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

Specific Targeted Research Project (STReP)
FET Open

D03 (WP7b): Hume-HAM Translation

Due date of deliverable: 1st November 2005
Actual submission date: 3rd March 2006

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

Lead contractor: Heriot-Watt University Revision: 1.8

Purpose: In order to obtain verifiable cost models for Hume source programs, we must develop formal models relating Hume source language constructs to the HAM abstract machine. This will allow us to verify the semantic correctness of the compiler. We will define a formal translation from Hume to HAM as a set of source to abstract machine instruction compilation rules, and verify these rules against the actual translations produced by the HAM compiler.

Results: We present the compilation rules of higher-order Hume, including exceptions, down to code on the Hume Abstract Machine (HAM).

Conclusion: These compilation rules have been verified against the `phamc` prototype compiler and thus accurately reflect the source code transformation during the compilation. The formal description of the compilation rules is the basis for follow-on work which will aim at proving cost-correspondence between the formal semantics of Hume and of HAM.

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

Hume-HAM Translation

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

Abstract

This document gives a formal description of the compilation of Hume programs to Hume Abstract Machine (HAM) instructions. Input to the translation is Hume with higher-order functions and exceptions, but without timeouts (these are still in the design phase and subject to changes). Output are sequences of HAM instructions plus special directives to the HAM for structuring the execution. The translation described here exactly reflects the code-generation part of the Hume-to-HAM compiler (`phamc`) and is therefore a suitable basis for mapping computation costs of Hume to HAM instructions.

1 Introduction

One essential feature of Hume is predictability of computation costs for arbitrary Hume expressions [HM02]. This is a prerequisite for developing static analyses for heap, stack and time consumption of Hume programs, which will be one major task in the project. As a basis for such an analysis this document formally describes the translation process from high-level Hume code down to instructions on the Hume Abstract Machine (HAM). This translation models those optimisations performed by the `phamc` prototype Hume compiler, which intentionally refrains from more aggressive optimisations in order to maintain predictability.

This document, together with the documents on a formal semantics for Hume (D12) and on the specification and cost model of the HAM (D4) is the basis for reasoning about functional behaviour as well as resource consumption. We refer to D4 and D12 for the specifications of source and target language of the compilation. At the current stage we focus on these documents as specifications and interfaces between the different groups. As immediate next steps we plan to use these formalisations to prove properties on the resource consumption, and in particular prove cost-correspondence between the models for Hume and the HAM.

In the rest of the document we first present the abstract syntax of Hume. We then give a series of translation rules for the components of the Hume language, producing sequences of HAM instructions as output. We present examples of compiling Hume to HAM, taken from the `phamc` compiler. Finally, we summarise.

2 Hume

Hume [HM] is a functionally-based research language aimed at applications requiring bounded time and space behaviour, such as real-time embedded systems. The challenge to be met by the Hume design is

to preserve the essential properties of costability and low-level interfacing that are required by real-time embedded systems whilst providing as high-level a programming environment as possible.

Figure 1 shows the Hume abstract syntax. The language uses a rule-based approach, with a purely functional expression notation embedded in an asynchronous process model. This simplifies both correctness proofs and the construction of cost models at the expression level. Process abstractions ("boxes") specify an asynchronous and stateless mapping of inputs to outputs, which are scheduled whenever required inputs become available. Boxes can be seen as stateless objects with a rigid communication structure, which both assists process/communication costing and simplifies the construction of deadlock/termination proofs. They are wired explicitly into a static process network (again simplifying both correctness and costing) using a single-buffer approach. Single-buffering allows tight controls over buffer sizes based on the types of values that are communicated, gives a simple and easily implemented semantics, and can be extended to multiple buffering stages by simply introducing additional intermediary boxes. Boxes are activated whenever required inputs become available, write the outputs they produce to the corresponding buffers, and then suspend. In this way, we achieve time- or event-triggered process activation, as well as repetition at the box level.

The expression layer of Hume provides several levels of increasing expressive power to program the behaviour of a box. In this document we use higher-order Hume, including exceptions. Therefore, functions can be passed to other functions, be stored in variables and partial application is permitted. Exceptions are permitted to transfer control to the exception handler attached to the current box. However, at present we do not model timeouts, which are still subject to minor changes in the design.

For example, we can define a Hume program to continuously poll standard input, recording the input status:

```
-- check for input within given time

stream input from "std_in";
stream output to "std_out";

data STATUS = WAITING | RECEIVED;

box getinput
in  (i::char)
out (o::char,handshake::STATUS)
match
  v -> (v,RECEIVED)
| * -> (*,WAITING);

wire getinput (input) (output,timer.monitor);
```

3 Compilation Rules

Figures 2–7 give rules for compiling Hume abstract syntax forms into the Hume Abstract Machine (HAM) instructions, as a formal compilation scheme similar to that for the G-machine [Aug87]. These rules have been used to construct a compiler (`phamc`) from Hume source code to the HAM.

The compilation scheme uses a simple sequence notation: $\langle\ i_1,\ \ldots,\ i_n\ \rangle$ denotes a sequence of n items. The @ operation concatenates two such sequences. Many rules also use an environment ρ which maps identifiers to $\langle\ \mathit{depth},\ \mathit{offset}\ \rangle$ pairs.

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

program ::=	$decl_1$; ... ; $decl_n$	$n \geq 1$
decl ::=	*box* | id = *expr* | id ⟨ $match_1$ | ⋯ | $match_n$ ⟩	$n \geq 1$
box ::=	`box` id *ins* *outs* `fair/unfair` *bmatches* [`box` *cmatches*]	
ins/outs ::=	⟨ $\mathrm{id}_1, \ldots, \mathrm{id}_n$ ⟩	$n \geq 0$
bmatches ::=	*expr* | ⟨ $bmatch_1$ | ⋯ | $bmatch_n$ ⟩	$n \geq 1$
cmatches ::=	*exnexpr* | ⟨ $cmatch_1$ | ⋯ | $cmatch_n$ ⟩	$n \geq 1$
bmatch ::=	⟨ $bpat_1$, ... , $bpat_n$ ⟩ → *expr*	$n \geq 1$
cmatch ::=	*cpat* → *exnexpr*	
match ::=	⟨ pat_1 , ... , pat_n ⟩ → *expr*	$n \geq 1$
exnmatch ::=	⟨ pat_1 , ... , pat_n ⟩ → *exnexpr*	$n \geq 1$
expr ::=	int | float | char | bool | string | `*`	
	| var $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| id $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| con $expr_1$ ⋯ $expr_n$	$n \geq 0$
	| ⟨ $expr_1$, ... , $expr_n$ ⟩	$n \geq 2$
	| `if` $expr_1$ `then` $expr_2$ `else` $expr_3$	
	| `case` *expr* `of` ⟨ $match_1$ | ⋯ | $match_n$ ⟩	$n \geq 1$
	| `let` ⟨ $vdecl_1$, ... , $vdecl_n$ ⟩ `in` *expr*	$n \geq 1$
	| *expr* within int `time` *raiseexpr*	
	| *expr* within int `stack` *raiseexpr*	
	| *expr* within int `heap` *raiseexpr*	
	| *raiseexpr*	
raiseexpr ::=	`raise` exn *exnexpr*	
exnexpr ::=	int | float | char | bool | string | `*` | var	
	| con $exnexpr_1$ ⋯ $exnexpr_n$	$n \geq 0$
	| ⟨ $exnexpr_1$, ... , $exnexpr_n$ ⟩	$n \geq 2$
	| `if` $exnexpr_1$ `then` $exnexpr_2$ `else` $exnexpr_3$	
	| `case` *exnexpr* `of` ⟨ $exnmatch_1$ | ⋯ | $exnmatch_n$ ⟩	$n \geq 1$
	| `let` ⟨ $exnvdecl_1$, ... , $exnvdecl_n$ ⟩ `in` *exnexpr*	$n \geq 1$
vdecl ::=	var = *expr*	
exnvdecl ::=	var = *exnexpr*	
bpat ::=	*pat* | `*` | _`*`	
cpat ::=	exn *pat*	
pat ::=	int | float | char | bool | string | _ | var	
	| con pat_1 ⋯ pat_n	$n \geq 0$
	| ⟨ pat_1 , ... , pat_n ⟩	$n \geq 2$

Figure 1: Hume abstract syntax

expr

$$
\begin{array}{lcl}
\mathcal{C}_E\ \rho\ (b) & = & \langle\ \text{MkBool}\ b\ \rangle \\
\mathcal{C}_E\ \rho\ (c) & = & \langle\ \text{MkChar}\ c\ \rangle \\
\mathcal{C}_E\ \rho\ (s) & = & \langle\ \text{MkString}\ s\ \rangle \\
\mathcal{C}_E\ \rho\ (i) & = & \langle\ \text{MkInt}\ i\ \rangle \\
\mathcal{C}_E\ \rho\ (f) & = & \langle\ \text{MkFloat}\ f\ \rangle \\
\mathcal{C}_E\ \rho\ (*) & = & \langle\ \text{MkNone}\ \rangle \\
\mathcal{C}_E\ \rho\ (\ e_1,\ \ldots,\ e_n\ ) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \langle\ \text{MkTuple}\ n\ \rangle \\
\mathcal{C}_E\ \rho\ \text{con}\ e_1,\ \ldots,\ e_n & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \langle\ \text{MkCon con}\ n\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (v) & = & \textbf{let}\ \langle\ d, m\ \rangle\ =\ \rho\ v\ \textbf{in} \\
 & & \textbf{if}\ d\ =\ 0\ \textbf{then}\ \langle\ \text{PushVar}\ m\ \rangle \\
 & & \textbf{else}\ \langle\ \text{PushVarF}\ d\ m\ \rangle\ \textbf{endif} \\
\\
\mathcal{C}_E\ \rho\ (p\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \langle\ \text{CallPrim}n\ p\ \rangle \\
\mathcal{C}_E\ \rho\ (f\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @ \\
 & & \textbf{let}\ m =\ \mathit{arity}\ f\ \textbf{in} \\
 & & \textbf{if}\ n\ =\ m\ \textbf{then}\ \langle\ \text{Call}\ f, \text{Slide}\ n\ \rangle \\
 & & \textbf{else if}\ n\ <m\ \textbf{then}\ \langle\ \text{MkFun}\ f\ m\ n\ \rangle \\
 & & \textbf{else}\ \langle\ \text{Call}\ f, \text{Slide}\ n\ \rangle\ \underbrace{@\ \langle\ \text{Ap 1},\ \text{Slide 2}\ \rangle}_{m-n}\ \textbf{endif} \\
\\
\mathcal{C}_E\ \rho\ (v\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n\ @\ \ldots\ @\ \mathcal{C}_E\ \rho\ e_1\ @\ \mathcal{C}_E\ \rho\ v\ @ \\
 & & \textbf{let}\ \langle\ d, m\ \rangle\ =\ \rho\ v\ \textbf{in} \\
 & & \textbf{if}\ d\ =\ 0\ \textbf{then}\ \langle\text{CallVar}\ m\ n,\ \text{SlideVar}\ m\ \rangle \\
 & & \textbf{else}\ \langle\text{CallVarF}\ d\ m\ n,\ \text{SlideVarF}\ d\ m\ \rangle\ \textbf{endif} \\
\\
\mathcal{C}_E\ \rho\ (\textbf{if}\ c\ \textbf{then}\ t\ \textbf{else}\ f) & = & \mathcal{C}_E\ \rho\ c\ @\ \langle\ \text{If}\ lt\ \rangle\ @\ \mathcal{C}_E\ \rho\ f\ @ \\
 & & \langle\ \text{Goto}\ ln,\ \text{Label}\ lt\ \rangle\ @\ \mathcal{C}_E\ \rho\ t\ @ \\
 & & \langle\ \text{Label}\ ln\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (\textbf{case}\ e\ \textbf{of}\ ms) & = & \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{Call}\ lc, \text{Slide 1}, \text{Goto}\ ln, \text{Label}\ lc\ \rangle\ @ \\
 & & \mathcal{C}_{Case}\ \rho\ ms\ @ \\
 & & \langle\ \text{Label}\ ln, \text{Function}\ lc\ (\mathit{labels}\ lc)\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (\textbf{let}\ d_1\ \ldots\ d_n\ \textbf{in}\ e) & = & \textbf{let}\ \rho'\ =\ \mathit{bindDefs}\ \langle\ d_1,\ \ldots,\ d_n\ \rangle\ \rho\ \textbf{in} \\
 & & \langle\ \text{Call}\ ll, \text{Goto}\ ln, \text{Label}\ ll, \text{CreateFrame}\ n\ \rangle\ @ \\
 & & \mathcal{C}_{Let}\ \rho\ 0\ d_1\ @\ \ldots\ @\ \mathcal{C}_{Let}\ \rho\ (n-1)\ d_n\ \ @ \\
 & & \mathcal{C}_E\ \rho'\ e\ @\ \langle\ \text{Return}, \text{Label}\ ln\ \rangle\ @ \\
 & & \langle\ \text{Function}\ ll\ \langle\rangle\ \rangle \\
\\
\mathcal{C}_E\ \rho\ (\textbf{raise}\ x\ e) & = & \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{Raise}\ x\rangle \\
\\
\mathcal{C}_E\ \rho\ (e\ \textbf{within}\ t\ \texttt{time}\ \ \textbf{raise}\ x) & = & \langle\ \text{Within}\ lx\ t\ \rangle\ @\ \mathcal{C}_E\ \rho\ e\ @ \\
 & & \langle\ \text{Goto}\ ln, \text{Label}\ lx, \text{RaiseWithin},\ \text{MkTuple 0},\ \text{Raise Timeout}\ x,\ \text{Label}\ ln,\ \text{Do} \\
\mathcal{C}_E\ \rho\ (e\ \textbf{within}\ p\ \texttt{stack}\ \ \textbf{raise}\ x) & = & \langle\ \text{WithinStackSpace}\ x\ p\ \rangle\ @\ \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{DoneWithinStackSpace}\ x\ \rangle \\
\mathcal{C}_E\ \rho\ (e\ \textbf{within}\ p\ \texttt{heap}\ \ \textbf{raise}\ x) & = & \langle\ \text{WithinHeapSpace}\ x\ p\ \rangle\ @\ \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{DoneWithinHeapSpace}\ x\ \rangle \\
\\
\mathcal{C}_{Case}\ \rho\ \langle\ r_1, \ldots, r_m\ \rangle & = & \textbf{let}\ n\ =\ \mathit{maxVars}\ \langle\ r_1,\ \ldots,\ r_m\ \rangle\ \textbf{in} \\
 & & \langle\ \text{CreateFrame}\ n\ \rangle\ @ \\
 & & \mathcal{C}_F\ \rho\ \langle\ r_1, \ldots, r_m\ \rangle \\
\\
\mathcal{C}_{Let}\ \rho\ n\ (\ id = e\ ) & = & \mathcal{C}_E\ \rho\ e\ @\ \langle\ \text{MakeVar}\ n\ \rangle
\end{array}
$$

Figure 2: Compilation Rules for Expressions

decl

$$\mathcal{C}_D\ \rho\ (\textbf{box}\ v\ ins\ outs\ \textbf{fair}\ rs\ \textbf{handle}\ xs) \;=\; \mathcal{C}_B\ \rho\ true\ b\ ins\ outs\ rs\ xs$$
$$\mathcal{C}_D\ \rho\ (\textbf{box}\ b\ ins\ outs\ \textbf{unfair}\ rs\ \textbf{handle}\ xs) \;=\; \mathcal{C}_B\ \rho\ false\ b\ ins\ outs\ rs\ xs$$

$$\begin{array}{l}\mathcal{C}_D\ \rho\ (v\ =\ \langle\ p_1\ \rightarrow\ e_1\ \ldots\ p_n\ \rightarrow\ e_n\ \rangle\ ) = \\ \quad \textbf{let}\ n\ =\ maxVars\ \langle\ p_1,\ \ldots,\ p_n\ \rangle\ \textbf{in} \\ \quad \langle\ \text{Label}\ f, \text{CreateFrame}\ n\ \rangle\ @ \\ \quad \mathcal{C}_F\ \rho\ \langle\ \langle\ p_1\ \rangle\ \rightarrow\ e_1,\ \ldots,\ \langle\ p_n\ \rangle\ \rightarrow\ e_n\ \rangle\ @ \\ \quad \langle\ \text{Function}\ f\ (labels\ f)\ \rangle\end{array}$$

Figure 3: Compilation Rules for Declarations

definitions taken from a declaration sequence — the *depth* of these new bindings is 0, whilst the depth of existing variable bindings in the environment is incremented by 1; *bindVars* does the same for a sequence of patterns; and *labels* generates new labels for a set of function/box/exception rules. Note that where labels lt, ln, lx etc. are used, these are assumed to be unique in the obvious way: there is at most one **Label** pseudo-instruction for each label in the translated program. Labels for boxes, functions and exception blocks are derived in a standard way from the (unique) name of the box or function.

The rules are structured by abstract syntax class. The rules for translating expressions ($\mathcal{C}_E$ etc. — Figure 2) are generally straightforward, but note that function frames are created to deal with *let*-, *case*- and *raise*-expressions, which then exploit the function calling mechanism. In the first two cases, this allows the creation of local stack frames. For *case*-expressions and *raise*-expressions, it allows the exploitation of the standard pattern matching instructions. It would obviously be possible to eliminate the function call for *let*-expressions provided the stack frame was properly set up in order to allow access to non-local definitions. For exceptions, since the transfer of control is permanent, it would be possible to replace the entire stack by the exception value and to use a **Goto** rather than a **Call**. In this case, each translated exception rule would finish with a **Schedule** rather than a **Return**. This has been avoided here purely for reasons of complexity.

The rules for translating box and function declarations are shown in Figure 4. These rules create new stack frames for the evaluation of the box or function, label the entry points and introduce appropriate pseudo-instructions. In the case of box declarations, it is also necessary to copy inputs to the stack using **CopyInput** instructions and to deal with fair matching and the exception handlers.

Box bodies are compiled using $\mathcal{C}_R/\mathcal{C}_{R'}$ (Figure 5). These rules compile matches for the outer level patterns using $\mathcal{C}_P$, then compile inner pattern matches using $\mathcal{C}_A$, before introducing **Consume** instructions for non-***** input positions, including _*****. The right hand side can now be compiled. If more than one result is to be produced, the tuple of outputs is unpacked onto the stack. A **CheckOutputs** is inserted to verify that the outputs can be written using appropriate **Write** instructions. Finally, a **Reorder** is inserted if needed to deal with fair matching, and a **Schedule** returns control to the scheduler. The compilation of function/handler bodies using $\mathcal{C}_F/\mathcal{C}_{F'}$ is similar, except that $\mathcal{C}_{P'}$ is used rather than $\mathcal{C}_P$, there is no need to deal with box inputs/outputs or fair matching, and a **Return** rather than **Schedule** is inserted at the end of each compiled rule.

box

$$
\begin{array}{l}
\mathcal{C}_B\ \rho\ f\ b\ (in_1, \ldots, in_i)\ (out_1, \ldots, out_m)\ rs\ xs\ \ = \\
\quad \textbf{let}\ n\ =\ maxVars\ \langle\ p_1,\ \ldots,\ p_n\ \rangle\ \textbf{in} \\
\quad \langle\ \text{Label}\ b\ \rangle\ @ \\
\quad \langle\ \text{CopyInput}\ (i-1),\ \ldots,\ \text{CopyInput}\ 0\ \rangle\ @ \\
\quad \langle\ \text{Push}\ 2, \text{CreateFrame}\ n\ \rangle\ @ \\
\quad (if\ f\ then\ \langle\ \text{StartMatches}\ \rangle\ else\ \langle\ \rangle)\ @\ \mathcal{C}_R\ \rho\ f\ m\ rs\ @ \\
\quad \mathcal{C}_H\ \rho\ xs\ @ \\
\quad \langle\ \text{Box}\ b\ \ldots\ \rangle
\end{array}
$$

exnmatches

$$
\begin{array}{l}
\mathcal{C}_X\ \rho\ \langle\ \langle\ x_1\ p_1\ \rangle\ \rightarrow\ e_1,\ \ldots,\ \langle\ x_n\ p_n\ \rangle\ \rightarrow\ e_n\ \rangle\ = \\
\quad \textbf{let}\ n\ =\ maxVars\ \langle\ p_1,\ \ldots,\ p_n\ \rangle\ \textbf{in} \\
\quad \langle\ \text{Label}\ lx, \text{CreateFrame}\ n\ \rangle\ @ \\
\quad \mathcal{C}_F\ \rho\ \langle\ \langle\ x_1\ p_1\ \rangle\ \rightarrow\ e_1\ \ldots\ \langle\ x_n\ p_n\ \rangle\ \rightarrow\ e_n\ \rangle\ @ \\
\quad \langle\ \text{Function}\ lx\ (labels\ lx)\ \rangle
\end{array}
$$

Figure 4: Compilation Rules for Declarations, Box Bodies and Exception Handlers

bmatches

$$
\begin{array}{lcl}
\mathcal{C}_R\ \rho\ f\ m\ \langle\ r_1,\ \ldots,\ r_n\ \rangle & = & \mathcal{C}_{R'}\ \rho\ f\ m\ r_1\ @\ \ldots\ @\ \mathcal{C}_{R'}\ \rho\ f\ m\ r_n \\
\\
\mathcal{C}_{R'}\ \rho\ f\ m\ (\ \langle\ p_1,\ \ldots,\ p_n \rangle\ \rightarrow\ e\ ) & = & \langle\ \text{Label}\ lr, \text{MatchRule}\ \rangle\ @ \\
 & & \mathcal{C}_P\ p_1\ @\ \ldots\ @\ \mathcal{C}_P\ p_n\ @ \\
 & & \mathcal{C}_A\ p_1\ @\ \ldots\ @\ \mathcal{C}_A\ p_n\ @ \\
 & & \mathcal{C}_C\ 0\ p_1\ @\ \ldots\ @\ \mathcal{C}_C\ (n-1)\ p_n\ @ \\
 & & \mathcal{C}_E\ \rho\ e\ @ \\
 & & (if\ m\ >\ 1\ then\ \langle\ \text{Unpack}\ \rangle\ else\ \langle\ \rangle)\ @ \\
 & & \langle\ \text{CheckOutputs}\ \rangle\ @ \\
 & & \langle\ \text{Write}\ (n-1)\ \ldots\ \text{Write}\ 0\ \rangle\ @ \\
 & & (if\ f\ then\ \langle\ \text{Reorder}\ \rangle\ else\ \langle\ \rangle)\ @ \\
 & & \langle\ \text{Schedule}\ \rangle
\end{array}
$$

matches

$$
\begin{array}{lcl}
\mathcal{C}_F\ \rho\ \langle\ r_1,\ \ldots,\ r_n\ \rangle & = & \mathcal{C}_{F'}\ \rho\ r_1\ @\ \ldots\ @\ \mathcal{C}_{F'}\ \rho\ r_n \\
\mathcal{C}_{F'}\ \rho\ (\ \langle\ p_1,\ \ldots,\ p_n\ \rangle\ \rightarrow\ e\ ) & = & \textbf{let}\ \rho'\ =\ bindVars\ \langle\ p_1\ ,\ \ldots,\ p_n\ \rangle \rho\ \textbf{in} \\
 & & \langle\ \text{Label}\ lf, \text{MatchRule}\ \rangle\ @ \\
 & & \mathcal{C}_{P'}\ p_1\ @\ \ldots\ @\ \mathcal{C}_{P'}\ p_n\ @ \\
 & & \mathcal{C}_A\ p_1\ @\ \ldots\ @\ \mathcal{C}_A\ p_n\ @ \\
 & & \mathcal{C}_E\ \rho'\ e\ @ \\
 & & \langle\ \text{Return}\ \rangle \\
\\
\mathcal{C}_C\ n\ (\textbf{*}) & = & \langle\ \rangle \\
\mathcal{C}_C\ n\ (\_\textbf{*}) & = & \langle\ \text{MaybeConsume}\ n\ \rangle \\
\mathcal{C}_C\ n\ (p) & = & \langle\ \text{Consume}\ n\ \rangle
\end{array}
$$

Figure 5: Compilation Rules for Rule Matches, Functions and Exception Handlers

bpat

$$
\begin{array}{lcl}
\mathcal{C}_P\ (\mathbf{*}) & = & \langle\ \text{MatchNone}\ \rangle \\
\mathcal{C}_P\ (\_\mathbf{*}) & = & \langle\ \text{MatchNone}\ \rangle \\
\mathcal{C}_P\ (p) & = & \langle\ \text{MatchAvailable}\ \rangle\ @\ \mathcal{C}_{P'}\ p
\end{array}
$$

pat

$$
\begin{array}{lcl}
\mathcal{C}_{P'}\ (b) & = & \langle\ \text{MatchBool}\ b\ \rangle \\
\mathcal{C}_{P'}\ (i) & = & \langle\ \text{MatchInt}\ i\ \rangle \\
\mathcal{C}_{P'}\ (f) & = & \langle\ \text{MatchFloat}\ f\ \rangle \\
\mathcal{C}_{P'}\ (c) & = & \langle\ \text{MatchChar}\ c\ \rangle \\
\mathcal{C}_{P'}\ (s) & = & \langle\ \text{MatchString}\ s\ \rangle \\
\mathcal{C}_{P'}\ (c\ p_1\ \ldots\ p_n) & = & \langle\ \text{MatchCon}\ c\ n\ \rangle \\
\mathcal{C}_{P'}\ (x\ p) & = & \langle\ \text{MatchExn}\ x\ \rangle \\
\mathcal{C}_{P'}\ (\ p_1,\ \ldots,\ p_n\ ) & = & \langle\ \text{MatchTuple}\ n\ \rangle \\
\mathcal{C}_{P'}\ (v) & = & \langle\ \text{MatchVar}\ v\ \rangle \\
\mathcal{C}_{P'}\ \_ & = & \langle\ \text{MatchAny}\ \rangle
\end{array}
$$

Figure 6: Compilation Rules for Patterns

Argument passing

$$
\begin{array}{lcl}
\mathcal{C}_A\ (c\ p_1\ \ldots\ p_n) & = & \mathcal{C}_{A'}\ \langle\ p_1, \ldots,\ p_n\ \rangle \\
\mathcal{C}_A\ (\ p_1,\ \ldots,\ p_n\ ) & = & \mathcal{C}_{A'}\ \langle\ p_1, \ldots,\ p_n\ \rangle \\
\mathcal{C}_A\ (x\ p) & = & \mathcal{C}_{A'}\ \langle\ p\ \rangle \\
\mathcal{C}_A\ p & = & \langle\ \rangle \\
 & & \\
\mathcal{C}_{A'}\ m\ \langle\ p_1, \ldots,\ p_n\ \rangle & = & \langle\ \text{CopyArg}\ m\ , \text{Unpack}\ \rangle\ @ \\
 & & \mathcal{C}_{P'}\ p_1\ @\ \ldots\ @\ \mathcal{C}_{P'}\ p_n\ @ \\
 & & \mathcal{C}_N\ 0\ p_1\ @\ \ldots\ @\ \mathcal{C}_N\ n-1\ p_n\ @\ \langle\ \text{Pop}\ n\ \rangle
\end{array}
$$

Nested Patterns

$$
\begin{array}{lcl}
\mathcal{C}_N\ m\ \langle\ p_1, \ldots,\ p_n\ \rangle & = & \langle\ \text{Copy}\ m\ , \text{Unpack}\ \rangle\ @ \\
 & & \mathcal{C}_{P'}\ p_1\ @\ \ldots\ @\ \mathcal{C}_{P'}\ p_n\ @ \\
 & & \mathcal{C}_N\ 0\ p_1\ @\ \ldots\ @\ \mathcal{C}_N\ n-1\ p_n\ @\ \langle\ \text{Pop}\ n\ \rangle
\end{array}
$$

Figure 7: Compilation Rules for Argument Passing

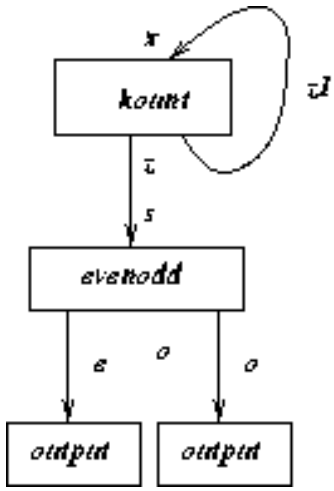

Figure 8: Even/odd example

Finally patterns are compiled using $\mathcal{C}_P/\mathcal{C}_{P'}$, where $\mathcal{C}_P$ inserts the **MatchNone**/ **MatchAvailable** instructions that are needed at the box level, and $\mathcal{C}_{P'}$ compiles simple patterns. Constructed values are matched in two stages: firstly the outer part (the constructor, tuple or exception) is matched, and then if the match is successful, the matched object is deconstructed on the stack to allow its inner components to be matched against the inner patterns. These nested patterns are compiled using $\mathcal{C}_A$ and $\mathcal{C}_N$. $\mathcal{C}_A$ inserts **CopyArg** and **Unpack** instructions to decompose function/box arguments, where $\mathcal{C}_N$ deals with the general nested case using **Copy** instructions to replicate items that are in the local stack frame.

4 Example code

This section presents example Hume programs and the HAM code generated by the `phamc` compiler.

4.1 Even/Odd

The following simple example program implements a counter, which feeds into an `evenodd` box, that outputs even numbers to the left and odd numbers to the right output wire (see Figure 8).

4.1.1 Hume code

```
program

stream outputE to "std_out";
stream outputO to "std_out";

type integer = int 32;

even s = s mod 2 == 0;

box kount
in (x::integer)
out (z::integer, z1::integer)
match
```

```
 (x) -> (x,x+1);

box evenodd
in  (s :: integer)
out (e :: integer, o :: integer)
match
 (x) -> if (even x)
          then (x,*)
          else (*,x) ;

box outE
in  (x :: integer)
out (z :: string)
match
 (x) -> ("EVEN " ++ x as string ++ "\n") ;

box outO
in  (x :: integer)
out (z :: string)
match
 (x) -> ("ODD " ++ x as string ++ "\n") ;

wire kount (kount.z1 initially 1) (evenodd.s, kount.x);
wire evenodd (kount.z) (outE.x, outO.x);
wire outE (evenodd.e) (outputE);
wire outO (evenodd.o) (outputO);
```

4.1.2 HAM code

```
Label "kount"
CopyInput 0
Push 3
CreateFrame 1

Label "kount_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
MkInt 1
PushVar 0
CallPrim "+"
PushVar 0
MkTuple 2
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "evenodd"
CopyInput 0
Push 3
```

```
CreateFrame 1

Label "evenodd_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
PushVar 0
Call "f_even"
Slide 1
If "t_evenodd_0_0"
PushVar 0
MkNone
MkTuple 2
Goto "n_evenodd_0_0"

Label "t_evenodd_0_0"
MkNone
PushVar 0
MkTuple 2

Label "n_evenodd_0_0"
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "outE"
CopyInput 0
Push 3
CreateFrame 1

Label "outE_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
MkString "\n"
PushVar 0
CallPrim "show"
CallPrim "++"
MkString "EVEN "
CallPrim "++"
CheckOutputs
Write 0
Schedule

Label "outO"
CopyInput 0
Push 3
CreateFrame 1
```

```
Label "outO_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
MkString "\n"
PushVar 0
CallPrim "show"
CallPrim "++"
MkString "ODD "
CallPrim "++"
CheckOutputs
Write 0
Schedule

Label "f_even"
CreateFrame 1

Label "f_even_0"
MatchRule
MatchVar 0
MatchedRule
MkInt 0
MkInt 2
PushVar 0
CallPrim "mod"
CallPrim "=="
Return
Function "f_even" "f_even_0"

Box "kount" "kount" 10 8 1 2 1 "kount_init" "kount_handler" NullT
Rule "kount" "kount_0"
Require "kount" True

Box "evenodd" "evenodd" 15 17 1 2 1 "evenodd_init" "evenodd_handler" NullT
Rule "evenodd" "evenodd_0"
Require "evenodd" True

Box "outE" "outE" 26 8 1 1 1 "outE_init" "outE_handler" NullT
Rule "outE" "outE_0"
Require "outE" True

Box "outO" "outO" 26 8 1 1 1 "outO_init" "outO_handler" NullT
Rule "outO" "outO_0"
Require "outO" True

Label "evenodd_init"
Schedule

Label "kount_init"
MkInt 1
Write 1
Schedule
```

```
Label "outE_init"
Schedule

Label "outO_init"
Schedule
Stream "outputE" Out "s_write" "std_out" 50 1 0 NullT
Stream "outputO" Out "s_write" "std_out" 50 1 0 NullT
Wire "outputE" 0 "outputE" 0 50 0 NullT
Wire "outputO" 0 "outputO" 0 50 0 NullT

Label "s_read"
Input
Write 0
Schedule

Label "s_write"
CopyInput 0
Consume 0
Output
Schedule

Label "s_timeout"
MkTuple 0
Raise "Timeout"

Label "s_soverflow"
MkTuple 0
Raise "StackOverflow"

Label "s_hoverflow"
MkTuple 0
Raise "HeapOverflow"
Wire "kount" 0 "kount" 1 2 0 NullT
Wire "evenodd" 0 "kount" 0 2 0 NullT
Wire "outE" 0 "evenodd" 0 2 0 NullT
Wire "outputE" 0 "outE" 0 2 0 NullT
Wire "outO" 0 "evenodd" 1 2 0 NullT
Wire "outputO" 0 "outO" 0 2 0 NullT
```

4.2 Fair Merge

This example realises a fair merge operation on two input streams, as depicted in Figure 9.

4.2.1 Hume code

```
program

stream output to "std_out";

type integer = int 32;

box k1
in  (x::integer)
out (z::integer, z1::integer)
match
```

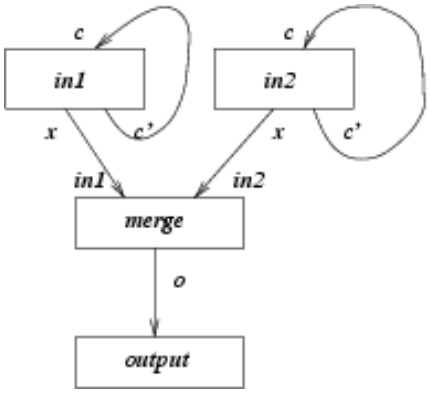

Figure 9: Wiring threads

```
 (x) -> (x,x);

box k2
in  (x::integer)
out (z::integer, z1::integer)
match
 (x) -> (x,x);

box merge
in  (i1, i2 :: integer)
out (o :: integer)
fair
  (x,*) -> x
| (*,y) -> y;

wire k1 (k1.z1 initially 1) (merge.i1, k1.x);
wire k2 (k2.z1 initially 2) (merge.i2, k2.x);
wire merge (k1.z, k2.z) (output);
```

4.2.2 HAM code

```
Label "k1"
CopyInput 0
Push 3
CreateFrame 1

Label "k1_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
PushVar 0
PushVar 0
MkTuple 2
Unpack
CheckOutputs
Write 0
Write 1
```

```
Schedule

Label "k2"
CopyInput 0
Push 3
CreateFrame 1

Label "k2_0"
MatchRule
MatchAvailable
MatchVar 0
Consume 0
MatchedRule
PushVar 0
PushVar 0
MkTuple 2
Unpack
CheckOutputs
Write 0
Write 1
Schedule

Label "merge"
CopyInput 1
CopyInput 0
Push 3
CreateFrame 1
StartMatches

Label "merge_0"
MatchRule
MatchAvailable
MatchVar 0
MatchNone
Consume 0
MatchedRule
PushVar 0
CheckOutputs
Write 0
Reorder
Schedule

Label "merge_1"
MatchRule
MatchNone
MatchAvailable
MatchVar 0
Consume 1
MatchedRule
PushVar 0
CheckOutputs
Write 0
Reorder
Schedule
```

```
Box "k1" "k1" 6 8 1 2 1 "k1_init" "k1_handler" NullT
Rule "k1" "k1_0"
Require "k1" True

Box "k2" "k2" 6 8 1 2 1 "k2_init" "k2_handler" NullT
Rule "k2" "k2_0"
Require "k2" True

Box "merge" "merge" 5 8 2 1 2 "merge_init" "merge_handler" NullT
Rule "merge" "merge_0" "merge_1"
Require "merge" True False
Require "merge" False True

Label "k1_init"
MkInt 1
Write 1
Schedule

Label "k2_init"
MkInt 2
Write 1
Schedule

Label "merge_init"
Schedule
Stream "output" Out "s_write" "std_out" 2 1 0 NullT
Wire "output" 0 "output" 0 2 0 NullT

Label "s_read"
Input
Write 0
Schedule

Label "s_write"
CopyInput 0
Consume 0
Output
Schedule

Label "s_timeout"
MkTuple 0
Raise "Timeout"

Label "s_soverflow"
MkTuple 0
Raise "StackOverflow"

Label "s_hoverflow"
MkTuple 0
Raise "HeapOverflow"
Wire "k1" 0 "k1" 1 2 0 NullT
Wire "k2" 0 "k2" 1 2 0 NullT
Wire "merge" 0 "k1" 0 2 0 NullT
Wire "merge" 1 "k2" 0 2 0 NullT
Wire "output" 0 "merge" 0 2 0 NullT
```

5 Summary

We have presented a formal description of the translation process from Hume to the Hume Abstract Machine (HAM). It is faithful to the compilation process implemented in the `phamc` compiler. It is sufficiently detailed to form the basis for reasoning about the resource consumption of Hume programs. We intend to formalise these translation functions in the Isabelle theorem prover, so that we can prove cost-correspondence between the cost model that we have developed for Hume (D12) and for the HAM (D4).

References

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

[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.