



IST-510255

EmBounded

Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D23 (WP7c): Hume to HAM Translator

Due date of deliverable: July 2007
Actual submission date: January 2008

Start date of project: 1st March 2005

Duration: 48 months

Lead contractor: Heriot-Watt University

Revision: 1.13

Purpose: The purpose of this deliverable is to produce a Hume to HAM translation tool which conforms to the Hume formal semantics, Deliverable D12 (WP7a) [Jos06] and operates using the formal translation rules, Deliverable D3 (WP7b) [LJH06]. It is also required to provide support for the chosen target architectures (i386, PPC and M32C) via the machine code compiler, Deliverable D24 (WP7c).

Results: The main result is the implementation of a fully tested Hume to HAM translator which supplied as part of this deliverable. Note that the translator is part of the full compiler suite which is also provided by D24 and D25 and shares the same installation media with those deliverables.

Conclusion: We have supported the other tools implemented as part of the EmBounded project by developing the Hume to HAM translator according to the needs of the project. It now conforms to the Hume language definition and provides support for the resource analysis and hardware target platform.

Project co-funded by the European Commission within the 6 th Framework Programme (2002-06)		
Dissemination Level		
PU	Public	*
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential only for members of the consortium (including the Commission Services)	

Hume to HAM Translator

Robert F. Pointon <rpointon@macs.hw.ac.uk>

School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland

Norman Scaife <Norman.Scaife@lasmea.univ-bpclermont.fr>

Laboratoire LASMEA, Blaise Pascal University

Abstract

This deliverable describes how the Hume to HAM translator has been developed to meet the EmBounded project criteria. The project objectives are stated and brief explanations of how these objectives were met are presented. The positioning of this deliverable is discussed in Section 4. The evidence supplied in support of these statements is listed. Finally, we cite academic papers related to this work.

Major Revisions		
Revision	Date	Changes
<i>1.13</i>	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
<i>1.11</i>	27 May 2008	appendices A B on abstract machine design and compilation outline (addressing Review Report Year 3)
<i>1.9</i>	6 Feb. 2008	initial version

Contents

1	Introduction	3
2	Objectives	3
3	Meeting the objectives	3
4	Relation with other deliverables	4
5	Material provided	4
A	Hume Abstract Machine Design	5
A.1	Outline Design	6
A.2	The Hume Abstract Machine Instructions	7
A.2.1	Heap Object Creation and Stack Manipulation (Figure 4).	8
A.2.2	Control Operations (Figure 5).	9
A.2.3	Pattern matching (Figure 6).	9
A.2.4	Exceptions (Figure 8).	10
A.2.5	Thread input/output and rescheduling operations (Figure 9).	10
B	Compilation Scheme	11

1 Introduction

This deliverable, D23, describes the Hume to HAM translator, its conformance to the Hume formal semantics, Deliverable D12 [Jos06], its derivation from the formally proven Hume to HAM translation scheme, Deliverable D03 [LJH06], and its input to the machine code compiler provided by Deliverable D24.

Deliverable D03 describes, formally, the derivation of HAM code from Hume source text using semantics-preserving transformations. Here we present the current implementation of the translator, `phamc`. Internally, this uses the translation scheme described in Deliverable D03.

An experimental version of this translator existed prior to EmBounded but there have been numerous changes, enhancements and extensions during its development and verification as part of the current work.

2 Objectives

This work is aimed towards achieving part of the EmBounded project overall objective 6: *development of underpinning specification, implementation and support environment for the Hume language*.

The specific objectives addressed by this deliverable are:

1. Update the existing experimental translator to reflect recent work on the Hume formal semantics.
2. Implement the extensions to the formal semantics required to support the resource analysis and to target the hardware platform specified for EmBounded (the Renesas M32C).
3. Test the resulting implementation for conformance to the language specification and the formally-proven translation scheme.

3 Meeting the objectives

The work on the Hume to HAM translator is ongoing but it has been updated continuously throughout the EmBounded project to support all the other utilities which use it.

The objectives stated above have been met:

1. Given the currently agreed (among EmBounded project partners) formal semantics for the Hume language (Deliverable D12), the Hume to HAM translator was substantially rewritten to conform to this semantics. Essentially, this semantics has been fixed for the lifetime of the EmBounded project so the core operation of the translator has not changed since the rewrite.
2. Numerous extensions to the translation, requiring additions to the HAM definition have been implemented. These have either been to support the resource analysis, particularly the inclusion of type information in the HAM, or to provide additional information for the target platform, for example interrupt vectors and hardware addresses for the M32C.
3. Testing is carried out on a small but significant set of example Hume programs but there is also, now, a large amount of Hume application code which exists as circumstantial evidence for the correctness of the translation.

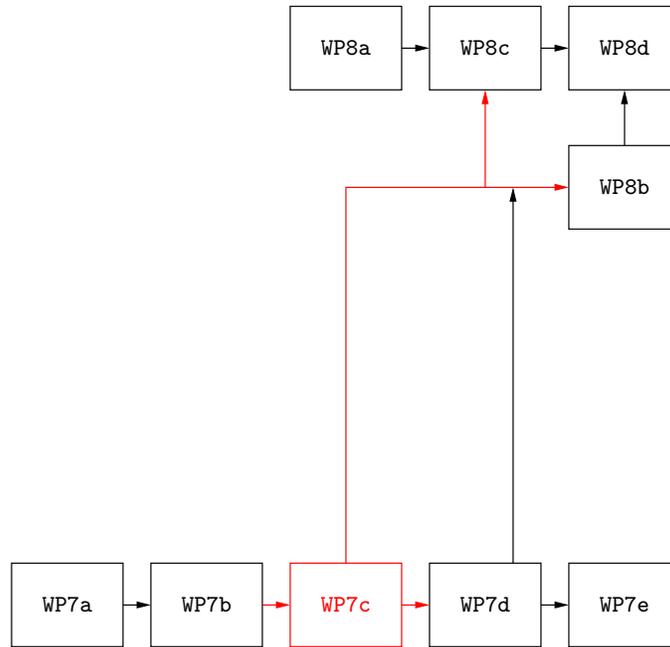


Figure 1: Inter-task dependencies for EmBounded

4 Relation with other deliverables

Figure 1 positions workpackage WP7c in the overall EmBounded task diagram. WP7c depends on WP7b (Formal translation from Hume to HAM instructions) and is required by WP7d (Hume model-checking), WP8b (Real-time computer vision algorithms) and WP8c (Evaluation of Hume). Three software deliverables have been produced in WP7c: D23 (Hume to HAM translator); D24 (Machine Code Generators); and D25 (Analyses integrated into the Compiler). A CD has been produced containing this software, and the software has also been made publicly available on the project web site (<http://www.embounded.org>).

Since work on the Hume compiler is central to the entire EmBounded project, a detailed diagram of all the interdependencies from these deliverables would be infeasibly complex. Instead, we present a table of dependencies based upon the requirements of the deliverables. Table 1 shows the deliverables which are interdependent with this deliverable.

5 Material provided

We provide the following material as evidence of our having met the objectives in this deliverable:

- CD containing the sources and installation instructions for the `humec` compiler implementation. A single CD has been produced containing all software produced by D23, D24 and D25. This software comprises a complete tool chain taking Hume source files as inputs and generating binary files for the chosen target architecture. The tools in the tool chain are:
 - Updated implementation of `phamc`, the Hume to HAM translator with support for the `humec` compiler (D23), and including integrated analyses (D25).
 - The implementation of the HAM to C compiler `ham2c` (D24).
 - Sources for the runtime system needed by the C compiler when linking the generated object code (D24).

Deliverable	Title	Notes
D03 (WP7b)	Formal translation from Hume to HAM	Compiler based upon formal translation
D05 (WP4a)	Report on stack size analysis	Compatibility with resource-analysis tools
D07 (WP8a)	Real-time testbed applications	Most of the applications work is based on the machine code compiler
D11 (WP4b)	Report on heap space analysis	Compatibility with resource-analysis tools
D12 (WP7a)	Formal semantics for Hume	Basis for the compilation scheme
D16 (WP3c)	Report on WCET analysis	Compatibility with resource-analysis tools
D24 (WP7c)	Machine code generators	Machine code compiler widely used by all other activities
D25 (WP7c)	Analyses integrated into compiler	Analyses highly interdependent with the compiler
D26 (WP7d)	Translator to support model-checking	Compatibility with verification tools
D33 (WP8c)	Report evaluating Hume	The compiler is evaluated along with the other tools in the project
D34 (WP7e)	Hume problem-solving environment	Compiler an integral part of the Hume PSE

Table 1: Deliverables dependent upon/depending on D23 (WP7c)

- An overall management program called `humec` which calls the other tools in the chain in the right order for producing binary files.
- Installation guide and User manual for the `phamc`, `ham2c` and `humec` programs.
- Copies of the academic papers produced by the EmBounded project that are related to the development and use of the Hume to HAM translator and machine code compiler [HFH⁺06b, HFH⁺06a, Poi06, MP06].

A Hume Abstract Machine Design

The design and implementation of the HAM to C translator is complex and has been discussed in other publications [Ham05]. Here we present, in two appendices, the design of the HAM abstract machine and a summary of the compilation scheme.

We will show how Hume programs may be compiled into an abstract machine representation. The goal of the prototype Hume Abstract Machine (pHAM) design is to provide a credible basis for research into bounded time and space computation, allowing formal cost models to be verified against a realistic implementation. An important part of this work is to provide a formal and precise translation scheme as given in Appendix B. We have therefore defined the pHAM at a level which is similar to, but slightly more abstract than, the JVM [LY99], giving a formalised description of the abstract machine in terms of a simple abstract register language which can be easily mapped to machine code. Absolute space- and time-performance (while an important long-term objective for Hume) is thus less important in this initial design than predictability, simplicity and ease of implementation.

name	interpretation	name	interpretation
<i>S</i>	stack	<i>pc</i>	program counter
<i>H</i>	heap	<i>pcr</i>	restart program counter
<i>sp</i>	stack pointer	<i>blocked</i>	box blocked
<i>hp</i>	heap pointer	<i>blockedon</i>	output on which blocked
<i>fp</i>	frame pointer	<i>EXNPC</i>	exception program counter
<i>mp</i>	match pointer	<i>ins</i>	input buffers
<i>inp</i>	input pointer	<i>outs</i>	output buffers
<i>rs</i>	current ruleset	<i>nIns</i>	number of inputs
<i>base</i>	base ruleset	<i>nOuts</i>	number of outputs
		<i>timeout</i>	current timeout value
		<i>thandler</i>	pc for timeout handler

Figure 2: Thread-specific registers, constants and memory areas – the *thread state record*

name	interpretation
<i>rules</i>	array of rule entry points
<i>nRules</i>	number of rules
<i>rp</i>	current rule pointer

Figure 3: Ruleset-specific registers and constants

A.1 Outline Design

The prototype Hume Abstract Machine is loosely based on the design of the classical G-Machine [Aug87] or SECD-Machine [Lan64], with extensions to manage concurrency and asynchronicity. Each Hume box is implemented as a thread with its own dynamic stack (*S*) and heap (*H*) and associated stack and heap pointers (*sp* and *hp*). These and the other items that form part of the individual *thread state record* are shown in Figure 2. Each function and box has an associated *ruleset* (Figure 3). The ruleset is used for two purposes: it gives the address of the next rule to try if matching fails; and it is used to reorder rules if fair matching is specified. The box ruleset is specified as the *base* field of the thread state record. Function rulesets are set as part of a function call.

The motivation for a separate stack for each thread is to maintain independence between thread states. Similarly, separate heaps allow a simple, real-time model of garbage collection where the *entire heap* for a box becomes garbage each time a thread completes, and all heaps can therefore be allocated from the same common dynamic memory area. The sizes of all stack and heap spaces are fixed at compile-time using the static analysis and in principle, small pointer ranges (e.g. 8 bits) can be used in either case. The main disadvantage of our present design is that we cannot use physically- or virtually-shared heap to communicate arguments and results between threads. Rather, such values are copied between heaps at the beginning and end of thread execution as explicit values within wire buffers. In effect, we have a simple copying garbage collector for which liveness can be trivially determined.

There is an analogy with the *working copies* of global variables that may be obtained by implementations of the JVM [LY99]. However, variable accesses in the JVM may occur at any point during thread execution, not only at the beginning/end as in the pHAM. Moreover, unlike the pHAM, which is stateless, the JVM maintains a virtually shared heap containing *master copies* of each variable. Our design is thus closer to that of Eden [BLOP96]: a reactive functional language based on Haskell.

MkBool b	$H[hp] := \mathbf{Bool} \ b; S[sp] := hp; ++sp; hp := hp + \mathcal{H}_{bool}; ++pc$
MkChar c	$H[hp] := \mathbf{Char} \ c; S[sp] := hp; ++sp; hp := hp + \mathcal{H}_{char}; ++pc$
MkInt32 i	$H[hp] := \mathbf{Int32} \ i; S[sp] := hp; ++sp; hp := hp + \mathcal{H}_{int32}; ++pc$
MkFloat32 f	$H[hp] := \mathbf{Float32} \ f; S[sp] := hp; ++sp;$ $hp := hp + \mathcal{H}_{float32}; ++pc$
MkString s	$H[hp] := \mathbf{String} \ s; S[sp] := hp; ++sp; hp := hp + ssize(s); ++pc$
...	
MkNone	$H[hp] := \mathbf{None}; S[sp] := hp; ++sp; hp := hp + \mathcal{H}_{none}; ++pc$
MkCon $c \ n$	$H[hp] := \mathbf{Con} \ c \ n \ (S[sp - 1]) \ \dots \ (S[sp - n - 1]); sp := sp - n;$ $S[sp - 1] := hp; hp := hp + \mathcal{H}_{con} + n; ++pc$
MkTuple n	$H[hp] := \mathbf{Tuple} \ n \ (S[sp - 1]) \ \dots \ (S[sp - n - 1]); sp := sp - n;$ $S[sp - 1] := hp; hp := hp + \mathcal{H}_{tuple} + n; ++pc$
MkVector n	$H[hp] := \mathbf{Vector} \ n \ (S[sp - 1]) \ \dots \ (S[sp - n - 1]); sp := sp - n;$ $S[sp - 1] := hp; hp := hp + \mathcal{H}_{vector} + n; ++pc$
MkFun $f \ m \ n$	$H[hp] := \mathbf{Fun} \ f \ m \ n \ (S[sp - 1]) \ \dots \ (S[sp - n - 1]); sp := sp - n;$ $S[sp - 1] := hp; hp := hp + \mathcal{H}_{fun} + n; ++pc$
Push n	$sp := sp + n; ++pc$
Pop n	$sp := sp - n; ++pc$
Slide n	$S[sp - n - 1] := S[sp - 1]; sp := sp - n; ++pc$
SlideVar n	$S[sp - n - 1] := S[sp - 1]; sp := sp - n; ++pc$
Copy n	$S[sp] := S[sp - n - 1]; ++sp; ++pc$
CopyArg n	$S[sp] := S[fp - \mathcal{S}_{frame} - n - 1]; ++sp; ++pc$
CreateFrame n	$S[sp] := fp; fp := sp + 1; sp := sp + n + 1; ++pc$
PushVar n	$S[sp] := S[fp + n]; ++sp; ++pc$
MakeVar n	$S[fp + n] := S[sp - 1]; --sp; ++pc$

Figure 4: Hume Abstract Machine Instructions (Heap and Stack Manipulation)

The pHAM design uses a pure stack calling convention. Function arguments are followed by a three-item subframe containing the return address, a pointer to the previous ruleset, and the previous frame pointer. In the rules that follow, the size of this subframe is given by the constant \mathcal{S}_{frame} . The local frame pointer fp points immediately after this subframe, to the address of the first local variable. For consistency, the same layout is used at the outer thread level. In this case, the box inputs are stored in the argument position, and the return address item is redundant. All values on the stack other than the saved return address, ruleset and frame pointer are local heap pointers (i.e. they are *boxed* [PL92]). Moreover, in the current design there is no separate *basic value stack* to handle scalar values as in some versions of the G-Machine [PL92], STG-Machine [Pey92] etc. nor are scalars and heap objects mixed on the stack as in the JVM [LY99].

A.2 The Hume Abstract Machine Instructions

The abstract machine instructions implement the abstract machine design described above. An operational description of these instructions is given in Figures 4–9. We use a number of auxiliary definitions: *copy* creates a copy of a wire value in the appropriate heap/wire; *getchar* reads a character from the

Goto l	$pc := l$
If l	if $S[sp - 1] = true$ then $pc := l$ else $++pc$ endif ; $--sp$
Call f	$S[sp] := pc + 1$; $S[sp + 1] := rs$; $++sp$; $rs := f.ruleset$; $rs.rp := 0$; $pc := rs.rules[0]$
Return l	$rs := S[fp - 2]$; $pc := S[fp - 3]$; $sp' := fp - \mathcal{S}_{frame}$; $fp := S[fp - 1]$; $S[sp'] := S[sp - 1]$; $sp := sp' + 1$
CallPrim1 p	$S[sp] := p(S[sp])$; $++pc$
CallPrim2 p	$S[sp - 1] := p(S[sp])$ ($S[sp - 1]$); $--sp$; $++pc$
CallPrim3 p	$S[sp - 1] := p(S[sp])$ ($S[sp - 1]$) ($S[sp - 2]$); $sp := sp - 2$; $++pc$
CallVar v x	let $H[v]$ be Fun f m n $a_1 \dots a_m$ in if $m + x \geq n$ then for $i = 1$ to m do $S[sp + i - 1] := a_i$; $sp := sp + m$; Call f ; else $H[hp] := \mathbf{Fun}$ f ($m + x$) n $a_1 \dots a_m$ $S[sp - 1] \dots S[sp - x - 1]$; $sp := sp - x$; $S[sp - 1] := hp$; $hp := hp + \mathcal{H}_{fun} + m + x$; $++pc$
AP n	$--sp$; CallVar($S[sp]$) n ;

Figure 5: Hume Abstract Machine Instructions (Control)

specified stream; *putvalue* writes a representation of its value argument to the specified stream; and *reschedule* terminates the current box execution, passing control to the abstract machine scheduler. \mathcal{H}_{con} , \mathcal{H}_{int32} etc. are constants defining the sizes of heap objects. A number of pseudo-instructions: **Box**, **Stream**, **Wire**, **Label**, **Function**, **Rule** and **Require** are also used to provide information about program structure that is exploited by the abstract machine implementation (Figure 7).

A.2.1 Heap Object Creation and Stack Manipulation (Figure 4).

Tagged objects are created in the heap, and pointers to the new object stored on the top of the stack. For scalar values (booleans, characters, integers, floats and strings – **MkInt32** etc.), the actual value is taken directly from the instruction stream. For strings, this value is a pointer into a global shared string table). The instruction **MkNone** creates a special value **None**, which is tested in the **CheckOutputs/Write** instructions. Finally, **MkCon** builds user-defined data structures of a given size, **MkTuple** builds tuples and **MkVector** builds vectors.

The abstract machine uses a number of simple and conventional stack manipulation operations: **Push** and **Pop** manipulate the stack pointer directly; and **Copy** and **CopyArg** copy stack locations or function arguments to the top of the stack. Two operations are used to restore stack frames following a call: **Slide** pops the stack frame, removing the function arguments after a call, but leaving the result on the top of the stack; **SlideVar** has a similar purpose, but is used where the call has been made indirectly through a closure. Three operations manipulate variables: **PushVar** copies a local variable to the stack; **PushVarF** (not shown) does the same for non-local variables; and **MakeVar** sets the value of a local variable.

MatchRule	$mp := fp - \mathcal{S}_{frame} + 1; inp := 0; pc := rs.rules[rp]; ++rp$
MatchNone	$--mp; ++inp; ++pc$
MatchAvailable	$\mathbf{if} \neg ins[inp].available \mathbf{then} pc := rules[rp] \mathbf{endif}; inp := inp + 1$
MatchBool b	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Bool} \ b \mathbf{then} pc := rules[rp] \mathbf{endif}$
MatchChar c	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Char} \ c \mathbf{then} pc := rules[rp] \mathbf{endif}$
MatchString s	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{String} \ s \mathbf{then} pc := rules[rp] \mathbf{endif}$
MatchInt32 i	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Int32} \ i \mathbf{then} pc := rules[rp] \mathbf{endif}$
MatchFloat32 f	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Float32} \ f \mathbf{then} pc := rules[rp] \mathbf{endif}$
...	
MatchCon $c \ n$	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Con} \ c \ n \mathbf{then} pc := rules[rp] \mathbf{endif}$
MatchTuple n	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Tuple} \ n \mathbf{then} pc := rules[rp] \mathbf{endif}$
MatchVector s	$--mp; \mathbf{if} H[S[mp]] \neq \mathbf{Vector} \ s \mathbf{then} pc := rules[rp] \mathbf{endif}$
Unpack	$\mathbf{let} \ \text{offset} =$ $\quad \mathbf{if} \ H[S[--sp]] = \mathbf{Tuple} \ n \mathbf{then} \ 2$ $\quad \mathbf{else} \ \mathbf{if} \ H[S[sp]] = \mathbf{Con} \ c \ n \mathbf{then} \ 3 \mathbf{in}$ $\quad \mathbf{else} \ \mathbf{if} \ H[S[sp]] = \mathbf{Vector} \ n \mathbf{then} \ 2 \mathbf{in}$ $\quad \quad \mathbf{for} \ i = 0 \mathbf{to} \ n - 1 \mathbf{do} \ S[sp++] := H[hp + \text{offset} + i]; \mathbf{endfor};$ $\quad ++pc$
StartMatches	$pc := base.rules[0]$
Reorder	$\mathbf{let} \ n = rs.nRules - 1; r = rs.rules[rp] \mathbf{in}$ $\quad \mathbf{for} \ i = rp \mathbf{to} \ n \mathbf{do}$ $\quad \quad rs.rules[i] := rs.rules[i + 1]$ $\quad \mathbf{endfor};$ $\quad rs.rules[n] := r; ++pc$

Figure 6: Hume Abstract Machine Instructions (Pattern Matching)

A.2.2 Control Operations (Figure 5).

The Hume abstract machine control instructions are shown in Figure 5. **Goto** sets the pc to the appropriate instruction. **If** does the same conditionally on the value on the top of the stack. **Call** calls the specified function, saves the current ruleset on the stack, and updates the ruleset. **CallPrim1/2/3** call primitive (built-in) functions with the corresponding numbers of arguments.

A.2.3 Pattern matching (Figure 6).

We use a set of high level pattern matching instructions rather than compiling into a series of case matches as with e.g. the STG-Machine [Pey92]. Thread matching is initiated by the **StartMatches** instruction, which sets the program counter to the first rule in the base ruleset. Identical matching operations are used both for box inputs and for function arguments. The operations are divided into three sets: the **MatchRule** operation which initialises the matching for a rule; the **MatchAvailable** and **MatchNone** operations which check box input availability (**MatchNone** for *-patterns); and the value matching operations such as **MatchInt32** or **MatchCon**, which use the current match pointer, mp . Nested matching is achieved by unpacking the arguments onto the stack using **Unpack**. Finally rules may be reordered if fair matching is required using **Reorder**.

Label l	l labels the next instruction
Function $f l_1 \dots l_n$	Function f has rules at labels $l_1 \dots l_n$
Box $b h s i o r$	Box b has heap h , stack s , i inputs, o outputs and r rules
Rule $b l_1 \dots l_n$	Box b has rules at labels $l_1 \dots l_n$
Require $b x_1 \dots x_n$	Box b requires inputs $x_1 \dots x_n$
Stream s In/Out $h s$	Stream h has heap h and stack s
Wire $wi i wo o h$	Wire connects input $wi.i$ to output $wo.o$ with heap h

Figure 7: Hume Abstract Machine Pseudo-Instructions

Raise x	$H[hp] := \mathbf{Exn} x (S[sp - 1]); S[sp - 1] := hp;$ $hp := hp + \mathcal{H}_{exn}; pc := \mathbf{EXNPC}$
Within $l t$	$S[sp] := timeout; S[sp + 1] := handler; sp := sp + 2;$ $timeout := t; handler := l;$
RaiseWithin x	$timeout := \mathbf{NEVER}; ++pc;$
DoneWithin	$handler := S[- - sp]; timeout := S[- - sp];$

Figure 8: Hume Abstract Machine Instructions (Exceptions)

A.2.4 Exceptions (Figure 8).

Exceptions are raised by the **Raise** instruction, which constructs the relevant exception value and then transfer control to the box's exception handler (EXNPC). **within**-expressions are managed by three instructions, which manipulate *timeout* and *handler*: provided the new timeout t is earlier than the current timeout, *timeout*, the **Within** instruction will stack the previous timeout value, together with the timeout handler, *handler*. If a timeout occurs, then control will be transferred to the timeout handler, whose first action will be to use a **RaiseWithin** instruction to disable the timeout, by setting the timeout so that it will never occur. Finally, if the expression doesn't trigger a timeout, then **DoneWithin** will restore the previous *timeout* and *handler* values from the stack. Similar instructions (not shown here) are used to handle space restrictions.

A.2.5 Thread input/output and rescheduling operations (Figure 9).

Thread input and output on wires is handled by two sets of operations. The **CopyInput** instruction copies the specified input from the input wire into the heap and places it on the top of the stack prior to matching. If matching is successful, input is *consumed* using the **Consume** operation, which resets the availability flag for the appropriate input wire, thereby permitting subsequent **Write** instructions to succeed for that wire.

Thread output is handled by two analogous operations. The **Write** operation writes the value on the top of the stack to the specified output wire. Before this can be done, the **CheckOutputs** operation is used to ensure that all required **Write** instructions will succeed. This is achieved by checking that all output wire buffers are empty, as indicated by the wire's *available* flag. If not, then the thread blocks until the value on the wire has been consumed, and the *available* flag has been cleared. If the heap value is **None** (corresponding to $*$ on the output), then the **Write** will not actually write anything to the output wire, and the *available* flag is therefore ignored by **CheckOutputs**.

Control is returned to the scheduler either when a thread blocks, either as a consequence of being unable to write some output during the **CheckOutputs** operation, or explicitly when a thread termi-

CopyInput n	$S[sp] := copy(ins[n].value); ++sp; ++pc$
Consume n	$ins[n].available := false; ++pc$
CheckOutputs	<pre> for $i = 0$ to $nOuts$ do if $H[S[sp - i - 1]] \neq \mathbf{None}$ and $outs[i].available$ then $blocked := true; blockedon := i; pcr := pc; reschedule;$ endif; endfor; $++pc$ </pre>
Write n	<pre> if $H[S[sp]] \neq \mathbf{None}$ then $outs[n].value := copy(H[S[sp]]); outs[n].available := true;$ endif; $++pc$ </pre>
Input s	<pre> let $c = getchar\ s$ in $H[hp] := Char\ c; S[sp] := hp; ++sp;$ $hp := hp + \mathcal{H}_{char}; ++pc$ </pre>
Output s	$putvalue(s, S[sp - 1]); --sp; ++pc$
Schedule	$reschedule$

Figure 9: Hume Abstract Machine Instructions (Threads)

nates as a consequence of executing the **Schedule** operation. In either case, the scheduler will select a new runnable thread to execute. If there is no runnable thread, then in the current implementation the system will terminate. In a distributed system, it would be necessary to check for global termination, including outstanding communications that could awaken some thread.

Finally, two operations are provided to manage stream and device input/output. A special I/O thread is attached to each stream/device by the **Stream** pseudo-instruction (Figure 7). Executing the **Input** operation blocks this thread if no input is available, but otherwise reads input into the thread's heap. The **Output** operation simply writes the top stack value to the appropriate device. Normal wire operations are used to interface other threads to these special I/O threads. For simplicity, we only show character-level I/O, but more complex I/O can also be managed in a similar way.

B Compilation Scheme

This section outlines a formal compilation scheme for translating Hume programs into pHAM instructions. Our intention is to demonstrate that a formal (and ultimately provable) model of compilation can be constructed for Hume. By constructing a formal translation to real machine code from pHAM code, it is then possible to verify both correctness of the compiler output and time/space cost models.

Figures 10–14 outline rules for compiling Hume abstract syntax forms into the abstract machine instructions described in Appendix A, as a formal compilation scheme similar to that for the G-machine [Aug87]. These rules have been used to construct a compiler from Hume source code to the pHAM, whose main component is a 500-line Haskell module translating abstract syntax to pHAM instructions.

The compilation scheme makes extensive use of a simple sequence notation: $\langle i_1, \dots, i_n \rangle$ denotes a sequence of n items. The $++$ operation concatenates two such sequences. Many rules also use an environment ρ which maps identifiers to $\langle depth, offset \rangle$ pairs.

Four auxiliary functions are used, but not defined here: $maxVars$ calculates the maximum number of variables in a list of patterns; $bindDefs$ augments the environment with bindings for the variable

$\mathcal{C}_E \rho (c e_1 \dots e_n)$	=	$\mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++ \langle \text{MkCon } c \ n \rangle$
$\mathcal{C}_E \rho (p e_1 \dots e_n)$	=	$\mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++ \langle \text{CallPrimn } p \rangle$
$\mathcal{C}_E \rho (f e_1 \dots e_n)$	=	let $a = \text{arity } f$ in $\mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++$ if $n = a$ then $\langle \text{Call } f, \text{Slide } n \rangle$ else if $n < a$ then $\langle \text{MkFun} \rangle$ else $\langle \text{Call } f, \text{Slide } n, \text{AP } (n - a) \rangle$
$\mathcal{C}_E \rho (v e_1 \dots e_n)$	=	$\mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++$ $\langle \text{CallVar } v \ n, \text{SlideVar } v \rangle$
$\mathcal{C}_E \rho (i)$	=	$\langle \text{MkInt32 } i \rangle$
\dots		
$\mathcal{C}_E \rho (*)$	=	$\langle \text{MkNone} \rangle$
$\mathcal{C}_E \rho (e_1, \dots, e_n)$	=	$\mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++ \langle \text{MkTuple } n \rangle$
$\mathcal{C}_E \rho \langle \langle e_1, \dots, e_n \rangle \rangle$	=	$\mathcal{C}_E \rho e_n ++ \dots ++ \mathcal{C}_E \rho e_1 ++ \langle \text{MkVector } n \rangle$
$\mathcal{C}_E \rho (\text{var})$	=	$\langle \text{PushVar } n \rangle$

Figure 10: Compilation Rules for Expressions (1)

definitions taken from a declaration sequence – the *depth* of these new bindings is 0, whilst the depth of existing variable bindings in the environment is incremented by 1; *bindVars* does the same for a sequence of patterns; and *labels* generates new labels for a set of function/box rules. Note that where labels *lt*, *ln*, *lx* etc. are used, these are assumed to be unique in the obvious way: there is at most one **Label** pseudo-instruction for each label in the translated program. Labels for boxes and function blocks are derived in a standard way from the (unique) name of the box or function. Finally, priming (e.g. ρ') has no semantic significance as in mathematics: it is used here for naming purposes only.

The rules are structured by abstract syntax class. The rules for translating expressions (\mathcal{C}_E etc. – Figures 10–11) are generally straightforward, but note that function frames are created to deal with *let*-expressions and other similar structures, which then exploit the function calling mechanism. This allows the creation of local stack frames. It would obviously be possible to eliminate the function call for *let*-expressions provided the stack frame was properly set up in order to allow access to non-local definitions. Note also the three cases for function application: respectively corresponding to the usual first-order case, to under-application of a function (where a closure will be created using **MkFun**), and to over-application of a function (where the closure will be applied to the additional arguments using **Ap**).

In order to avoid the increase in closure sizes that would result from lambda-lifting, we instead use a *static link pointer* approach, where each new stack frame is linked at runtime to the frame corresponding to the function that statically encloses the current function body. This gives a space and time cost model that can be more easily related to the source, since we will not transform value definitions into functions as a result of lifting free variables. It also reduces the number and size of functional closures that must be created. However, it does increase the (fixed) size for each stack frame, since a static link pointer to the enclosing scope must be stored in each frame. In order to exploit this approach, we introduce variants of the **PushVar**, **SlideVar**, **CallVar** instructions that use the static link pointer to locate the variable definition from the correct statically linked frame. For simplicity, we have omitted these instructions here.

$\mathcal{C}_E \rho (\mathbf{if} \ c \ \mathbf{then} \ t \ \mathbf{else} \ f)$	$=$	$\mathcal{C}_E \rho \ c \ ++ \ \langle \mathbf{If} \ lt \ \rangle \ ++ \ \mathcal{C}_E \rho \ f \ ++$ $\langle \mathbf{Goto} \ ln, \ \mathbf{Label} \ lt \ \rangle \ ++ \ \mathcal{C}_E \rho \ t \ ++$ $\langle \mathbf{Label} \ ln \ \rangle$
$\mathcal{C}_E \rho (\mathbf{case} \ e \ \mathbf{of} \ ms)$	$=$	$\mathcal{C}_E \rho \ e \ ++ \ \langle \mathbf{Call} \ lc, \ \mathbf{Slide} \ 1, \ \mathbf{Goto} \ ln, \ \mathbf{Label} \ lc \ \rangle \ ++$ $\mathcal{C}_{Case} \rho \ ms \ ++$ $\langle \mathbf{Label} \ ln, \ \mathbf{Function} \ lc \ (\mathbf{labels} \ lc) \ \rangle$
$\mathcal{C}_E \rho (\mathbf{let} \ d_1 \ \dots \ d_n \ \mathbf{in} \ e)$	$=$	$\mathbf{let} \ \rho' = \mathbf{bindDefs} \ \langle d_1 ; \dots ; d_n \ \rangle \ \rho \ \mathbf{in}$ $\langle \mathbf{Call} \ ll, \ \mathbf{Goto} \ ln, \ \mathbf{Label} \ ll, \ \mathbf{CreateFrame} \ n \ \rangle \ ++$ $\mathcal{C}_{Let} \rho \ 0 \ d_1 \ ++ \ \dots \ ++ \ \mathcal{C}_{Let} \rho \ (n-1) \ d_n \ ++$ $\mathcal{C}_E \rho' \ e \ ++ \ \langle \mathbf{Return}, \ \mathbf{Label} \ ln \ \rangle \ ++$ $\langle \mathbf{Function} \ ll \ \rangle$
$\mathcal{C}_E \rho (e \ \mathbf{as} \ \tau)$	$=$	$\mathcal{C}_E \rho \ e \ ++ \ \langle \mathbf{CallPrim1} \ \mathbf{Coerce} \ - \ \tau \ \rangle$
$\mathcal{C}_E \rho (e \ \mathbf{within} \ c)$	$=$	$\langle \mathbf{Within} \ lw \ c \ \rangle \ ++ \ \langle \mathcal{C}_E \rho \ e \ ++ \ \langle \mathbf{Goto} \ ln \ \rangle \ ++$ $\langle \mathbf{Label} \ lw, \ \mathbf{RaiseWithin} \ \rangle \ ++$ $\langle \mathbf{MkTuple} \ 0, \ \mathbf{Raise} \ \mathit{Timeout/HeapOverflow} \ \rangle \ ++$ $\langle \mathbf{Label} \ ln, \ \mathbf{DoneWithin} \ \rangle$
$\mathcal{C}_E \rho (\mathbf{raise} \ \text{exnid} \ e)$	$=$	$\mathcal{C}_E \rho \ e \ ++ \ \langle \mathbf{Raise} \ \text{exnid} \ \rangle$
$\mathcal{C}_E \rho (e \ \mathbf{::} \ \tau)$	$=$	$\mathcal{C}_E \rho \ e$
$\mathcal{C}_{Case} \rho \ \langle r_1, \dots, r_m \ \rangle$	$=$	$\mathbf{let} \ n = \mathbf{maxVars} \ \langle r_1, \dots, r_m \ \rangle \ \mathbf{in}$ $\langle \mathbf{CreateFrame} \ n \ \rangle \ ++$ $\mathcal{C}_F \rho \ \langle r_1, \dots, r_m \ \rangle$
$\mathcal{C}_{Let} \rho \ n \ (v = e)$	$=$	$\mathcal{C}_E \rho \ e \ ++ \ \langle \mathbf{MakeVar} \ n \ \rangle$

Figure 11: Compilation Rules for Expressions (2)

The rules for translating box and function declarations are shown in Figure 12. These rules create new stack frames for the evaluation of the box or function, label the entry points and introduce appropriate pseudo-instructions. In the case of box declarations, it is also necessary to copy inputs to the stack using **CopyInput** instructions and to deal with fair matching.

Box bodies are compiled using $\mathcal{C}_R/\mathcal{C}_{R'}$ (Figure 13). These rules compile matches for the outer level patterns using \mathcal{C}_P , then compile inner pattern matches using \mathcal{C}_A , before introducing **Consume** instructions for non-* input positions. The RHS can now be compiled. If more than one result is to be produced, the tuple of outputs is unpacked onto the stack. A **CheckOutputs** is inserted to verify that the outputs can be written using appropriate **Write** instructions. Finally, a **Reorder** is inserted if needed to deal with fair matching, and a **Schedule** returns control to the scheduler. The compilation of function/handler bodies using $\mathcal{C}_F/\mathcal{C}_{F'}$ is similar, except that $\mathcal{C}_{P'}$ is used rather than \mathcal{C}_P , there is no need to deal with box inputs/outputs or fair matching, and a **Return** rather than **Schedule** is inserted at the end of each compiled rule.

Finally patterns are compiled using $\mathcal{C}_P/\mathcal{C}_{P'}$ (Figure 14), where \mathcal{C}_P inserts the **MatchNone/MatchAvailable** instructions that are needed at the box level, and $\mathcal{C}_{P'}$ compiles simple patterns. Constructed values

$\mathcal{C}_D \rho (\mathbf{box} \ b \ ins \ outs \ \mathbf{fair} \ rs \ \mathbf{handle} \ xs)$	$= \mathcal{C}_B \rho \ true \ b \ ins \ outs \ rs$
$\mathcal{C}_D \rho (\mathbf{box} \ b \ ins \ outs \ \mathbf{unfair} \ rs \ \mathbf{handle} \ xs)$	$= \mathcal{C}_B \rho \ false \ b \ ins \ outs \ rs$
$\mathcal{C}_D \rho (v = e)$	$= \langle \text{Label } v, \text{CreateFrame } 0 \rangle ++ \mathcal{C}_E \rho \ e ++ \langle \text{Return} \rangle$
$\mathcal{C}_D \rho (f = p_1 \rightarrow e_1 ; \dots p_n ; \rightarrow e_n) =$	$\mathbf{let} \ n = \maxVars \langle p_1, \dots, p_n \rangle \ \mathbf{in}$
	$\langle \text{Label } f, \text{CreateFrame } n \rangle ++$
	$\mathcal{C}_F \rho \langle p_1 \rightarrow e_1 \dots p_n \rightarrow e_n \rangle ++$
	$\langle \text{Function } f \ (\text{labels } f) \rangle$
$\mathcal{C}_D \rho (\mathbf{wire} \ p_1 \ \mathbf{to} \ p_2) =$	$\langle \text{Wire } \dots \rangle$
$\mathcal{C}_D \rho (\mathbf{data} \ t \ v_1 \ \dots \ v_n) =$	$\langle \rangle$
$\mathcal{C}_B \rho \ f \ b \ (in_1, \dots, in_i) \ (out_1, \dots, out_m) \ rs =$	$\mathbf{let} \ n = \maxVars \ rs \ \mathbf{in}$
	$\langle \text{Label } b \rangle ++$
	$\langle \text{CopyInput } (i - 1), \dots, \text{CopyInput } 0 \rangle ++$
	$\langle \text{Push } 2, \text{CreateFrame } n \rangle ++$
	$(\mathbf{if} \ f \ \mathbf{then} \ \langle \text{StartMatches} \rangle \ \mathbf{else} \ \langle \rangle) ++ \mathcal{C}_R \rho \ f \ m \ rs ++$
	$\langle \text{Box } b \ \dots \rangle$

Figure 12: Compilation Rules for Declarations and Box Bodies

$\mathcal{C}_R \rho \ f \ m \ \langle r_1 ; \dots ; r_n \rangle$	$= \mathcal{C}_{R'} \rho \ f \ m \ r_1 ++ \dots ++ \mathcal{C}_{R'} \rho \ f \ m \ r_n$
$\mathcal{C}_{R'} \rho \ f \ m \ (p_1 \ \dots \ p_n \rightarrow e)$	$= \langle \text{Label } lr, \text{MatchRule} \rangle ++$
	$\mathcal{C}_P \ p_1 ++ \dots ++ \mathcal{C}_P \ p_n ++$
	$\mathcal{C}_A \ p_1 ++ \dots ++ \mathcal{C}_A \ p_n ++$
	$\mathcal{C}_C \ 0 \ p_1 ++ \dots ++ \mathcal{C}_C \ (n - 1) \ p_n ++$
	$\mathcal{C}_E \rho \ e ++$
	$(\mathbf{if} \ m > 1 \ \mathbf{then} \ \langle \text{Unpack} \rangle \ \mathbf{else} \ \langle \rangle) ++$
	$\langle \text{CheckOutputs} \rangle ++$
	$\langle \text{Write } (n - 1), \dots, \text{Write } 0 \rangle ++$
	$(\mathbf{if} \ f \ \mathbf{then} \ \langle \text{Reorder} \rangle \ \mathbf{else} \ \langle \rangle) ++$
	$\langle \text{Schedule} \rangle$
$\mathcal{C}_C \ n \ (*)$	$= \langle \rangle$
$\mathcal{C}_C \ n \ (p)$	$= \langle \text{Consume } n \rangle$
$\mathcal{C}_F \rho \ (r_1 ; \dots ; r_n)$	$= \mathcal{C}_{F'} \rho \ r_1 ++ \dots ++ \mathcal{C}_{F'} \rho \ r_n$
$\mathcal{C}_{F'} \rho \ (p_1 \ \dots \ p_n \rightarrow e)$	$= \mathbf{let} \ \rho' = \text{bindVars} \langle p_1, \dots, p_n \rangle \rho \ \mathbf{in}$
	$\langle \text{Label } lf, \text{MatchRule} \rangle ++$
	$\mathcal{C}_{P'} \ p_1 ++ \dots ++ \mathcal{C}_{P'} \ p_n ++$
	$\mathcal{C}_A \ p_1 ++ \dots ++ \mathcal{C}_A \ p_n ++$
	$\mathcal{C}_E \ \rho' \ e ++$
	$\langle \text{Return} \rangle$

Figure 13: Compilation Rules for Rule Matches and Functions

are matched in two stages: firstly the constructor is matched, and then if the match is successful, the matched object is deconstructed on the stack to allow its inner components to be matched against the

$\mathcal{C}_P (*)$	=	$\langle \text{MatchNone} \rangle$
$\mathcal{C}_P (p)$	=	$\langle \text{MatchAvailable} \rangle \text{++ } \mathcal{C}_{P'} p$
$\mathcal{C}_{P'} (i)$	=	$\langle \text{MatchInt32 } i \rangle$
...		
$\mathcal{C}_{P'} (c p_1 \dots p_n)$	=	$\langle \text{MatchCon } c n \rangle$
$\mathcal{C}_{P'} (p_1 \dots p_n)$	=	$\langle \text{MatchTuple } n \rangle$
$\mathcal{C}_{P'} \langle \langle p_1 \dots p_n \rangle \rangle$	=	$\langle \text{MatchVector } n \rangle$
$\mathcal{C}_{P'} (var)$	=	$\langle \text{MatchVar } var \rangle$
$\mathcal{C}_{P'} -$	=	$\langle \text{MatchAny} \rangle$
$\mathcal{C}_A (c p_1 \dots p_n)$	=	$\mathcal{C}_{A'} \text{CopyArg} \langle p_1, \dots, p_n \rangle$
$\mathcal{C}_A (p_1, \dots, p_n)$	=	$\mathcal{C}_{A'} \text{CopyArg} \langle p_1, \dots, p_n \rangle$
$\mathcal{C}_A (x p)$	=	$\mathcal{C}_{A'} \text{CopyArg} \langle p \rangle$
$\mathcal{C}_A p$	=	$\langle \rangle$
$\mathcal{C}_{A'} i \langle p_1, \dots, p_n \rangle$	=	$\langle i, \text{Unpack} \rangle \text{++}$ $\mathcal{C}_{A'} \text{Copy } p_1 \text{++} \dots \text{++ } \mathcal{C}_{A'} \text{Copy } p_n \text{++}$ $\mathcal{C}_{P'} p_1 \text{++} \dots \text{++ } \mathcal{C}_{P'} p_n$

Figure 14: Compilation Rules for Patterns

inner patterns. These nested patterns are compiled using \mathcal{C}_A and $\mathcal{C}_{A'}$. $\mathcal{C}_{A'}$ inserts either **CopyArg** and **Unpack** instructions to decompose function/box arguments, or **Copy** and **Unpack** instructions to deal with nested pattern matches, where it is only necessary to replicate arguments that are already in the local stack frame.

References

- [Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, 1987. A.1, B
- [BLOP96] S. Breitinger, R. Loogen, Y. Ortéga Mallén, and Ricardo Peña Marí. Eden — the Paradise of Concurrent Functional Programming. In *Europar'96*, volume 1123 of *LNCS*, 1996. A.1
- [Ham05] Kevin Hammond. Exploiting purely functional programming to obtain bounded resource behaviour: The hume approach. In *CEFP*, pages 100–134, 2005. A
- [HFH⁺06a] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hoffmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable resource bounds for real-time embedded systems. In *Proc. Workshop on Innovative Techniques for Certification of Embedded Systems*, 2006. 5
- [HFH⁺06b] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hoffman, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable WCET analysis for a functional programming language. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, number 06902 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. 5

- [Jos06] S. Jost. Formal Hume Semantics. EmBounded Project Deliverable, March 2006. Deliverable D12. (document), 1
- [Lan64] P. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964. A.1
- [LJH06] H-W. Loidl, S. Jost, and K. Hammond. Hume-HAM Translation. EmBounded Project Deliverable, March 2006. Deliverable D3. (document), 1
- [LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition edition, April 1999. A, A.1
- [MP06] Greg Michaelson and Robert Pointon. Recursion, iteration and hume scheduling. In *Trends in Functional Programming (TFP'06)*, Nottingham UK, April 2006. 5
- [Pey92] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine. *J. of Functional Programming*, 2(2):127–202, 1992. A.1, A.2.3
- [PL92] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall, 1992. A.1
- [Poi06] Robert F. Pointon. Compiling vs costing - experiences with hume. In *Implementation of Functional Languages (IFL'06)*, Budapest Hungary, September 2006. 5