin :: Coins, button :: Buttons )
match
    'N' -> ( Nickel, * )
  | 'D' -> ( Dime,   * )
  | 'C' -> ( *,      BCoffee )
  | 'T' -> ( *,      BTea )
  | 'X' -> ( *,      BCancel )
  | _   -> ( *,      * )
  ;

-- output handling box

box outp
in  ( drink :: Drinks, refund :: Int )
out ( s :: string 15)
match
  ( d, * ) -> "Vending " ++ showdrink d ++ "\n"
| ( *, r ) -> "Refunding " ++ r as string ++ "\n"
;

showdrink Coffee = "Coffee";
showdrink Tea = "Tea";
```

Finally we must specify how the boxes are wired into a static process network.

```
stream stdout      to   "std_out";
stream stdin       from "std_in";

wire stdin         to   inp.c;
wire inp.coin      to   coffee.coin;
wire inp.button    to   coffee.button;
```

```
wire cofee.drink   to   outp.drink;
wire coffee.value' to   coffee.value initially 0;
wire coffee.refund to   outp.refund.
wire outp.s        to   stdout;
```

2.1 Space Usage Results

Table 1 shows results obtained from running several reactive FSM-Hume programs under the prototype Hume Abstract Machine (pHAM). H_A represents actual heap usage, H_P represents predicted heap usage, S_A represents actual stack usage and S_P represents predicted heap usage. Predictions are obtained from our source-level static analysis for FSM-Hume programs [7]. All figures are given in terms of words of memory (where a word occupies 4 bytes in the current implementation).

In each case, we have broken figures down to the individual box level and also provided figures for the memory used by the wires. In this way we can calculate the total dynamic memory requirement for the abstract machine (having also included a small amount of per-box overhead for the program counter and other information – 23 words per box).

Overall space usage is 2.7KB for the digital watch, 3.0KB for the railroad crossing simulation, 3.3KB for the mine drainage controller, 0.8KB for the coffee machine simulation and 389KB for the vehicle simulation. The relatively large space requirement of the latter is explained by its manual conversion from a PR-Hume original, and by the use of large data structures for which compiler optimisations have not yet been implemented.

Our results clearly show that it is possible to construct a number of representative representative reactive programs in Hume with good space prediction.

3 IMPLEMENTATION

Implementations of Hume have been produced for Linux, MacOSX, the mobile phone operating system Symbian OS [5], the hard real-time variant of Linux, RTLinux [1] and the Renesas M32-based development board[1]. These implementations are based on a compact high-level bytecode representation, whose code size is about 50% that of Java bytecode and whose performance is signicantly better than that of Java.

The RTLinux and Renesas ports both offer hard real-time performance with high reliability. All implementations offer hard space performance, with the Renesas implementation, for example, requiring less than 16KB for the complete Hume runtime system, bytecodes and all dynamic data needed to implement a real-time version of the classic Simon computer game. Tests on the RTLinux implementation have also shown that high reliability can be obtained under stringent real-time

[1] See `http://www.hume-lang.org` for stable versions. More recent implementations may be available to researchers on request.

conditions. For example, we have run one application continuously for 72 hours with sub-millisecond response times on a 350MHz Pentium II, and with less than 64KB total memory requirement. We are in the process of developing native code compilers and implementations.

4 CONCLUSIONS

This extended abstract has described the use of Hume for programming reactive systems giving one prototypical example, a vending machine example originally inspired by Robin Milner's CCS exemplar. We have given space cost results for a number of reactive programs. In the full paper we will show that it is possible to construct a number of representative representative reactive programs in Hume with good space prediction.

Acknowledgements

This work is generously supported by EPSRC grants GR/R70545 and EP/C001346/1 and by EU Framework VI IST-510255 (EmBounded).

REFERENCES

[1] 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.

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

[3] G. Bollela and et al. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[4] F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[5] Kevin Dixon. Symbian os version 7.0s functional description. Technical report, Symbian Ltd, 2005.

[6] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[7] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. IFL 2002 – Implementation of Functional Languages, Madrid, Spain*, LNCS 2670. Springer-Verlag, 2003, to appear.

[8] R.J.M. Hughes. Why Functional Programming Matters? *The Computer Journal*.

[9] S-B. Scholz. *Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen (in German)*. PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, October 1996.

	H_A	H_P	$H_P - H_A$	S_A	S_P	$S_P - S_A$
Digital Watch						
button	5	5	0	6	6	0
display	254	266	12	24	32	8
separate	49	53	4	28	30	2
watch	52	66	8	29	33	4
wires	110	110	0	–	–	–
Total	470	500	30	87	101	14
Railroad Crossing Simulation						
control	25	25	0	15	15	0
log	286	308	22	26	28	2
separate	45	45	0	42	42	0
speed	13	15	2	18	21	3
trainA/B	40	40	0	14	15	1
wires	29	32	3	–	–	0
Total	478	505	27	127	134	7
Vehicle Tracking Simulation						
control	57	60	3	36	42	6
env	49099	49580	581	128	139	11
vehicle	49164	49648	484	132	142	10
wires	120	126	6	–	–	–
Total	98440	99360	920	296	323	27
Coffee Vending Machine						
coffee	20	23	3	21	24	3
inp	10	10	0	7	7	0
outp	20	32	12	14	17	3
wires	15	15	0	0	0	0
Total	65	80	15	42	48	6
Mine Drainage Controller						
airflow	16	16	0	12	14	2
logger	103	183	80	25	30	6
methane	16	16	0	12	14	2
operator	29	28	9	17	22	5
pump	42	51	9	17	22	5
supervisor	29	29	0	24	28	4
water	54	54	0	24	29	5
wires	80	94	14	–	–	–
Total	425	483	58	162	166	4

TABLE 1. Stack and Heap Costs for a Number of FSM-Hume Programs