IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D04 (WP2a): Cost Model (Revised)

Due date of deliverable: 1st January 2006
Actual submission date: 3rd March 2006

Start date of project: 1st March 2005 Duration: 36 months

Lead contractor: Ludwig-Maximilians-Universität Revision: 1.10

Purpose: Build a low-level *formal cost model*, reflecting the costs of executing Hume Abstract Machine (HAM) expressions on one particular architecture.

Results: We present cost models for heap space, stack space and time consumption of HAM code.

Conclusion: This document presents both a formal specification of the HAM, as well as cost models for the aforementioned resources. We give a small-step, operational semantics for HAM instructions, and we define resource algebras for heap, stack and time that can be added to this operational semantics in a modular way and describe both expression level and system level resource consumption.

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)	

Specification and Cost Model for the Hume Abstract Machine (Version 1.1)

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>,
Institut für Informatik, Theoretische Informatik, Ludwig-Maximilians Universität, München
Kevin Hammond <kh@dcs.st-and.ac.uk>
School of Computing Science, Univ of St Andrews, St Andrews, Scotland

Abstract

This document is both a formal specification of the Hume Abstract Machine (HAM) semantics as well as a cost model for its time, stack and heap space consumption. The specification reflects changes made to the HAM since project start. It contains instructions to support higher-order functions and exceptions, but no explicit support for timeouts (these are still in the design phase and subject to changes). The cost models are described as resource algebras that can be freely added to the HAM specification. The latter is encoded as a 2-level, small step operational semantics.

Version 1.1: This update reflects and accounts for the Renesas M32C being used as target processor (mainly affecting the time cost model).

1 Introduction

For the execution of Hume [HM] programs we define an abstract machine, the Hume Abstract Machine (HAM). This definition is an extension of the initial design, described in [Ham03], by constructs for higher-order functions and exceptions. Additionally, we formally specify the components of the machine and its behaviour in the form of a 2-level, small-step, operational semantics and give a reference implementation of the instructions of the HAM. The operational semantics uses the approach of resource algebras, which we have developed in a previous project [ABM06], to collect information on the resource consumption during execution. The resource algebras are designed in a modular way and can be instantiated without modifying the rules of the operational semantics. We define the cost model for the HAM by giving resource algebras for stack space, heap space and time consumption. The values for stack and heap space are independent of the underlying processor. For the time information we have performed measurements of the HAM interpreter running on a PowerPC architecture.

As the specification of the abstract machine used to execute Hume programs, this document serves as interface between groups working on the compilation infrastructure for Hume. Together with the document on the Hume formal semantics (D12), and the formal description of the compilation of Hume to HAM (D3), it gives a description about the costs for executing Hume and HAM programs. This is in turn a prerequisite for developing static analyses of resource consumption. Finally, this formal specification is the basis for the work in WP7 on the certification of the resource consumption for Hume code.

The structure of this document is as follows. Section 2 describes the HAM design in general, the data structures used and the behaviour of the machine by presenting a reference implementation of the HAM instructions in pseudo-C. Section 3 describes the concept of a resource algebra and instantiates it for stack space, heap space and time consumption. Section 4 provides a formal specification of the HAM as a 2-level, small-step operational semantics. Finally, Section 5 summarises.

Comments on Version 1.1 of the cost model: This version of the HAM-level cost model updates the original version in the following aspects. Firstly, as target platform the Renesas M32C/85U embedded processor is used, and the time cost model as been replaced to reflect costs for HAM operations on this processor. Secondly, the space cost models reflect changes in the Hume compiler, in particular the costs as incurred by the new native code generator. The cost model in Version 1.1 has been validated against measured costs on the target platform in Deliverable D28.

2 Hume Abstract Machine Design and Reference Implementation

name	interpretation
S	stack
H	heap
sp	stack pointer
hp	heap pointer
fp	frame pointer
slp	function frame pointer
mp	match pointer
inp	input pointer
rs	current ruleset
$base$	base ruleset

name	interpretation
pc	program counter
pcr	restart program counter
$blocked$	box blocked
$blockedon$	output on which blocked
$INITPC$	initial program counter
ins	input buffers
$outs$	output buffers
$nIns$	number of inputs
$nOuts$	number of outputs

Figure 1: Box-specific registers, constants and memory areas — the *box state record*

name	interpretation
$rules$	array of rule entry points
$nRules$	number of rules
rp	current rule pointer

Figure 2: Ruleset-specific registers and constants

The goal of the Hume Abstract Machine (HAM) 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. 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.

The 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 *state record* are shown in Figure 1. Each function and box has an associated *ruleset* (Figure 2). 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 state record. Function rulesets are set as part of a function call.

Separate stacks are needed to maintain independent state records. Separate heaps allow a simple model of garbage collection where the entire heap becomes garbage each time a box completes. Small

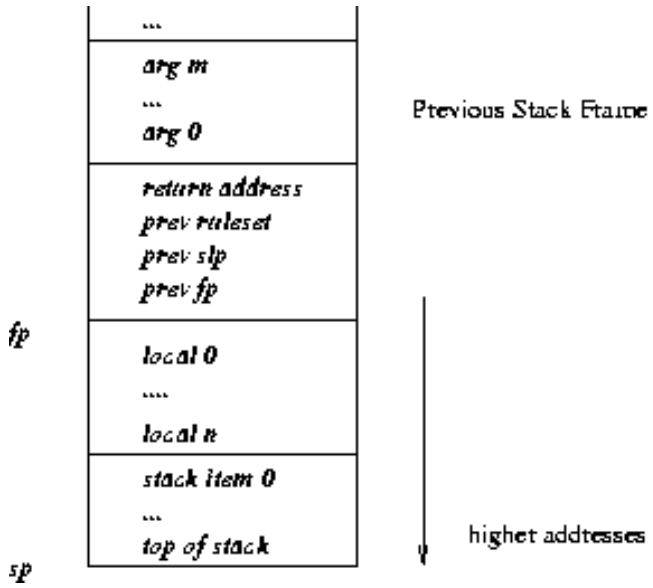

Figure 3: Stack frame layout in the Hume Abstract Machine

pointer ranges can be used in both cases (8-bit stack and heap pointers are possible for a number of applications). This helps conserve space. The corresponding disadvantage of this design is the need to communicate arguments and results between boxes rather than using a physically or virtually shared heap. This is achieved in the HAM reference implementation by copying such values between heaps at the beginning and end of the box execution. 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 HAM. Moreover, unlike the HAM, 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.

The layout of a typical stack frame is shown in Figure 3. The abstract machine design uses a pure stack calling convention. Function arguments are followed by a four-item frame-header containing the return address, a pointer to the previous ruleset, the static link pointer and the previous frame pointer. In this description, the size of this subframe is given by the constant $\mathcal{S}_{frame}$. The local frame pointer fp points immediately after the frame-header, to the address of the first local variable. The function frame pointer *slp* points to the beginning of the nearest frame representing a function (note that let etc instructions in Hume also allocate frames). For consistency, the same layout is used at the outer box level. In this case, the box inputs are stored in the argument position, and the return address item is redundant. Note that all values on the stack other than the saved return address, ruleset and frame pointer are pointers to the local heap (i.e. they are *boxed* [PL92]): 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] or recent versions of the STG-Machine.

2.1 Box Scheduling

The implementation maintains a vector of boxes, each with its own state record. Boxes are connected through *wires*, which are shared communication buffers each connecting an *in* of one box to an *out* of another (or the same) box. Each wire comprises a pair of a value (*value*) and a flag indicating that the value is valid (*available*), used to ensure correct locking.

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

Figure 4: Scheduling Algorithm

Boxes are scheduled under the control of a built-in scheduler. The exact scheduling order is not fixed by the HAM semantics. Implementations are free to realise any order that satisfies the conditions stated in Section 4.4. A box is deemed to be *runnable* if all the required inputs are available for any of its rules to be executed (Figure 4). A compiler-specified matrix is used to determine whether an input is needed: for some box t, $box[t].required[r, i]$ is true if input i is required to run rule r of that box. A single execution cycle comprises the following (all realised via explicit HAM instructions):

1. initialise stack- and heap-pointers and the base ruleset;
2. check input availability against possible matches;
3. copy data from input wires into the local heap for matching;
4. match available inputs against rules;
5. consume those inputs that have been matched and which are not ignored in the selected rule;
6. create a stack frame to hold local variables;

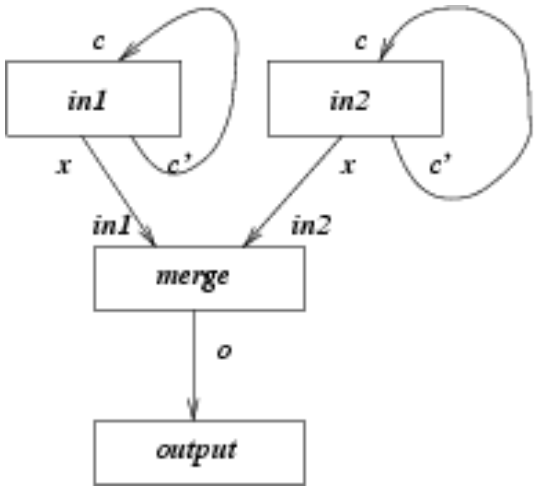

Figure 5: Hume example program as a network of boxes

constant	value (words)
$\mathcal{H}_{\mathsf{Constr}}$	2
$\mathcal{H}_{\mathsf{Tuple}}$	2
$\mathcal{H}_{float}$	2
$\mathcal{H}_{int}$	2
$\mathcal{H}_{\mathsf{Bool}}$	2
$\mathcal{H}_{\mathsf{Char}}$	2
$\mathcal{H}_{\mathsf{Str}}$	1

constant	value (words)
$\mathcal{H}_{\mathsf{Fun}}$	4
$\mathcal{H}_{\mathsf{Exn}}$	2
$\mathcal{H}_{\mathsf{Int}}$	2
$\mathcal{H}_{\mathsf{Float}}$	2
$\mathcal{S}_{saved}$	4

Table 1: Sizes of headers for heap and stack objects in the prototype Hume Abstract Machine

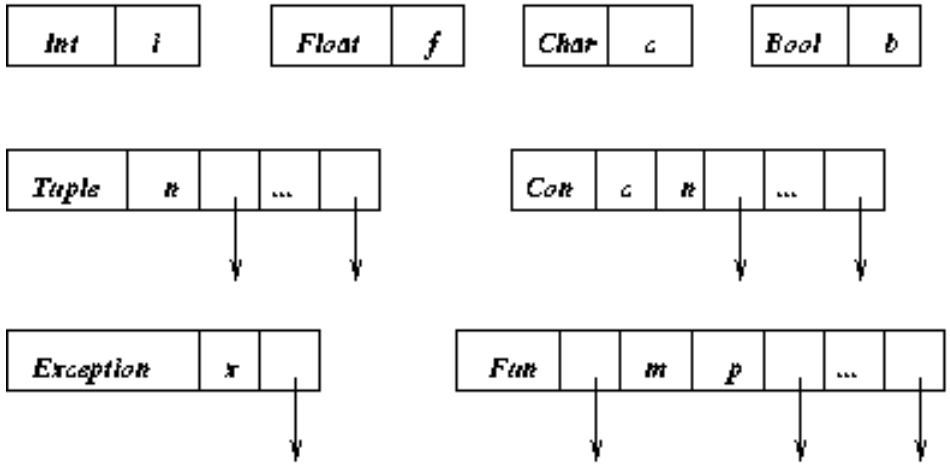

Figure 6: Heap representations in the Hume Abstract Machine

7. bind variables to input values;
8. evaluate the RHS of the selected rule;
9. check that outputs can be written to all output wires;
10. write non-ignored outputs to the corresponding wires;
11. reorder match rules according to the fairness criteria.

Note that all wires are single-buffered. A box will therefore block when writing to a wire which contains an output that has not yet been consumed. In order to ensure a consistent semantics, a single check is performed on all output wires just before any output is written. The check ignores * output positions. The box suspends if any of the needed output wires is occupied.

2.2 Heap Representations

In the prototype design, all heap cells are *boxed* with tags distinguishing different kinds of objects. Furthermore, tuple (Tuple), constructor (Constr) and exception structures (Exn) require *size* fields, and constructors also require a *constructor tag* field. Function closures (Fun) contain a pointer to an instruction sequence, the number of needed (m) and of already provided (p) arguments, and a list of pointers to those provided arguments. All data objects in a structure are referenced by pointer. There is one special representation: strings are represented as a tagged sequence of bytes. Clearly,

heap usage could be reduced using a more compact representation such as that used by modern high-performance implementations such as SML-NJ [Mac] or the STG-Machine [Pey92]. For now, we are, however, primarily concerned with bounding and predicting memory usage. Small changes to data representations can be easily incorporated into both models and implementations at a future date without affecting the fundamental results of cost modelling, except by reducing absolute costs in both cases.

2.2.1 Heap and Stack Sizes.

The sizes of the stack and heap spaces associated with each box are fixed at compile-time using a static analysis to compute the upper bound of stack and heap usage [HM02]. This analysis is already defined for flat Hume programs containing no recursion (FSM-Hume), and is currently being extended to more general cases. For the examples that have been tested to date, the analysis calculates space usage to within 15% of the actual dynamic usage. Since this will include situations where, for example, only one input of a 2-input box is ever available at run-time, or where a conditional branch is never chosen dynamically, this represents a good upper-bound estimate of the space usage. The simple analysis has been integrated into our prototype compiler (`phamc`), and results from the analysis are used in this paper where appropriate.

2.2.2 Exceptions.

Two forms of exceptions can occur during the execution of HAM code: synchronous and asynchronous exception. While synchronous exceptions are related to the execution of a particular piece of HAM code (e.g. division-by-zero), asynchronous exceptions can occur at any point and need to be checked by an external system.

The most important synchronous exceptions are stack- and heap-overflow. Before each (block of) instruction, that involves an increase of the stack- or heap-pointer, a stack- or heap-check is necessary, which makes sure that enough stack- or heap space is available, and if not raises an exception. For the pseudo-code in the reference implementation we assume that such stack- and heap-checks are added for any HAM instruction.

The most important asynchronous exception is a timeout. We assume that the HAM is embedded into a system that provides facilities for setting a timer and for checking when such a timer expires. In this case, the handler code for timeouts in the current box must be executed.

2.3 The Abstract Machine Instructions

The abstract machine instructions implement the abstract machine design described above. These instructions are shown in Figures 7–11. They are classified into stack, heap and control-flow operations, which are fairly standard, and matching and scheduling operations, which reflect the specific nature of the Hume design. The description of the instructions uses two auxiliary functions: *maxVars* calculates the maximum number of variables in a list of patterns; and *labels* generates new labels for a set of function/box rules. Where labels *lt*, *ln* etc. are used, these are assumed to be unique.

2.3.1 Heap Object Creation (Fig 7).

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) the actual value is taken directly from the instruction stream (in the case of a string, this is a pointer into the global string table). The corresponding constructor instructions are **MkBool**, **MkChar**, **MkInt***, **MkFloat*** and **MkString**. The instruction **MkNone** creates a special **None** value in the heap. This is used to prevent the writing of a particular output under dynamic programmer control.

MkBool b	$H[hp] = \mathsf{Bool}\ b;\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Bool}};\ ++pc$
MkChar c	$H[hp] = \mathsf{Char}\ c;\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Char}};\ ++pc$
MkInt i	$H[hp] = \mathsf{Int}\ i;\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Int}};\ ++pc$
MkFloat f	$H[hp] = \mathsf{Float}\ f;\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Float}};\ ++pc$
MkString s	$H[hp] = \mathsf{Str}\ s;\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Str}} + ssize(s);\ ++pc$
MkNone	$H[hp] = \mathsf{None};\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{None}};\ ++pc$
MkCon $c\ n$	$H[hp] = \mathsf{Constr}\ c\ n\ (S[sp-1])\ \ldots\ (S[sp-n-1]);\ sp := sp - n;$ $S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Constr}} + n;\ ++pc$
MkTuple n	$H[hp] = \mathsf{Tuple}\ n\ (S[sp-1])\ \ldots\ (S[sp-n-1]);\ sp := sp - n;$ $S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Tuple}} + n;\ ++pc$
MkFun $l\ m\ p$	$H[hp] = \mathsf{Fun}\ l\ m\ p\ (S[sp-1])\ \ldots\ (S[sp-p-1]);$ $sp := sp - p;\ S[sp] := hp;\ ++sp;\ hp := hp + \mathcal{H}_{\mathsf{Fun}} + p;\ ++pc$

Figure 7: Abstract Machine Instructions: Heap operations

Push n	$sp := sp + n;\ ++pc$
Pop n	$sp := sp - n;\ ++pc$
Slide n SlideVar v SlideVarF $d\ v$	$\begin{cases} m := n; \\ m := H[S[fp+v]+1]; \\ fp' := fp; \textbf{while}\ d \neq 0\ \textbf{do}\ fp' := S[fp'-2]; --d\ \textbf{endwhile};\ m := H[S[fp'+v]+1] \end{cases}$ $S[sp-m-1] := S[sp-1];\ sp := sp - m;\ ++pc$
Copy n	$S[sp] := S[sp-n-1];\ ++sp;\ ++pc$
CopyArg n	$S[sp] := S[fp - \mathcal{S}_{saved} - n - 1];\ ++sp;\ ++pc$
CreateFrame n	$S[sp] := fp;\ fp := sp + \mathcal{S}_{saved};\ sp := sp + \mathcal{S}_{saved} + n;\ lp := sp;\ ++pc$
PushVar v	$S[sp] := S[fp+v];\ ++sp;\ ++pc$
PushVarF $d\ v$	$fp' := fp;\ \textbf{while}\ d\ \neq\ 0\ \textbf{do}\ fp' := S[fp'-2];\ --d\ \textbf{endwhile};$ $S[sp] := S[fp'+v];\ ++sp;\ ++pc$
MakeVar v	$S[fp+v] := S[sp-1];\ --sp;\ ++pc$

Figure 8: Abstract Machine Instructions: Stack operations

Finally, two instructions build structured values. **MkCon** takes two arguments from the instruction stream, a constructor tag and a number of arguments, and builds the corresponding constructor in the heap using the relevant number of arguments from the stack. **MkTuple** is similar except that it has no constructor tag parameter, and builds a tuple rather than a constructor. **MkFun** takes the top p elements from the stack and builds a function closure in the heap. The construction of an exception closure is encoded in the **Raise** instruction, together with the transfer of control to the exception handler of the current box.

2.3.2 Stack Operations (Fig 8).

The abstract machine uses a number of simple stack manipulation operations. **Push** increments the stack pointer by a constant. This is used to create fixed space on the stack. **Pop** decrements the stack pointer. **Slide** pops the stack by a fixed amount, and ensures that the top of stack after the stack is the same as before the pop. **SlideVar** and **SlideVarF** perform the same operation but read the amount for the slide from a local or non-local variable, respectively. This is used, for example, to remove the arguments to a function when it returns. **Copy** duplicates the contents of a stack location relative to the current top of stack. Finally, **CopyArg** copies the specified box or function argument to the top of stack.

Three operations are provided on variables. **PushVar** copies the specified variable to the top of stack. **PushVarF** does the same, but from a non-local target stack frame (specified as the depth d in the frame list and the relative offset v, both given in the instruction). **MakeVar** assigns the value on the top of the stack to the corresponding local variable v.

2.3.3 Control-flow Operations (Fig 9).

The abstract machine control instructions are conventional. **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, saving the current ruleset on the stack for future use. The new ruleset and program counter are derived from the label for the function that is called. **CallVar** and **CallVarF** call the function which is pointed to by the a local or non-local variable, respectively. This variable must point to a Fun closure in the heap, which contains a pointer to the code to be executed, as well as a list of already provided arguments. **Ap** calls the function pointed to by the top-of-stack element. **CallPrim1** and **CallPrim2** call primitive functions with one or two arguments, respectively.

2.3.4 Matching Operations (Fig 10).

Matching is initiated by the **StartMatches** instruction, which sets the program counter to the first rule in the base rule set.

The same 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, and matches with a closing **MatchedRule** indicating the end of a matching block; the **MatchAvailable** and **MatchNone** operations which check box input availability; and the value matching operations such as **MatchBool** etc.

Matching takes place against a special stack pointer, the match pointer *mp*, which records the current match position. This is initialised in **MatchRule** to be just above the first argument to the box or function. The input pointer *inp* is also initialised to support input availability checking. Finally, the program counter is initialised to the start of the next rule.

The availability checking operations are used only for box inputs. **MatchAvailable** checks whether the next box input is available. If not, then the entire rule match fails, and the next rule is tried.

Goto l	$pc := l$
If l	**if** $S[sp-1] := true$ **then** $pc := l$ **else** $++pc$ **endif**; $--sp$
Call f	$S[sp++] := pc+1$; $S[sp++] := fn$; $S[sp++] := slp$; $slp := sp+1$; $fn := f.ruleset$; $fn.rp := 0$; $pc := fn.rules[0]$
TailCall $f\ n\ d\ sz$	**while** $d \neq 0$ **do** $fp := S[fp-2]$; $--d$ **endwhile**; **for** i **in** $1..n$ **do** $S[fp - \mathcal{S}_{saved} - i] := S[sp-i]$ **endfor** $sp := fp + sz$; $lp := sp$; $fn := f.ruleset$; $fn.rp := 0$; $pc := fn.rules[0]$
CallVarF $d\ v\ n$ CallVar $v\ n$ Ap n	$\left\{ \begin{array}{ll} \textbf{while}\ d \neq 0\ \textbf{do}\ fp := S[fp-2];\ --d\ \textbf{endwhile};\ c := S[fp+v] & \textbf{if}\ \text{CallVarF} \\ c := S[fp+v] & \textbf{if}\ \text{CallVar} \\ c := S[sp] & \textbf{if}\ \text{Ap} \end{array} \right.$ $f := H[c+1]$; $m := H[c+2]$; $p := H[c+3]$ **if** $(n+p \geq m)$ **then** **for** i **in** $p-1..0$ **do** $S[sp++] := H[c+4+i]$ **endfor**; $S[sp++] := pc$; $S[sp++] := fn$; $S[sp++] := slp$; $fn.rp := 0$; $pc := H[c+1]$; **else** $H[hp] := \mathsf{Fun}\ f\ m\ (p+n)\ H[c+4]\ \ldots\ H[c+4+(p-1)]\ S[sp-1]\ \ldots\ S[sp-n-1]$; $hp := hp+p+n+4$; **endif**;
Return	$fp := S[fp-1]$; $slp := S[fp-2]$; $fn := S[fp-3]$; $pc := S[fp-4]$; $sp' := fp - \mathcal{S}_{saved}$; $S[sp'] := S[sp-1]$; $sp := sp'+1$
CallPrim1 p	$S[sp-1] := p\ (S[sp])\ (S[sp-1])$; $++pc$
CallPrim2 p	$S[sp-2] := p\ (S[sp])\ (S[sp-1])\ (S[sp-2])$; $--sp$; $++pc$
Raise x	$H[hp] := \text{Exn}\ x\ S[sp]$; $--sp$; $sp := fp := slp := 0$; $S[sp++] := hp$; $hp := hp + \mathcal{H}_{\mathsf{Exn}} + 1$; $sp := sp + \mathcal{S}_{saved}$; $pc := fn.rules[fn.handler]$

Figure 9: Abstract Machine Instructions: Control-flow

MatchRule	$mp := fp - \mathcal{S}_{saved} + 1;\ inp := 0;\ pc := fn.rules[fn.rp];\ ++fn.rp;\ sp := lp$
MatchedRule	$sp := lp$
MatchNone	$--mp;\ ++inp;\ ++pc$
MatchAvailable	**if** $ins[inp].available$ **then** $++pc$ **else** $pc := fn.rules[fn.rp]$ **endif**; $inp := inp + 1$
MatchBool b	$--mp;$ **if** $H[S[mp]] \neq$ Bool b **then** $pc := fn.rules[fn.rp]$ **endif**
MatchChar c	$--mp;$ **if** $H[S[mp]] \neq$ Char c **then** $pc := fn.rules[fn.rp]$ **endif**
MatchString s	$--mp;$ **if** $H[S[mp]] \neq$ Str s **then** $pc := fn.rules[fn.rp]$ **endif**
MatchInt i	$--mp;$ **if** $H[S[mp]] \neq$ Int i **then** $pc := fn.rules[fn.rp]$ **endif**
MatchFloat f	$--mp;$ **if** $H[S[mp]] \neq$ Float f **then** $pc := fn.rules[fn.rp]$ **endif**
MatchCon $c\ n$	$--mp;$ **if** $H[S[mp]] \neq$ Constr $c\ n\ xs$ **then** $pc := fn.rules[fn.rp]$ **endif**
MatchTuple n	$--mp;$ **if** $H[S[mp]] \neq$ Tuple $n\ xs$ **then** $pc := fn.rules[fn.rp]$ **endif**
MatchExn x	$--mp;$ **if** $H[S[mp]] \neq$ Exn $x\ xs$ **then** $pc := fn.rules[fn.rp]$ **endif**
Unpack	$--sp;$ **if** $H[S[sp]] =$ Tuple $n\ xs$ **then** $offset := 2$ **else if** $H[S[sp]] =$ Constr $c\ n\ xs$ **then** $offset := 3$; **else if** $H[S[sp]] =$ Fun $f\ m\ n$ **then** $offset := 4$; **endif**; **for** $i := 0$ **to** $n - 1$ **do** $S[sp] := H[S[sp] + offset + i];\ ++sp$; **endfor**; $++pc$
StartMatches	$pc := base.rules[0]$
Reorder	$n := fn.nrules - 1;\ r := fn.rules[fn.rp];$ **for** $i := fn.rp$ **to** n **do** $fn.rules[i] := fn.rules[i + 1]$ **endfor**; $fn.rules[n] := r;\ ++pc$

Figure 10: Abstract Machine Instructions: Rule matching

CopyInput n	$S[sp] := copy\ (ins[n]);\ ++sp;\ ++pc$
Consume n	$ins[n].available := false; ++pc$
MaybeConsume n	**if** $ins[n].available$ **then** $ins[n].available := false$ **endif**; $++pc$

CheckOutputs	**for** $i\ :=\ 0$ **to** $nouts$ **do**
	if $H[S[sp-i-1]]\ \neq$ None **and** $\neg\ outs[i].available$ **then**
	$blocked := true;\ bon := i;\ pcr := pc;\ reschedule;$
	endif;
	endfor; $++pc$

Write n	$--sp$; **if** $H[S[sp]]\ \neq$ None **then** $outs[n] := copy(H[S[sp]])$; **endif**; $++pc$
Input	$H[hp] :=$ Char $getchar;\ S[sp] := hp;\ ++sp;$ $hp := hp + \mathcal{H}_{char};\ ++pc$
Output	$putvalue(S[sp-1]);\ --sp;\ ++pc$
Within $h\ t$	$fn.rules[fn.within\_handler] := h; S[sp] := lo(t);; S[sp+1] := hi(t); sp := sp+2$
WithinStackSpace $h\ p$	$fn.rules[fn.withinspace\ \_handler] := h; S[sp+1] := splim; splim := sp+p; sp := sp+1$
WithinHeapSpace $h\ p$	$fn.rules[fn.withinspace\ \_handler] := h; S[sp+1] := hplim; hplim := hp+p; sp := sp+1$
DoneWithin	$unsetTimer()$
RaiseWithin $h\ t$	$setTimer(S[sp], S[sp-1])$
DoneWithinStackSpace h	$splim := S[sp-1]; S[sp-1] := S[sp]; sp := sp-1$
DoneWithinHeapSpace h	$hplim := S[sp-1]; S[sp-1] := S[sp]; sp := sp-1$
Schedule	$reschedule$

Figure 11: Abstract Machine Instructions: Scheduling and Wire I/O

Otherwise the input pointer *inp* is incremented. **MatchNone** simply increments *inp* without checking input availability. This is used to implement ***** and **_*** in patterns.

The match operations match the heap value pointed to by *mp* against the specified value. Failure means the next rule is tried. Otherwise the next match position is checked. Constructors and tuple matches are applied only to the outer level. Nested matching is achieved by unpacking the arguments to the constructor or tuple onto the stack using the **Unpack** instruction.

Finally, after successful matching, rules may be reordered by **Reorder** if fair matching is required. This is ensured by moving the successful rule to the end of the ruleset. As a consequence, a least recently used policy is implemented.

2.3.5 Scheduling, input and output operations (Fig 11).

Box input and output is handled by two sets of operations. The **CopyInput** 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 semaphore for the appropriate input wire, thereby permitting new write operations on that wire.

Output is handled by two similar 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 operations will succeed. This is achieved by checking that all output wire buffers are empty (as indicated by the wire's *availability* semaphore). If not, then the box blocks

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

Figure 12: Abstract Machine Pseudo-Instructions

`sp := 0`	stack pointer
`hp := 0`	heap pointer
`fp := 0`	frame pointer
`slp := 0`	function frame pointer
`inp := 0`	in pointer
`fn := base`	code base
`pc := INITPC`	program counter
`blocked := false`	blocker flag
`splim := SPLIM`	end of stack area
`hplim := HPLIM`	end of heap area

Figure 13: Box initialisation

until the value on the wire has been consumed, and the *availability* semaphore 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 *availability* semaphore is ignored by **CheckOutputs**.

Two operations are provided to manage stream input/output. A special box is attached to each stream input and output device. Executing the **Input** operation blocks the box if no input is available. Otherwise the input is read into the box's heap, and can then be written to its output wire using normal abstract machine operations. The **Output** operation simply writes the value on the top of stack to the appropriate device.

Control is returned to the scheduler either when a box blocks as a consequence of being unable to write some output during the **CheckOutputs** operation, or explicitly when a box terminates as a consequence of exiting the **Schedule** operation. In either case, the scheduler will select a new runnable box to execute. If there is no runnable box, then in the concurrent 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 box.

2.3.6 Box initialisation (Fig 13).

When a box is scheduled, its registers are initialised as shown in Figure 13. The initialisation during the wire input initialisation phase is similar except that different base and `INITPC` values are used, corresponding to the *_init* code for the box. Registers other than *blocked* and *pc* are not initialised if the box is restarted after it is blocked. In this case, *blocked* is set to *false* and *pc* is set to *pcr*. The registers *splim* and *hplim* hold the upper bounds for the stack and the heap areas, initialised with the constants `SPLIM` and `HPLIM`. These registers can be temporarily modified by the family of **Within** instructions.

3 Cost Modelling via Resource Algebras

As the basis for reasoning about resource consumption in Hume programs, we now present cost models of the HAM for heap space, stack space and time consumption. One of our main design goals is to separate the tracking of these resources as far as possible from the description of the functional behaviour of the abstract machine. We therefore use the concept of *resource algebras* [ABM06] as introduced in the MRG project.

A *resource algebra* $\mathcal{R}$ is a partially ordered monoid $(R, 0, +, \leq)$, i.e. $(R, 0, +)$ is a monoid and $(R, \leq)$ is a partially ordered set, where 0 is the minimum element, and + is order preserving on both sides. Moreover, $\mathcal{R}$ has constants in R for each expression former of the language. These constants denote the costs of the corresponding expression. In general, these constants are parameterised with the current state of the computation, in particular the current stack and heap. These costs are combined by the rules of the operational semantics in Section 4 with the + operator.

3.1 A Resource Algebra for Heap Space

In instantiating the concept of a resource algebra with heap costs, we use the natural numbers $\mathbb{N}$ as domain R, with addition as +, the natural number zero as 0 and the less-or-equal relation as $\leq$. Table 2 defines the changes in heap size for all instructions in the HAM language. The tables $\mathcal{P}rim1_{s,\eta,f}$ and $\mathcal{P}rim2_{s,\eta,f}$ define the heap costs for unary and binary primitive operations, with the current stack s, heap η, and the primitive function f as arguments.

The heap consumption of the constructors is defined by the constants in Table 1. The functions *ssize* returns the length of a string. For tuples, constructors, exceptions and function closures, the number of arguments has to be added to the header size. In the case of an exception this is always 1. The stack operations do not consume any heap space. Matching operations only compare values, and therefore consume neither heap nor stack space. An input operation allocates a new character in the heap and therefore consumes one character cell. Raising an exception involves first the construction of an exception closure. The 1 reserves space for the runtime value passed via the exception.

For primitive operations we use a table mapping the operation, and its runtime arguments, to the corresponding heap consumption: $\mathcal{P}rim1_{s,\eta,f}, \mathcal{P}rim2_{s,\eta,f}$. The notation for manipulating stacks is explained in Figure 14.

3.2 A Resource Algebra for Stack Space

For modelling stack space we again use $(\mathbb{N}, +, \leq)$ as resource algebra. In this case the constants are defined in Table 3.

For all heap operations the stack size increases by 1, since a pointer to the newly allocated heap object is left on the stack. Pushing adds and popping subtracts 1 from the stack size. The slide operations reduce the stack size. In the case of `Slide` the amount is part of the instruction, in the case of `SlideVar` and `SlideVarF` the amount is the value of a local or non-local variable, respectively. For a non-local variable this means that i stack frames are popped before a variable lookup is made to the j-th variable in the *locals* component of the frame. The notation $\mid s^{vals} \mid$ is used to denote the length of the *vals* component in the topmost stack frame. In `MatchRule` and `MatchedRule` this component can be discarded, hence the negative value.

Copy operations add a pointer onto the top of the stack. A `CreateFrame` instructions pushes as many (dummy) elements onto the stack as prescribed by its argument i. Matching operations only compare values, and therefore consume neither heap nor stack space. A `Write` operation writes the heap elemented, pointed to by the top-of-stack element, to an output stream and then pops the pointer from the stack.

$$\mathcal{R}^{\texttt{MkBool b}}_{s,\eta} = \mathcal{H}_{\mathsf{Bool}} \qquad \mathcal{R}^{\texttt{MkChar x}}_{s,\eta} = \mathcal{H}_{\mathsf{Char}}$$
$$\mathcal{R}^{\texttt{MkString x}}_{s,\eta} = \mathcal{H}_{\mathsf{Str}} + \mathit{ssize}\ s \qquad \mathcal{R}^{\texttt{MkInt i}}_{s,\eta} = \mathcal{H}_{\mathsf{Int}}$$
$$\mathcal{R}^{\texttt{MkFloat f}}_{s,\eta} = \mathcal{H}_{\mathsf{Float}} \qquad \mathcal{R}^{\texttt{MkTuple i}}_{s,\eta} = \mathcal{H}_{\mathsf{Tuple}} + i$$
$$\mathcal{R}^{\texttt{MkCon i j}}_{s,\eta} = \mathcal{H}_{\mathsf{Constr}} + j \qquad \mathcal{R}^{\texttt{MkFun f m p}}_{s,\eta} = \mathcal{H}_{\mathsf{Fun}} + p$$
$$\mathcal{R}^{\texttt{MkNone}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Push i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Pop i}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{Slide i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{SlideVar i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{SlideVarF i}}_{s,\eta} = 0$$

$$\mathcal{R}^{\texttt{Copy i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{CopyArg i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{CreateFrame i}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{PushVar i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{PushVarF d i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MakeVar i}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{Goto lbl}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{If lbl}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Call f}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Return}}_{s,\eta} = 0$$

$$\mathcal{R}^{\texttt{StartMatches}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchRule}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchedRule}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{MatchNone}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchAvailable}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{AVAILSET}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{MatchAny}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchBool b}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchChar x}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{MatchString x}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchInt i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchFloat i}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{MatchTuple i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchCon i j}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{MatchExn x}}_{s,\eta} = 0$$

$$\mathcal{R}^{\texttt{Unpack}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Reorder}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{CheckOutputs}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{CopyInput i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{COPYALLINPUTS}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Consume i}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{MaybeConsume i}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{CONSUMESET}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Write i}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{Input}}_{s,\eta} = \mathcal{H}_{\mathsf{Char}} \qquad \mathcal{R}^{\texttt{Output}}_{s,\eta} = 0 \qquad \mathcal{R}^{\texttt{Schedule}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{Raise i}}_{s,\eta} = \mathcal{H}_{\mathsf{Exn}} + 1$$

$$\mathcal{R}^{\texttt{TailCall f n d sz}}_{s,\eta} = 0$$
$$\mathcal{R}^{\texttt{CallVar f n}}_{s,\eta} = \begin{cases} 0 & \text{if } \mathit{arity}\ s_f^{\mathit{locals}} \leq n + \mathit{providedArgs}\ s_f^{\mathit{locals}} \\ n+4 & \text{otherwise} \end{cases}$$
$$\mathcal{R}^{\texttt{CallVarF d n}}_{s,\eta} = \begin{cases} 0 & \text{if } \mathit{arity}\ (s - \square^i)_j^{\mathit{locals}} \leq n + \mathit{providedArgs}\ (s - \square^i)_j^{\mathit{locals}} \\ n+4 & \text{otherwise} \end{cases}$$
$$\mathcal{R}^{\texttt{CallPrim1 f}}_{s,\eta} = \mathcal{P}\mathit{rim1}_{s,\eta,f}\ s_0 \qquad \mathcal{R}^{\texttt{CallPrim2 f}}_{s,\eta} = \mathcal{P}\mathit{rim2}_{s,\eta,f}\ s_0(s - \top_0)$$

Table 2: Heap consumption of HAM instructions

An `Unpack` operation simply consumes the top-of-stack pointer unless it is a tuple or constructor. In the latter cases, it thereafter adds all the elements of the tuple or constructor onto the stack. In a `Raise` instruction the entire stack is discarded, a new frame is built on the empty stack and a pointer to an exception closure is pushed onto the stack. Therefore, we have to subtract the entire stack size, computed as a sum over the sizes of all stack frames, and then add the frame header size plus 1. In a `TailCall` instruction the top n frames are discarded, a new frame header created, and o dummy values (for local variables) added to the stack. A `Return` operation removes the topmost frame from the stack. The size of the stack frame is the frame header size plus the number of local variables plus the size of the expression stack. The stack consumption of a unary primitive operation is 0, because it overwrites its argument on the stack. For a binary primitive operation the stack consumption is -1 because both arguments can be discarded and only the result value is pushed.

3.3 A Resource Algebra for Time

For modelling time consumption we use rational numbers as the base domain with the usual addition and less-or-equal relation $(\mathbb{R}, +, \leq)$ as resource algebra.

In Table 4 we summarise upper bounds for execution time on our target processor, a Renesas M32C. We have derived these bounds by using AbsInt's aiT tool [**?**], which performs machine-code-level analysis

$\mathcal{R}^{\texttt{MkBool b}}_{s,\eta} =$
$\mathcal{R}^{\texttt{MkChar x}}_{s,\eta} =$
$\mathcal{R}^{\texttt{MkString x}}_{s,\eta} =$
$\mathcal{R}^{\texttt{MkInt i}}_{s,\eta} =$
$\mathcal{R}^{\texttt{MkFloat f}}_{s,\eta} =$
$\mathcal{R}^{\texttt{GETCONST}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MkTuple i}}_{s,\eta} = -n+1$
$\mathcal{R}^{\texttt{MkCon i j}}_{s,\eta} = -n+1$
$\mathcal{R}^{\texttt{MkVector i j}}_{s,\eta} = -n+1$
$\mathcal{R}^{\texttt{MkFun f m p}}_{s,\eta} = -p+1$
$\mathcal{R}^{\texttt{MkNone}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{Push i}}_{s,\eta} = i$
$\mathcal{R}^{\texttt{PushVar i}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{Pop i}}_{s,\eta} = -i$
$\mathcal{R}^{\texttt{Slide i}}_{s,\eta} = -i$
$\mathcal{R}^{\texttt{SlideVar i}}_{s,\eta} = -(\eta\ s_i^{locals})$
$\mathcal{R}^{\texttt{SlideVarF i j}}_{s,\eta} = -((s-\square^i)^{locals}_j)$

$\mathcal{R}^{\texttt{Copy i}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{CopyArg i}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{CreateFrame i}}_{s,\eta} = i+1$
$\mathcal{R}^{\texttt{PushVar i}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{PushVarF d i}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{MakeVar i}}_{s,\eta} = -1$
$\mathcal{R}^{\texttt{Goto lbl}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{If lbl}}_{s,\eta} = -1$
$\mathcal{R}^{\texttt{Call f}}_{s,\eta} = 3$

$\mathcal{R}^{\texttt{StartMatches}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchRule}}_{s,\eta} = -\mid s^{vals} \mid$
$\mathcal{R}^{\texttt{MatchedRule}}_{s,\eta} = -\mid s^{vals} \mid$
$\mathcal{R}^{\texttt{MatchNone}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchAvailable}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{AVAILSET}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchNone b}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchBool b}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchChar x}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchString x}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchInt i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchFloat i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchTuple i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchCon i j}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchVector i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MatchExn x}}_{s,\eta} = 0$

$\mathcal{R}^{\texttt{Reorder}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{CheckOutputs}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{CopyInput i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{COPYALLINPUTS i}}_{s,\eta} = i$
$\mathcal{R}^{\texttt{Consume i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{MaybeConsume i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{CONSUMESET i}}_{s,\eta} = 0$
$\mathcal{R}^{\texttt{Write i}}_{s,\eta} = -1$
$\mathcal{R}^{\texttt{Input}}_{s,\eta} = 1$
$\mathcal{R}^{\texttt{Output}}_{s,\eta} = -1$
$\mathcal{R}^{\texttt{Schedule}}_{s,\eta} = 0$

$$\mathcal{R}^{\texttt{TailCall f n sz}}_{s,\eta} = -\mid s^{locals} \mid$$

$$\mathcal{R}^{\texttt{CallVar f n}}_{s,\eta} = \begin{cases} providedArgs\ s_f^{locals} + 3 & \text{if } arity\ s_f^{locals} \leq n\ + providedArgs\ s_f^{locals} \\ -n+1 & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{CallVarF f d n}}_{s,\eta} = \begin{cases} providedArgs\ s_f^{locals} + 3 & \text{if } arity(s-\square^i)^{locals}_j \leq n + providedArgs\ (s-\square^i)^{locals}_j \\ -n+1 & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{Unpack}}_{s,\eta} = \begin{cases} n-1 \text{ if } \eta\ s_0 = \mathsf{Tuple}\ n\ xs \\ n-1 \text{ if } \eta\ s_0 = \mathsf{Constr}\ c\ n\ xs \\ n-1 \text{ if } \eta\ s_0 = \mathsf{Fun}\ f\ m\ n\ xs \\ -1 \text{ otherwise} \end{cases}$$

$$\mathcal{R}^{\texttt{Return}}_{s,\eta} = -(\mathcal{S}_{saved} + \mid s^{locals} \mid + \mid s^{vals} \mid) + 1$$

$$\mathcal{R}^{\texttt{CallPrim1 f}}_{s,\eta} = 0$$

$$\mathcal{R}^{\texttt{CallPrim2 f}}_{s,\eta} = -1$$

$$\mathcal{R}^{\texttt{Raise x}}_{s,\eta} = \mathcal{S}_{saved} + 1 - \Sigma_{0 \leq i < \infty}(\mathcal{S}_{saved} + \mid (s-\square^i)^{locals} \mid + \mid (s-\square^i)^{vals} \mid)$$

$$\mathcal{R}^{\texttt{TailCall f n d sz}}_{s,\eta} = \mathcal{S}_{saved} + sz - \Sigma_{0 \leq i \leq d}(\mathcal{S}_{saved} + \mid (s-\square^i)^{locals} \mid + \mid (s-\square^i)^{vals} \mid)$$

Table 3: Stack consumption of HAM instructions

of worst case execution time. Details on these measurements are reported in Deliverable D14 [**?**]. This tool accounts for low-level architecture issues such as the states of the pipeline and, if present, of the cache. However, comparisons of the sum of HAM instruction costs with analysing entire basic blocks have shown that these low-level architecture issues have little influence on the overall costs on the M32C processor. We therefore don't model these low-level issues in this resource algebra for time and compute the costs of a basic block as the sum of the costs of its HAM instructions. In the future, we might consider a system that uses the aiT tool on basic blocks of the program under consideration.

The current version of the Hume native code-generator pre-allocates constants in the text area of the compiled code. Therefore, all `Mk...` instructions in the HAM code are replaced by the new `GETCONSTANT` HAM instruction, which only fetches a pointer to the pre-allocated constant. In the costs

for the `Return` instruction, f indicates that the current function, from which to return, is the f-th function in the program. This is due to the fact that the `Return` on the M32C uses a switch statement over all possible return points. The HAM instruction `AVAILSET n` is a new instruction that checks the availability of n input wires. The HAM instruction `CONSUMESET n` is a new instruction that consumes the values from n input wires after a successful match.

Table 5 summarises the bounds on execution time for primitive functions used in the Hume compiler. They have been derived by using the aiT tool.

$\mathcal{R}^{\texttt{MkBool b}}_{s,\eta} = 85$	$\mathcal{R}^{\texttt{Copy i}}_{s,\eta} = 31$	$\mathcal{R}^{\texttt{StartMatches}}_{s,\eta} = 111$	
$\mathcal{R}^{\texttt{MkChar x}}_{s,\eta} = 84$	$\mathcal{R}^{\texttt{CopyArg i}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{MatchRule}}_{s,\eta} = 20$	
$\mathcal{R}^{\texttt{MkInt i}}_{s,\eta} = 83$	$\mathcal{R}^{\texttt{CreateFrame i}}_{s,\eta} = 72$	$\mathcal{R}^{\texttt{MatchedRule}}_{s,\eta} = 10$	$\mathcal{R}^{\texttt{Reorder}}_{s,\eta} =$
$\mathcal{R}^{\texttt{MkFloat f}}_{s,\eta} = 91$	$\mathcal{R}^{\texttt{PushVar i}}_{s,\eta} = 39$	$\mathcal{R}^{\texttt{MatchNone}}_{s,\eta} = 11$	$\mathcal{R}^{\texttt{CheckOutputs}}_{s,\eta} = 602$
$\mathcal{R}^{\texttt{GETCONST i}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{PushVarF d i}}_{s,\eta} = 11d + 35$	$\mathcal{R}^{\texttt{MatchAny}}_{s,\eta} = 11$	$\mathcal{R}^{\texttt{CopyInput i}}_{s,\eta} = 78$
$\mathcal{R}^{\texttt{MkTuple i}}_{s,\eta} = 52i + 78$	$\mathcal{R}^{\texttt{MakeVar i}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{MatchAvailable}}_{s,\eta} = 13$	$\mathcal{R}^{\texttt{COPYALLINPUTS i}}_{s,\eta} = 4 + 74i$
$\mathcal{R}^{\texttt{MkCon i j}}_{s,\eta} = 54i + 107$	$\mathcal{R}^{\texttt{Goto lbl}}_{s,\eta} = 3$	$\mathcal{R}^{\texttt{AVAILSET n}}_{s,\eta} = 6 + 7n$	$\mathcal{R}^{\texttt{Consume i}}_{s,\eta} = 31$
$\mathcal{R}^{\texttt{MkFun f m p}}_{s,\eta} = 60p + 166$	$\mathcal{R}^{\texttt{Call f}}_{s,\eta} = 70$	$\mathcal{R}^{\texttt{MatchBool b}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{MaybeConsume i}}_{s,\eta} = 28$
$\mathcal{R}^{\texttt{MkNone}}_{s,\eta} = 25$	$\mathcal{R}^{\texttt{Return}}_{s,\eta}{}^{1} = 51 + 15f$	$\mathcal{R}^{\texttt{MatchChar x}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{CONSUMESET i}}_{s,\eta} = 32i$
$\mathcal{R}^{\texttt{Push i}}_{s,\eta} = 9$		$\mathcal{R}^{\texttt{MatchString x}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{Write i}}_{s,\eta} =$
$\mathcal{R}^{\texttt{Pop i}}_{s,\eta} = 9$		$\mathcal{R}^{\texttt{MatchInt i}}_{s,\eta} = 32$	$\mathcal{R}^{\texttt{Input}}_{s,\eta} =$
$\mathcal{R}^{\texttt{Slide i}}_{s,\eta} = 53$		$\mathcal{R}^{\texttt{MatchFloat i}}_{s,\eta} = 35$	$\mathcal{R}^{\texttt{Output}}_{s,\eta} =$
$\mathcal{R}^{\texttt{SlideVar i}}_{s,\eta} = 76$		$\mathcal{R}^{\texttt{MatchTuple i}}_{s,\eta} = 11$	$\mathcal{R}^{\texttt{Schedule}}_{s,\eta} = 602$
$\mathcal{R}^{\texttt{SlideVarF i j}}_{s,\eta} = 11j + 79$		$\mathcal{R}^{\texttt{MatchCon i j}}_{s,\eta} = 30$	$\mathcal{R}^{\texttt{Raise i}}_{s,\eta} = 377$
		$\mathcal{R}^{\texttt{MatchExn x}}_{s,\eta} = 30$	
		$\mathcal{R}^{\texttt{MatchVector i}}_{s,\eta} = 11$	

$\mathcal{R}^{\texttt{TailCall f n d sz}}_{s,\eta} = 22 + 36d + 27n$
$\mathcal{R}^{\texttt{CallVar n}}_{s,\eta} = 13$
$\mathcal{R}^{\texttt{CallVarF d n}}_{s,\eta} = 13d$
$\mathcal{R}^{\texttt{MkString x}}_{s,\eta} = 13 \mid x \mid +140$
$\mathcal{R}^{\texttt{Unpack x}}_{s,\eta} = 44 \mid x \mid +51$
$\mathcal{R}^{\texttt{CopyInput i}}_{s,\eta} = \begin{cases} 78 \text{ if Int} \\ 74 \mid s \mid +4 \text{ if Str} \end{cases}$
$\mathcal{R}^{\texttt{If lbl}}_{s,\eta} = 30$
$\mathcal{R}^{\texttt{CallPrim1 f}}_{s,\eta} = \mathcal{P}rim1_{s,\eta,f}\ s_0$
$\mathcal{R}^{\texttt{CallPrim2 f}}_{s,\eta} = \mathcal{P}rim2_{s,\eta,f}\ s_0(s - \top_0)$

Table 4: WCET of HAM instructions on a M32C (in cycles)

4 Formalisation of the HAM

The goal of this section is to give a formal specification of the HAM semantics, corresponding to the behaviour of the references implementation in the previous section. This formalisation is the basis of an HAM operational semantics that is encoded in Isabelle. This encoding is developed alongside this document and will be the basis for automated certification of Hume code as needed in WP4.

==.c = 125					
==.n = 125	>.c = 124		/.n = 206		@ = 89
==.w = 125	>.n = 124		/.w = 206		vecdef =
==.i = 125	>.w = 124		/.i = 206		vecmap =
==.f = 125	>.i = 124	+.n = 116	/.f = 959	toInt.i = 2973	update =
==.s = 125	>.f = 209	+.w = 116	/ = 959	toInt.n = 2973	vecmake =
==.e = 125	>.s = 209	+.i = 116	mod = 1294	toInt.w = 2973	sqrt =
== = 125	> = 209	+.f = 1106	mod.n = 1294	toInt.f = 2973	ln =
!=.c = 243	<=.c = 122	+ = 1106	mod.w = 1294	toInt = 2973	log10 =
!=.n = 243	<=.n = 122	-.n = 116	mod.i = 1294	toFloat.n = 699	sin =
!=.w = 243	<=.w = 122	-.w = 116	\% = 1294	toFloat.w = 699	cos =
!=.i = 243	<=.i = 122	-.i = 116	div.n = 206	toFloat.i = 699	tan =
!=.f = 243	<=.f = 211	-.f = 1112	div.w = 206	toFloat.f = 699	asin =
!=.s = 243	<=.s = 211	- = 1112	div.i = 206	toFloat = 699	acos =
!=.e = 243	<= = 211	*.n = 124	div = 206	toChar =	atan =
!= = 243	<.c = 124	*.w = 124	&& = 153	show =	sinh =
>=.c = 122	<.n = 124	*.i = 124	|| = 156	length =	cosh =
>=.n = 122	<.w = 124	*.f = 356	^& = 151	++ =	tanh =
>=.w = 122	<.i = 124	* = 356	^| = 151	++.s =	atan2 = 14305
>=.i = 122	<.f = 209		^ = 151		** =
>=.f = 211	<.s = 209		~ = 151		exp =
>=.s = 211	< = 209		not = 118		
>= = 211					

Table 5: WCET of primitive operations on a M32C (in cycles)

4.1 Basic Definitions

As basic types we use *Locn* as locations into the heap, *Ref* as references (either a proper location or a null-pointer *Nullref*). Heap values (*HVal*) are a tagged union of basic types, compound types of tuples or constructors, exceptions or function closures. We use disjoint name spaces for *Label*, *Function* and box names (*BName*).

$$
\begin{array}{lcl}
\mathit{Locn} & \equiv & \mathbb{N} \\
\mathit{Ref} & \equiv & \mathit{Nullref} \mid \mathit{Ref}\ \mathit{Locn} \\
\mathit{HVal} & \equiv & \mathsf{Int}\ \mathbb{I} \mid \mathsf{Bool}\ \mathbb{B} \mid \mathsf{Char}\ \mathbb{R} \mid \mathsf{Str}\ \mathbb{R} \mid \mathit{R}\ \mathit{Ref} \mid \mathsf{Tuple}\ \mathbb{N}\ (\mathit{Ref}\ \mathit{list}) \mid \mathsf{Constr}\ \mathbb{N}\ \mathbb{N}\ (\mathit{Ref}\ \mathit{list}) \mid \mathsf{Fun}\ \mathit{Function}\ \mathbb{N}\ \mathbb{N}\ (\mathit{Ref}\ \mathit{list}) \mid \mathsf{Exn}\ \mathbb{N}\ \mathit{Ref} \mid \mathsf{None}
\end{array}
$$

Stack values are references into the heap. A stack is a list of frame records. A frame record (*Frame*) contains the return address (*ret*), the arguments (*args*), the local variabls (*locals*) and the expression stack (*vals*). By default, stack operations such as push and pop work on the value stack. See Figure 14 for the notation used to access and modify stack entries.

The heap is a finite map of locations to heap values, i.e. a function from locations to either the value *None* or *Some* x, where x is an *HVal*. An IO record collects all state info, not recorded in stack or heap. These are mainly related to wire input/output and rule matching. In particular, name of the box (b), a flag whether the box is blocked (*blocked*), a pointer to the next rule in the rule set of the box

(*rp*), and pointers to the next wire (*inp*) and the next stack value (*mp*) to be checked in a rule match.

$$
\begin{array}{lcl}
\mathit{SVal} & \equiv & \mathit{Ref} \\
\mathit{Frame} & \equiv & (\!|\ \mathit{ret} :: \mathit{Label},\ \mathit{args} :: \mathit{SVal\ list},\ \mathit{locals} :: \mathit{SVal\ list},\ \mathit{vals} :: \mathit{SVal\ list}\ |\!) \\
\mathit{Stack} & \equiv & \mathit{Frame\ list} \\
 & & \\
\mathit{Heap} & \equiv & \mathit{Locn} \rightsquigarrow_f \mathit{HVal} \\
\mathit{Wire} & \equiv & \mathbb{N} \\
\mathit{IO} & \equiv & (\!|\ b :: \mathit{BName},\ \mathit{blocked} :: \mathit{bool},\ \mathit{crp} :: \mathbb{N},\ \mathit{rp} :: \mathbb{N},\ \mathit{inp} :: \mathit{Wire}, \mathit{mp} :: \mathbb{N}\ |\!)
\end{array}
$$

4.2 A Roadmap through the Operational Semantics

The initial set of rules deals with the construction of heap values. The next set encodes basic operations on the stack components. Note that we abstract over several pointers present in the reference implementation, by using a structured stack representation as a list of frames.

The main control-flow operations are (conditional) jumps and calls covered in the rules stepGoto, stepIfTrue, stepIfFalse, stepTailCall, stepCall, stepReturn, stepCallPrim1, stepCallPrim2 at the end of the list of rules. To determine the target of a (conditional) jump the table Σ^{Lab}, mapping labels to instructions sequences, is used. For function calls a similar table Σ^{Fun} and for exceptions Σ^{Exn} is used.

The rules relevant for higher-order functions are stepCallVarSaturated, stepCallVarUnaturated, stepCallVarFSaturated, stepCallVarFUnsaturated, stepApSaturated, stepApUnsaturated, stepSlideVar, stepSlideVarF. A higher-order function can be called from a local variable `CallVar` a non-local variable `CallVarF` or from the top of the stack `Ap`. In each case we distinguish between the case where enough arguments are provided to perform the call, i.e. the call is saturated, or whether another function closure needs to be allocated. The only rule needed for exceptions is stepRaise.

The interface between expression level and system level is defined by the rule stepCheckOutputsFalse. Note that no rules for a `Schedule` as first instruction or an empty instruction list exists. Thus, in this case, no reduction can be done on expression level, which operationally amounts to yielding control to the scheduler and performing a rule on the system level.

4.3 Rules of the Operational Semantics

The following judgement of the small-step semantics on Hume expression level

$$s, \eta, io, \theta \vdash \mathtt{cs} \Downarrow_n^m (s', \eta', io', \mathtt{cs}', p)\ \theta'$$

is read as: with an initial stack s, initial heap η, IO record io and with the wire environment θ, the HAM instruction sequence (a list) cs evaluates in $n - m$ steps to a final stack s', final heap η', a final IO record io', using p resources and leaving the code sequence $\mathtt{cs}'$ as continuation. The new wire environment is θ'. The semantics of resources is intentionally left open in this semantics. It can be instantiated to every structure corresponding to a resource algebra as discussed in Section 3. Being a small-step semantics, the result of one step will be a state with a continuation code sequence.

While the usage of a stack and heap are standard, the IO record collects additional information on the state of a box execution, that is primarily needed during box input/output and matching operations. The pointers *mp* and *inp* are indices into the arguments component of the top stack frame (s^{args}) and into the set of in-wires for the current box. The `MatchBool` etc. instructions check whether the current argument is a boolean value, and if not continue with the next rule. The `CopyInput` instruction is the only one that uses the *inp* pointer in order to copy the value from the current wire into the argument portion of the stack frame.

l	a reference to location l in the heap
fresh S	returns an element x, s.t. $x \notin S$
dom f	the domain of a partial function f, i.e. the values x for which $f\ x$ is defined
s_0	get the top-of-stack element from the stack s
$s + v$	push value v on top of the stack s
$s + xs$	push all elements in the list xs on top of the stack s
$s - \top$	pop the top element from the stack s
$s - \top^n$	pop the n topmost elements from the stack s
$s - \square^i$	pop the top i frames from the stack s
$s + \bot^n$	push n dummy values onto the stack s
$s - \top^n + s_0$	pop elements 1 to n from the stack but keep the top-of-stack
$s_{0..n-1}$	get the elements 0 to n-1 from the stack s
$s_n^{locals} := s_0$	assign top-of-stack to n-th local variable
$s + \boxed{\text{c,f}}$	allocate a new frame (for function f) on the stack, with return address c
$s - \square$	remove the topmost frame from the stack s
$\boxed{\text{f=l}} = s_{\boxed{\text{i}}}$	binds l to the value of field f in the i-th frame on stack s
io_f	access the value of field f in record io
$io(\!\vert\ f := x\ \vert\!)$	set the value of field f in record io to x
$f(x \mapsto y)$	update the function f to map x to y
$wire^{b,m}$	the name of the m-th in-wire of box b
$wire_{b,m}$	the name of the m-th out-wire of box b

Figure 14: Notation used in the operational semantics

The notation used in the rules of the operational semantics are summarised in Figure 14. We use # for list cons, @ for list append, _++ for incrementing an integer variable. The construct $(\!\vert\ \cdot\ \vert\!)$ constructs a record, $r(\!\vert\ x := \cdot\ \vert\!)$ modifies the record r by updating the field x. We use $f\ x$ for applying a function f to argument x, and $f(x \mapsto \cdot)$ for updating the function f with a mapping of x to $\cdot$.

4.3.1 Heap operations

The heap operations are fairly standard. They allocate a fresh location, by calling *fresh* (*dom* η). It is guaranteed by definition that *fresh* $S \notin S$. A pointer to the newly allocated heap object is left on the stack. In the case of tuples and constructors, the top n elements are taken from the stack and put into the heap closure.

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkBool x})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \eta(l \mapsto (\mathsf{Bool}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkBool x}}_{s,\eta})\ \theta} \quad (\textsc{stepMkBool})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkChar x})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \eta(l \mapsto (\mathsf{Char}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkChar x}}_{s,\eta})\ \theta} \quad (\textsc{stepMkChar})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkString x})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \eta(l \mapsto (\mathsf{Str}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkString x}}_{s,\eta})\ \theta} \quad (\textsc{stepMkString})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkInt x})\#\texttt{cs} \Downarrow^n_{n+1} ((s + l, \eta(l \mapsto (\mathsf{Int}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkInt x}}_{s,\eta})\ \theta} \quad (\textsc{stepMkInt})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkFloat x})\#\texttt{cs} \Downarrow^n_{n+1} ((s + l, \eta(l \mapsto (\mathsf{Float}\ x)), io, \texttt{cs}, \mathcal{R}^{\texttt{MkFloat x}}_{s,\eta})\ \theta} \quad (\textsc{stepMkFloat})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkTuple j})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^j + l, \eta(l \mapsto (\mathsf{Tuple}\ x\ s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{MkTuple j}}_{s,\eta})\ \theta} \quad (\textsc{stepMkTuple})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkCon i j})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^j + l, \eta(l \mapsto (\mathsf{Constr}_i\ j\ s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{MkCon i j}}_{s,\eta})\ \theta} \quad (\textsc{stepMkCon})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkFun f m p})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^p + l, \eta(l \mapsto (\mathsf{Fun}\ f\ m\ p\ (s_{0..p-1}))), io, \texttt{cs}, \mathcal{R}^{\texttt{MkFun f m p}}_{s,\eta})\ \theta} \quad (\textsc{stepMkFun})$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{MkNone})\#\texttt{cs} \Downarrow^n_{n+1} (s + l, \eta(l \mapsto \mathsf{None}), io, \texttt{cs}, \mathcal{R}^{\texttt{MkNone}}_{s,\eta})\ \theta} \quad (\textsc{stepMkNone})$$

4.3.2 Stack operations

The stack operations usually modify the expression stack, i.e. the *vals* component of the top stack frame. Superscripting accesses a particular component of the top stack frame; subscripting such a component selects an element; e.g. s^{args}_x accesses the argument x of the top stack frame. The notation $s -_{\square}{}^i$ is used for popping the top i frames from stack s. For more details on notation see Figure 14.

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{Push i})\#\texttt{cs} \Downarrow^n_{n+1} (s + \bot^i, \eta, io, \texttt{cs}, \mathcal{R}^{\texttt{Push i}}_{s,\eta})\ \theta} \quad (\textsc{stepPush})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{Pop i})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top^i, \eta, io, \texttt{cs}, \mathcal{R}^{\texttt{Pop i}}_{s,\eta})\ \theta} \quad (\textsc{stepPop})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{Slide i})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top^{i+1} + s_0, \eta, io, \texttt{cs}, \mathcal{R}^{\texttt{Slide i}}_{s,\eta})\ \theta} \quad (\textsc{stepSlide})$$

$$\frac{\mathsf{Int}\ i = \eta\ s^{locals}_x}{s, \eta, io, \theta \vdash (\texttt{SlideVar x})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top^{i+1} + s_0, \eta, io, \texttt{cs}, \mathcal{R}^{\texttt{SlideVar x}}_{s,\eta})\ \theta} \quad (\textsc{stepSlideVar})$$

$$\frac{\mathsf{Int}\ i = \eta\ (s - \square^{x})_{y}^{locals}}{s, \eta, io, \theta \vdash (\texttt{SlideVarF x y})\#\texttt{cs} \Downarrow_{n+1}^{n} ((s - \top^{i+1} + s_0, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{SlideVarF x y}})\ \theta} \quad (\textsc{stepSlideVarF})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{Copy i})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + s_i, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{Copy i}})\ \theta} \quad (\textsc{stepCopy})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{CopyArg i})\#\texttt{cs} \Downarrow_{n+1}^{n} ((s + s_i^{args}, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{CopyArg i}})\ \theta} \quad (\textsc{stepCopyArg})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{CreateFrame i})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + \boxed{\text{c,f}} + \bot^{i}, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{CreateFrame i}})\ \theta} \quad (\textsc{stepCreateFrame})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{PushVar i})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + s_i^{locals}, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{PushVar i}})\ \theta} \quad (\textsc{stepPushVar})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{PushVarF i j})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + (s - \square^{i})_{j}^{locals}, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{PushVarF i j}})\ \theta} \quad (\textsc{stepPushVarF})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{MakeVar i})\#\texttt{cs} \Downarrow_{n+1}^{n} ((s_i^{locals} := s_0) - \top, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{MakeVar i}})\ \theta} \quad (\textsc{stepMakeVar})$$

4.3.3 Control-flow operations

The following set of rules describes the control flow on the expression level of Hume. Here we assume that the compiler has generated for each box several tables, mapping labels, including function names, and exceptions to their corresponding instruction sequences: $\Sigma^{Lab}, \Sigma^{Exn}$. For convenience we drop the box name from the table lookup. Note, that the box name is always available as io_b.

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{Goto l})\#\texttt{cs} \Downarrow_{n+1}^{n} (s, \eta, io, \Sigma_{\texttt{l}}^{Lab}, \mathcal{R}_{s,\eta}^{\texttt{Goto l}})\ \theta} \quad (\textsc{stepGoto})$$

$$\frac{l = s_0 \quad \eta\ l = \mathsf{Bool}\ \mathit{true}}{s, \eta, io, \theta \vdash (\texttt{If lbl})\#\texttt{cs} \Downarrow_{n+1}^{n} (s - \top, \eta, io, \Sigma_{\texttt{lbl}}^{Lab}, \mathcal{R}_{s,\eta}^{\texttt{If lbl}})\ \theta} \quad (\textsc{stepIfTrue})$$

$$\frac{l = s_0 \quad \eta\ l = \mathsf{Bool}\ \mathit{false}}{s, \eta, io, \theta \vdash (\texttt{If lbl})\#\texttt{cs} \Downarrow_{n+1}^{n} (s - \top, \eta, io, \texttt{cs}, \mathcal{R}_{s,\eta}^{\texttt{If lbl}})\ \theta} \quad (\textsc{stepIfFalse})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{Call f})\#(\texttt{Label c})\#\texttt{cs} \Downarrow_{n+1}^{n} (s + \boxed{\text{c,f}}, \eta, io, \Sigma_{\texttt{f}}^{Fun}, \mathcal{R}_{s,\eta}^{\texttt{Call f}})\ \theta} \quad (\textsc{stepCall})$$

$$\frac{s' \quad = \quad ((s - \square^{d}) + \boxed{\text{c,f}})^{args} := s_{0..j-1} + \top^{z}}{s, \eta, io, \theta \vdash (\texttt{TailCall f j d z})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^{n}_{n+1} (s', \eta, io, \Sigma^{Fun}_{\texttt{f}}, \mathcal{R}^{\texttt{TailCall f j d z}}_{s,\eta})\ \theta} \text{(stepTailCall)}$$

$$\frac{\boxed{\text{ret=l}} = s_{\boxed{0}}}{s, \eta, io, \theta \vdash (\texttt{Return})\#\texttt{cs} \Downarrow^{n}_{n+1} ((s - \square + s_0, \eta, io, \Sigma^{Lab}_{l}, \mathcal{R}^{\texttt{Return}}_{s,\eta})\ \theta} \text{(stepReturn)}$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{CallPrim1 f})\#\texttt{cs} \Downarrow^{n}_{n+1} (s - \top^{1} + (f\ s_0), \eta, io, \texttt{cs}, \mathcal{R}^{\texttt{CallPrim1 f}}_{s,\eta})\ \theta} \text{(stepCallPrim1)}$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{CallPrim2 f})\#\texttt{cs} \Downarrow^{n}_{n+1} (s - \top^{2} + (f\ s_0\ s_1), \eta, io, \texttt{cs}, \mathcal{R}^{\texttt{CallPrim2 f}}_{s,\eta})\ \theta} \text{(stepCallPrim2)}$$

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s, \eta, io, \theta \vdash (\texttt{Raise x})\#\texttt{cs} \Downarrow^{n}_{n+1} (\emptyset + l, \eta(l \mapsto (\mathsf{Exn}\ x\ s_0)), io, \Sigma^{Exn}_{x}, \mathcal{R}^{\texttt{Raise x}}_{s,\eta})\ \theta} \text{(stepRaise)}$$

Three HAM instructions can be used for applying a higher-order function to some arguments: `CallVar` for a local variable, `CallVarF` for a non-local variable, and `Ap` for a function closure pointed to by the top-of-stack element. Each of these 3 come in a version for unsaturated application, i.e. the number of arguments provided is smaller than the arity of the function, and in a saturated version. In the former case a new function closure is built in the heap. In the latter case the function is applied to all its arguments.

$$\frac{\mathsf{Fun}\ f\ m\ p\ xs = \eta\ s^{locals}_{i} \quad p + j \geq m}{s, \eta, io, \theta \vdash (\texttt{CallVar i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^{n}_{n+1} ((s - s_{0..j-1} + \boxed{\text{c,f}})^{args} := xs@s_{0..j-1}, \eta, io, \Sigma^{Fun}_{\texttt{f}}, \mathcal{R}^{\texttt{CallVar i j}}_{s,\eta})\ \theta} \text{(stepCallVarSaturated)}$$

$$\frac{\mathsf{Fun}\ f\ m\ p\ xs = \eta\ s^{locals}_{i} \quad p + j < m \quad l = \mathit{fresh}\ \eta}{s, \eta, io, \theta \vdash (\texttt{CallVar i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^{n}_{n+1} (s - s_{0..j-1}, \eta(l \mapsto \mathsf{Fun}\ f\ m\ (p + j)\ (xs@s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{CallVar i j}}_{s,\eta})} \text{(stepCallVarUnsaturated)}$$

$$\frac{\mathsf{Fun}\ f\ m\ p\ xs = \eta\ (s - \square^{d})^{locals}_{i} \quad p + j \geq m}{s, \eta, io, \theta \vdash (\texttt{CallVarF d i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^{n}_{n+1} ((s - s_{0..j-1} + \boxed{\text{c,f}})^{args} := xs@s_{0..j-1}, \eta, io, \Sigma^{Fun}_{\texttt{f}}, \mathcal{R}^{\texttt{CallVarF d i j}}_{s,\eta})\ \theta} \text{(stepCallVarFSaturated)}$$

$$\frac{\mathsf{Fun}\ f\ m\ p\ xs = \eta\ (s - \square^{d})^{locals}_{i} \quad p + j < m \quad l = \mathit{fresh}\ \eta}{s, \eta, io, \theta \vdash (\texttt{CallVarF d i j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^{n}_{n+1} (s - s_{0..j-1}, \eta(l \mapsto \mathsf{Fun}\ f\ m\ (p + j)\ (xs@s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{CallVarF}}_{s,\eta}} \text{(stepCallVarFUnsaturated)}$$

$$\frac{\mathsf{Fun}\ f\ m\ p\ xs = \eta\ s_0 \quad p + j \geq m}{s, \eta, io, \theta \vdash (\texttt{Ap j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} ((s - s_{0..j-1} + \boxed{\text{c,f}})^{args} := xs@s_{0..j-1}, \eta, io, \Sigma^{Fun}_{\texttt{f}}, \mathcal{R}^{\texttt{Ap j}}_{s,\eta})\ \theta}\ (\textsc{stepApSaturated})$$

$$\frac{\mathsf{Fun}\ f\ m\ p\ xs = \eta\ s_0 \quad p + j < m \quad l = \mathit{fresh}\ \eta}{s, \eta, io, \theta \vdash (\texttt{Ap j})\#(\texttt{Label c})\#\texttt{cs} \Downarrow^n_{n+1} (s - s_{0..j-1}, \eta(l \mapsto \mathsf{Fun}\ f\ m\ (p + j)\ (xs@s_{0..j-1})), io, \texttt{cs}, \mathcal{R}^{\texttt{Ap j}}_{s,\eta})\ \theta}\ (\textsc{stepApUnsaturated})$$

4.3.4 Matching, scheduling and I/O operations

The following functions are used to retrieve wire names: $wire^{b,i}$ is the name of the i-th input wire of box b; $wire_{b,i}$ is the name of the i-th output wire of box b. These can be calculated from the static information in Box. Note that box name and wire are usually retrieved from the io record. The function *outputable* checks whether all needed output wires are free, corresponding to the innermost loop in Figure 4.

The rules for matching always compare the heap cell, pointed to by the top-of-stack element, with a value encoded in the operation itself. If the expected kind of heap cell is found, evaluation continues with the next instruction, otherwise the next rule is tried.

In these rules several fields of the IO record are used to perform matching. The field io_{rp} points to the next rule. If a match fails with the current rule, execution will continue at $\Sigma^{RuleSet}_{io_{rp}}$, where $\Sigma^{RuleSet}$ is the rule-set of the current box. The field io_{inp} points to the wire that is examined for input. Before performing matching the HAM code must check availability of a data item on this wire. The field io_{mp} points to a position on the stack representing the current value. It is copied from the wire to the stack before a match can be performed.

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{StartMatches})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{rp} +\!\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{StartMatches}}_{s,\eta})\ \theta}\ (\textsc{stepStartMatches})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{MatchRule})\#\texttt{cs} \Downarrow^n_{n+1} (s^{vals} := [], \eta, io(\!|\ io_{inp} := 0, io_{mp} := 0, io_{rp} +\!\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchRule}}_{s,\eta})\ \theta}\ (\textsc{stepMatchRule})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{MatchedRule})\#\texttt{cs} \Downarrow^n_{n+1} (s^{vals} := [], \eta, io, \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchedRule}}_{s,\eta})\ \theta}\ (\textsc{stepMatchedRule})$$

$$\frac{}{s, \eta, io, \theta \vdash (\texttt{MatchNone})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{inp} +\!\!+, io_{mp} +\!\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchNone}}_{s,\eta})\ \theta}\ (\textsc{stepMatchNone})$$

$$\frac{\exists\ v.\ \theta\ wire^{io_b, io_{inp}} = \mathit{Some}\ v}{s, \eta, io, \theta \vdash (\texttt{MatchAvailable})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{inp} +\!\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchAvailable}}_{s,\eta})\ \theta}\ (\textsc{stepMatchAvailableTrue})$$

$$\frac{\neg\exists\ v.\ \theta\ wire^{io_b, io_{inp}} = Some\ v}{s, \eta, io, \theta \vdash (\texttt{MatchAvailable})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{inp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchAvailable}}_{s,\eta})\ \theta}\ (\textsc{stepMatchAvailableFalse})$$

$$\frac{\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Bool}\ b)}{s, \eta, io, \theta \vdash (\texttt{MatchBool b})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchBool b}}_{s,\eta})\ \theta}\ (\textsc{stepMatchBoolTrue})$$

$$\frac{\neg(\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Bool}\ b))}{s, \eta, io, \theta \vdash (\texttt{MatchBool b})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchBool b}}_{s,\eta})\ \theta}\ (\textsc{stepMatchBoolFalse})$$

$$\frac{\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Char}\ c)}{s, \eta, io, \theta \vdash (\texttt{MatchChar c})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchChar c}}_{s,\eta})\ \theta}\ (\textsc{stepMatchCharTrue})$$

$$\frac{\neg(\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Char}\ c))}{s, \eta, io, \theta \vdash (\texttt{MatchChar c})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchChar c}}_{s,\eta})\ \theta}\ (\textsc{stepMatchCharFalse})$$

$$\frac{\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Str}\ x)}{s, \eta, io, \theta \vdash (\texttt{MatchString x})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchString x}}_{s,\eta})\ \theta}\ (\textsc{stepMatchStringTrue})$$

$$\frac{\neg(\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Str}\ x))}{s, \eta, io, \theta \vdash (\texttt{MatchString x})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchString x}}_{s,\eta})\ \theta}\ (\textsc{stepMatchStringFalse})$$

$$\frac{\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Int}\ i)}{s, \eta, io, \theta \vdash (\texttt{MatchInt i})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchInt i}}_{s,\eta})\ \theta}\ (\textsc{stepMatchIntTrue})$$

$$\frac{\neg(\eta\ s^{args}_{io_{mp}} = Some\ (\mathsf{Int}\ i))}{s, \eta, io, \theta \vdash (\texttt{MatchInt i})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \Sigma^{RuleSet}_{io_{rp}}, \mathcal{R}^{\texttt{MatchInt i}}_{s,\eta})\ \theta}\ (\textsc{stepMatchIntFalse})$$

$$\frac{\exists\ xs.\ \eta\ s^{args}_{io_{mp}} = Some(\mathsf{Tuple}\ m\ xs)}{s, \eta, io, \theta \vdash (\texttt{MatchTuple m})\#\texttt{cs} \Downarrow^n_{n+1} (s, \eta, io(\!|\ io_{mp} +\!+\ |\!), \texttt{cs}, \mathcal{R}^{\texttt{MatchTuple m}}_{s,\eta})\ \theta}\ (\textsc{stepMatchTupleTrue})$$

$$\frac{\neg(\exists\ xs.\ \eta\ s^{args}_{io_{mp}} = \mathit{Some}\ (\mathsf{Tuple}\ m\ xs))}{s,\eta,io,\theta \vdash (\texttt{MatchTuple m})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io(\!|\ io_{mp} +\!\!+\ |\!),\Sigma^{RuleSet}_{io_{rp}},\mathcal{R}^{\texttt{MatchTuple m}}_{s,\eta})\ \theta}\ (\textsc{stepMatchTupleFalse})$$

$$\frac{\exists\ xs.\ \eta\ s^{args}_{io_{mp}} = \mathit{Some}(\mathsf{Constr}\ i\ j\ xs)}{s,\eta,io,\theta \vdash (\texttt{MatchCon i j})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io(\!|\ io_{mp} +\!\!+\ |\!),\texttt{cs},\mathcal{R}^{\texttt{MatchCon i j}}_{s,\eta})\ \theta}\ (\textsc{stepMatchConTrue})$$

$$\frac{\neg(\exists\ xs.\ \eta\ s^{args}_{io_{mp}} = \mathit{Some}\ (\mathsf{Constr}\ i\ j\ xs))}{s,\eta,io,\theta \vdash (\texttt{MatchCon i j})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io(\!|\ io_{mp} +\!\!+\ |\!),\Sigma^{RuleSet}_{io_{rp}},\mathcal{R}^{\texttt{MatchCon i j}}_{s,\eta})\ \theta}\ (\textsc{stepMatchConFalse})$$

$$\frac{\exists\ l.\ \eta\ s^{args}_{io_{mp}} = \mathit{Some}(\mathsf{Exn}\ x\ l)}{s,\eta,io,\theta \vdash (\texttt{MatchExn x})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io(\!|\ io_{mp} +\!\!+\ |\!),\texttt{cs},\mathcal{R}^{\texttt{MatchExn x}}_{s,\eta})\ \theta}\ (\textsc{stepMatchExnTrue})$$

$$\frac{\neg(\exists\ l.\ \eta\ s^{args}_{io_{mp}} = \mathit{Some}\ (\mathsf{Exn}\ x\ l))}{s,\eta,io,\theta \vdash (\texttt{MatchExn x})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io(\!|\ io_{mp} +\!\!+\ |\!),\Sigma^{RuleSet}_{io_{rp}},\mathcal{R}^{\texttt{MatchExn x}}_{s,\eta})\ \theta}\ (\textsc{stepMatchExnFalse})$$

The following rules unpack structured data, i.e. they bring the contents of a tuple, constructor or function closure onto the stack.

$$\frac{\exists\ l\ m\ xs.s_0 = l\ \wedge\ \eta\ l = \mathit{Some}\ (\mathsf{Tuple}\ m\ xs)\ \wedge\ s' = (s - \top) + xs}{s,\eta,io,\theta \vdash (\texttt{Unpack})\#\texttt{cs} \Downarrow^n_{n+1} (s',\eta,io,\texttt{cs},\mathcal{R}^{\texttt{Unpack}}_{s,\eta})\ \theta}\ (\textsc{stepUnpackTuple})$$

$$\frac{\exists\ l\ t\ m\ xs.s_0 = l\ \wedge\ \eta\ l = \mathit{Some}\ (\mathsf{Constr}_t\ m\ xs)\ \wedge\ s' = (s - \top) + xs}{s,\eta,io,\theta \vdash (\texttt{Unpack})\#\texttt{cs} \Downarrow^n_{n+1} (s',\eta,io,\texttt{cs},\mathcal{R}^{\texttt{Unpack}}_{s,\eta})\ \theta}\ (\textsc{stepUnpackCon})$$

$$\frac{\exists\ l\ f\ m\ p\ xs.s_0 = l\ \wedge\ \eta\ l = \mathit{Some}\ (\mathsf{Fun}\ f\ m\ p\ xs)\ \wedge\ s' = (s - \top) + xs}{s,\eta,io,\theta \vdash (\texttt{Unpack})\#\texttt{cs} \Downarrow^n_{n+1} (s',\eta,io,\texttt{cs},\mathcal{R}^{\texttt{Unpack}}_{s,\eta})\ \theta}\ (\textsc{stepUnpackFun})$$

The following operations copy box input into the heap, check whether a box can write to its output wires, and perform the actual write operation. Note that the compiler must ensure that all `Write` operations are guarded by `CheckOutput` operations.

$$\frac{l = \mathit{fresh}\ (\mathit{dom}\ \eta)}{s,\eta,io,\theta \vdash (\texttt{CopyInput m})\#\texttt{cs} \Downarrow^n_{n+1} (s + l,\eta(l \mapsto (\theta\ \mathit{wire}^{io_b,m})),io,\texttt{cs},(\mathcal{R}^{\texttt{CopyInput m}}_{s,\eta}))\ \theta}\ (\textsc{stepCopyInput})$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{Consume m})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io,\texttt{cs},\mathcal{R}^{\texttt{Consume m}}_{s,\eta})\ \theta(wire^{io_b,m} \mapsto None)} \quad \text{(stepConsume)}$$

$$\frac{\exists\ v.\ \theta\ wire^{io_b,m} = Some\ v}{s,\eta,io,\theta \vdash (\texttt{MaybeConsume m})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io,\texttt{cs},\mathcal{R}^{\texttt{MaybeConsume m}}_{s,\eta})\ \theta(wire^{io_b,m} \mapsto None)} \quad \text{(stepMaybeConsumeTrue)}$$

$$\frac{\neg\ \exists\ v.\ \theta\ wire^{io_b,m} = Some\ v}{s,\eta,io,\theta \vdash (\texttt{MaybeConsume m})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io,\texttt{cs},\mathcal{R}^{\texttt{MaybeConsume m}}_{s,\eta})\ \theta} \quad \text{(stepMaybeConsumeFalse)}$$

$$\frac{outputable\ io\ \theta}{s,\eta,io,\theta \vdash (\texttt{CheckOutputs})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io,\texttt{cs},\mathcal{R}^{\texttt{CheckOutputs}}_{s,\eta})\ \theta} \quad \text{(stepCheckOutputsTrue)}$$

$$\frac{\neg\ (outputable\ io\ \theta)}{s,\eta,io,\theta \vdash (\texttt{CheckOutputs})\#\texttt{cs} \Downarrow^n_{n+1} (s,\eta,io(\!|\ blocked := true\ |\!),(\texttt{Schedule})\#(\texttt{CheckOutputs})\#\texttt{cs},\mathcal{R}^{\texttt{CheckOutputs}}_{s,\eta})\ \theta} \quad \text{(stepCheckOutputsFalse)}$$

$$\frac{l = s_0 \quad \eta\ l = Some\ x}{s,\eta,io,\theta \vdash (\texttt{Write m})\#\texttt{cs} \Downarrow^n_{n+1} ((s - \top),\eta,io,\texttt{cs},\mathcal{R}^{\texttt{Write m}}_{s,\eta})\ \theta(wire_{io_b,m} \mapsto Some\ x)} \quad \text{(stepWrite)}$$

The operations `Input` and `Output` interact with the outside world, enabling character input and output. Since we are mainly interested in the internal behaviour of the system, in particular its resource consumption, we do not model the exact values and use a dummy character value $\perp$ instead. Costs are still accurately modelled, provided they are constant for all possible character values.

$$\frac{l = fresh\ (dom\ \eta)}{s,\eta,io,\theta \vdash (\texttt{Input})\#\texttt{cs} \Downarrow^n_{n+1} (s + l,\eta(l \mapsto (\mathsf{Char}\ \perp)),io,\texttt{cs},\mathcal{R}^{\texttt{Input}}_{s,\eta})\ \theta} \quad \text{(stepInput)}$$

$$\frac{l = fresh\ (dom\ \eta)}{s,\eta,io,\theta \vdash (\texttt{Output})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top,\eta,io,\texttt{cs},\mathcal{R}^{\texttt{Output}}_{s,\eta})\ \theta} \quad \text{(stepOutput)}$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{WithinStack h p})\#\texttt{cs} \Downarrow^n_{n+1} (s + io_{splim},\eta,io(\!|\ splim := |\ s\ |\ +p\ |\!),\texttt{cs},\mathcal{R}^{\texttt{WithinStack h p}}_{s,\eta})\ \theta} \quad \text{(stepWithinStackSpace)}$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{WithinHeap h p})\#\texttt{cs} \Downarrow^n_{n+1} (s + io_{hplim},\eta,io(\!|\ hplim := |\ dom\ \eta\ |\ +p\ |\!),\texttt{cs},\mathcal{R}^{\texttt{WithinHeap h p}}_{s,\eta})\ \theta} \quad \text{(stepWithinHeapSpace)}$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{DoneWithinStack h})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \eta, io(\!| \mathit{splim} := s_0 |\!), \texttt{cs}, \mathcal{R}^{\texttt{DoneWithinStack h}}_{s,\eta})\ \theta} \quad (\textsc{stepDoneWithinStack})$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{DoneWithinHeap h})\#\texttt{cs} \Downarrow^n_{n+1} (s - \top, \eta, io(\!| \mathit{hplim} := s_0 |\!), \texttt{cs}, \mathcal{R}^{\texttt{DoneWithinHeap h}}_{s,\eta})\ \theta} \quad (\textsc{stepDoneWithinHeap})$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{Within h t})\#\texttt{cs} \Downarrow^n_{n+1} (s + t, io, \texttt{cs}, \mathcal{R}^{\texttt{Within h t}}_{s,\eta})\ \theta} \quad (\textsc{stepWithin})$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{DoneWithin})\#\texttt{cs} \Downarrow^n_{n+1} (s, \mathit{unsetTimer}\ io, \texttt{cs}, \mathcal{R}^{\texttt{DoneWithin}}_{s,\eta})\ \theta} \quad (\textsc{stepDoneWithin})$$

$$\frac{}{s,\eta,io,\theta \vdash (\texttt{RaiseWithin h t})\#\texttt{cs} \Downarrow^n_{n+1} (s - s_{0..2-1}, \mathit{setTimer}\ io\ s_0\ s_1, \texttt{cs}, \mathcal{R}^{\texttt{RaiseWithin h t}}_{s,\eta})\ \theta} \quad (\textsc{stepRaiseWithin})$$

4.3.5 Expression-level semantics

As a small-step semantics the above rules specify one step in the evaluation, depending on the next element in the instruction list `cs`. The semantics on expression level is the transitive closure over this relation in the following sense.

$$\frac{c \neq \texttt{Schedule} \quad s,\eta,io,\theta \vdash (\texttt{c}\#\texttt{cs}) \Downarrow^m_{m+1} (s'',\eta'',io'',\texttt{cs}'',p'')\ \theta'' \quad s'',\eta'',io'',\theta'' \vdash \texttt{cs}'' \rightsquigarrow_n (s',\eta',io',\texttt{cs}',p')\ \theta'}{s,\eta,io,\theta \vdash (\texttt{c}\#\texttt{cs}) \rightsquigarrow_{m+n+1} (s',\eta',io',\texttt{cs}',p'' + p')\ \theta'} \quad (\textsc{ssem})$$

The continuation cs' is part of the result, because the computation may block in the CheckOutputs rule. The costs of the entire computation is the sum of the costs of the components, using the $+$ operation as defined by the relevant resource algebra. The step counters m, n are only needed for technical reasons (to enable induction over this step counter) and do not represent resource consumption in any way. Note that the rule is restricted to the case where the first instruction is different from `Schedule`. This assures that in the case of a `Schedule` control is yielded to the scheduler, which then picks the next runnable box, removes the initial `Schedule` and continues with evaluating the box.

4.3.6 Exceptions

The above rule for the expression-level semantics is slightly simplified in the sense that it doesn't check for (synchronous) exceptions. Such checks can be added to the rule, provided that the resource algebra used in the semantics models the resource that might become empty. For example, in the case of heap space a pre-condition of the form

$$| \mathit{hplim} - \mathit{hp} | < \mathcal{R}^{\texttt{c}}_{s,\eta}$$

can be added, if $\mathcal{R}$ models heap consumption. If this condition is false, another rule can be added for raising a heap-overflow exception.

4.4 System level

As static information we use for each box a mapping *Wiring* that maps a wire index to the corresponding box name and wire index it is connected to. The tables Σ^{Fun}, Σ^{Lab} and Σ^{Exn} are used to map function names, labels and exceptions to their corresponding instruction sequences. The $\Sigma^{RuleSet}$ is a partial mapping from natural numbers to instruction sequences, containing the code for the rules in the box. Here the domain needs to be a total order to allow reordering. A *Box* is then represented as a rule-set (a mapping from natural numbers to instruction sequences) and such a wiring.

$$\begin{array}{lcl}
\Sigma^{RuleSet} & \equiv & \mathbb{N} \rightsquigarrow_f \mathit{Instr\ list} \\
\Sigma^{Fun} & \equiv & \mathit{Function} \Rightarrow \mathit{Instr\ list} \\
\Sigma^{Lab} & \equiv & \mathit{Label} \Rightarrow \mathit{Instr\ list} \\
\Sigma^{Exn} & \equiv & \mathit{Exception} \Rightarrow \mathit{Instr\ list} \\
\mathit{Wiring} & \equiv & \mathit{Wire} \Rightarrow (\mathit{BName} \times \mathit{Wire}) \\
\mathit{Box} & \equiv & (\Sigma^{RuleSet} \times \Sigma^{Lab} \times \Sigma^{Exn} \times \mathit{Wiring})
\end{array}$$

We model the dynamic state of the entire system as follows. The entire system (*System*) is a triple (bs, β, θ), where bs is a collection of box names still to be processed in the current iteration, β is a (total) mapping of box-names to box states, and θ is a (total) mapping of wire-names to wire states. A *box state* (*BState*) is a quadrupel (s, η, io, cs), where s is the stack , η is the heap, io is the IO record, and cs is the continuation, i.e. the code still to be processed by the box. The *wire state* (*WState*) is simply an optional heap value, i.e. either a proper heap value (*Some* x) or an empty flag (*None*).

$$\begin{array}{lcl}
\mathit{WState} & \equiv & \mathit{HVal\ option} \\
\mathit{BState} & \equiv & \mathit{Stack} \times \mathit{Heap} \times \mathit{IO} \times \mathit{Instr\ list} \\
\mathit{System} & \equiv & \mathit{BName\ set} \times (\mathit{BName} \Rightarrow \mathit{BState}) \times (\mathit{WName} \Rightarrow \mathit{WState})
\end{array}$$

NeededIn maps for each box, each rule in the ruleset to the set of needed inputs, i.e. a set of wires on which input must be available for the rule to fire. *NeededOut* maps for each box, each rule in the ruleset to the set of needed inputs, i.e. a set of wires to which this rule will write. These tables are filled with the compiler directives shown in Figure 12.

$$\begin{array}{lcl}
\mathit{NeededIn} & :: & \mathit{Box} \Rightarrow \mathbb{N} \Rightarrow \mathit{Wire\ set} \\
\mathit{NeededOut} & :: & \mathit{Box} \Rightarrow \mathbb{N} \Rightarrow \mathit{Wire\ set}
\end{array}$$

The global table *RuleTab* maps a box name to a *Box* structure, containing its static information. Here we are only interested in the rule-set component of the box structure and we use the shorthand $\mathit{RuleTab}_1$ for the first projection on the corresponding rule tab entry.

$$\begin{array}{lcl}
\mathit{RuleTab} & :: & \mathit{BName} \Rightarrow \mathit{Box} \\
\mathit{RuleTab}_1 & :: & \mathit{BName} \Rightarrow \mathit{RuleSet}
\end{array}$$

We need the following additional definitions to specify the semantics on system level:

$$
\begin{array}{lcl}
\mathit{outputable} & :: & \mathit{IO} \Rightarrow (\mathit{WName} \Rightarrow \mathit{WState}) \Rightarrow \mathit{bool} \\
\mathit{outputable\ io\ \theta} & \equiv & (\forall w.\ w \in (\mathit{NeededOut}\ (\mathit{RuleTab}\ io_b)\ io_{rp}) \longrightarrow \\
 & & \exists\ v.\theta\ \mathit{wire}_{io_b,m} = \mathit{Some}\ v) \\
 & & \\
\mathit{runnable} & :: & (\mathit{WName} \Rightarrow \mathit{WState}) \Rightarrow \mathit{BState} \Rightarrow \mathit{bool} \\
\mathit{runnable}\ \theta\ (s,\eta,io,cs) & \equiv & \neg(io_{blocked}) \wedge \\
 & & (\exists r.\ r \in \mathit{dom}\ (\mathit{RuleTab}_1\ io_b) \wedge \\
 & & \quad (\forall w.\ w \in (\mathit{NeededIn}\ (\mathit{RuleTab}\ io_b)\ r) \longrightarrow \exists\ v.\theta\ \mathit{wire}^{io_b,w} = \mathit{Some}\ v)) \\
 & & \\
\mathit{restartable} & :: & (\mathit{WName} \Rightarrow \mathit{WState}) \Rightarrow \mathit{BState} \Rightarrow \mathit{bool} \\
\mathit{restartable}\ \theta\ (s,\eta,io,cs) & \equiv & io_{blocked} \wedge \\
 & & (\forall w.\ w \in (\mathit{NeededOut}\ (\mathit{RuleTab}\ io_b)\ io_{rp}) \longrightarrow\ \theta\ \mathit{wire}^{io_b,w} = \mathit{None})
\end{array}
$$

The transition relation on system level has the form $s \rightharpoonup s'$ meaning that in one step, the system state s proceeds to state s'. The transitive closure over the small-step semantics on expression level, written as $\rightsquigarrow_m$, is defined in the previous section. We use $\emptyset$ to denote the empty heap, mapping all locations to *None*, and the empty stack, an empty list of frames.

The rules defining the HAM semantics on system level are as follows. We use S as a short-hand for bs, β, θ, p. We distinguish between three cases: if the box, selected from the scheduling queue, is a runnable box, then the continuation of the box must be empty, and execution continues with the next rule in the box; note that in this case the box starts with an empty heap and stack; if the box, selected from the scheduling queue, is a restartable box, then the continuation must start with a `Schedule`, which is removed from the code and the remaining code is executed on expression level; if the scheduling queue is empty, it is re-filled using the $\mathcal{U}_S$ operation. The costs recorded in the last field of the state, combine the costs of the individual operations on scheduling level referring to constants such as $\mathcal{R}_S^{\in}$ as explained in the following sub-section.

$$
\frac{\begin{array}{c} bn \in_{bs,\beta,\theta,p} bs \quad (s,\eta,io,[]) = \beta\ bn \quad \mathit{runnable}\ \theta\ (s,\eta,io,[]) \\ cs = \mathit{RuleTab}_1\ io_b\ io_{rp} \quad \emptyset,\emptyset,io,\theta \vdash cs \rightsquigarrow_m (s',\eta',io',cs',p')\ \theta' \end{array}}{(bs,\beta,\theta,p) \rightharpoonup ((bs \ominus_{bs,\beta,\theta,p} bn) \oplus_{bs,\beta,\theta,p} bn,\ \beta(bn \mapsto (s',\eta',io',cs')),\ \theta', p + \mathcal{R}_S^{\in} + \mathcal{R}_S^{\ominus} + p' + \mathcal{R}_S^{\oplus})}\ (\textsc{scheduleRunnable})
$$

$$
\frac{\begin{array}{c} bn \in_{bs,\beta,\theta,p} bs \quad (s,\eta,io,cs'') = \beta\ bn \quad \mathit{restartable}\ \theta\ (s,\eta,io,cs'') \\ (\texttt{Schedule}\#cs) = cs'' \quad s,\eta,io,\theta \vdash cs \rightsquigarrow_m (s',\eta',io',cs',p')\ \theta' \end{array}}{(bs,\beta,\theta,p) \rightharpoonup ((bs \ominus_{bs,\beta,\theta,p} bn) \oplus_{bs,\beta,\theta,p} bn, \beta(bn \mapsto (s',\eta',io',cs')),\ \theta', p + \mathcal{R}_S^{\in} + \mathcal{R}_S^{\ominus} + p' + \mathcal{R}_S^{\oplus})}\ (\textsc{scheduleRestartable})
$$

$$
\frac{\neg\ \exists\ bn \in_{bs,\beta,\theta,p} bs}{(bs,\beta,\theta,p) \rightharpoonup (\mathcal{U}_{bs,\beta,\theta,p}, \mathcal{C}_B\beta\ \theta, \mathcal{C}_W\beta\ \theta, p + \mathcal{R}_S^{\mathcal{U}} + \mathcal{R}_S^{\mathcal{C}_B} + \mathcal{R}_S^{\mathcal{C}_W})}\ (\textsc{scheduleNIL})
$$

Note that this specification of the system level is parameterised by the concrete membership relation $\in_S$ and by the operations for picking the next box to be executed $\ominus_S$ and putting it back into the system $\oplus_S$. In the case where there is no next box available for scheduling, we assume an operation for generating a new collection of schedulable boxes, $\mathcal{U}_S$, an operation for modifying the box mapping, $\mathcal{C}_B$, and an operation for modifying the wire mapping, $\mathcal{C}_W$).

We expect implementations of Hume to use well-known scheduling policies such as round-robin scheduling based on some total ordering of the boxes derived from the source code (this is what's

used in the reference implementation of the HAM). However, we leave the concrete realisation of the scheduling operations ($\in_S, \ominus_S, \oplus_S$) and of the synchronisation operations ($\mathcal{U}_S, \mathcal{C}_B, \mathcal{C}_W$) to the implementor. Naturally, the overall costs of the system will be depend on the scheduling policy. Thus, all these parametric operations have to come with corresponding cost functions, describing how the costs coming out of the expression level shall be combined.

The following operations implement round-robin scheduling. We initially sort the boxes by a global ordering derived from the source code. The membership relation only examines the head of the list. Picking the next element means taking the head of the list. Putting a box back means adding it to the end of the list. Since the length of the scheduling queue never decreases no global synchronisation is needed, and the definition of $\mathcal{U}_S, \mathcal{C}_B, \mathcal{C}_W$ is arbitrary.

$$
\begin{array}{lcl}
\mathcal{D} & \equiv & \mathit{BName\ list} \\
b \in_S bs & \equiv & b = hd\ bs \\
\ominus_S & \equiv & \lambda bs\ b.tl\ bs \\
\oplus_S & \equiv & \lambda bs\ b.bs@[b] \\
\mathcal{U}_S & \equiv & [] \\
\mathcal{C}_B\ \beta\ \theta & \equiv & \beta \\
\mathcal{C}_W\ \beta\ \theta & \equiv & \theta
\end{array}
$$

4.4.1 Cost modelling on system level

The cost model on the system level is realised similarly to the expression level. We use a resource algebra $(R, +, \leq)$, where R must be the same data structure as on the expression level, but $+$ and $\leq$ may be different operations. For example, in a parallel implementation of the HAM the overall time consumption is not necessarily the sum over all the time consumptions of all boxes plus system time. The constants in the system-level resource algebra must correspond to the basic scheduling operations: $\mathcal{R}_S^{\ominus}$ for getting a box name from the scheduling queue, $\mathcal{R}_S^{\oplus}$ for putting a box back into the scheduling queue, $\mathcal{R}_S^{\in}$ for testing whether a box name is in the scheduling queue; $\mathcal{R}_S^{\mathcal{U}}$ for re-filling the scheduling queue; $\mathcal{R}_S^{\mathcal{C}_B}$ for synchronising all boxes, and $\mathcal{R}_S^{\mathcal{C}_W}$ for synchronising all wires. Note, that in general all these costs depend on the entire system state. In the case of modelling time consumption in a round-robin scheduler, the last three constants would be 0; $\mathcal{R}_S^{\in}$, $\mathcal{R}_S^{\ominus}$, $\mathcal{R}_S^{\ominus}$ would be the costs for testing for list membership, extracting an element from the list and adding an element to the end of the list.

5 Summary

As specification of the HAM behaviour we presented a 2 level, small-step operational semantics as well as a reference implementation in pseudo C. The former will act as the basis for reasoning about resource consumption. The latter is more concrete and thus more clearly exhibits the costs involved in executing HAM instructions. These costs are captured in the operational semantics via operations of a resource algebra applied to counters that are part of the system state. Since the rules of the semantics only refer to these abstract operations of the resource algebra, they can be chosen independently from the rules of the semantics, and we have shown instances for heap space, stack space and time consumption, thus giving cost models of the HAM for these resources. On top of the small-step semantics we have defined a system-level semantics, that deals with the scheduling of multiple boxes in Hume. Here we parameterise the semantics with concrete scheduling operations, accompanied again by a system-level resource algebra for composing costs on system level. Thus, we arrive at a simple, yet flexible, formalisation of HAM behaviour and resource consumption on system level.

References

[ABM06] D. Aspinall, L. Beringer, and A. Momigliano. Optimisation Validation. In *COCV06 — Fifth Workshop on Compiler Optimization Meets Compiler Verification*, Vienna, Austria, April 2, 2006. To appear in ENTCS.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, 1987.

[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.

[Ham03] K. Hammond. An Abstract Machine for Resource Bounded Computations in Hume. In *IFL03 — Intl Workshop on Implementation of Functional Languages*, pages 79–94, Edinburgh, Scotland, September 2003. Draft proceedings.

[HM] K. Hammond and G. Michaelson. The Hume Report. Available from: `http://www-fp.dcs.st-and.ac.uk/hume/report/hume-report.ps`. Version 0.3.

[HM02] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *14th Intl Workshop on Implementation of Functional Languages*, pages 386–403, Madrid, Spain, September 2002.

[JLH07] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14.

[Lan64] P. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition edition, April 1999.

[Mac] D. MacQueen. Standard ML of New Jersey. Web page. `http://www.smlnj.org/`.

[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.

[PL92] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages: a Tutorial.* Prentice-Hall, 1992.