Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach

Kevin Hammond

School of Computer Science,
University of St Andrews, St Andrews, Scotland.
Email: kh@dcs.st-and.ac.uk. Tel: +44-1334-463241, Fax: +44-1334-463278

Abstract. This chapter describes Hume: a functionally-based language for programming with bounded resource usage, including time and space properties. The purpose of the Hume language design is to explore the expressibility/costability spectrum in resource-constrained systems, such as real-time embedded or control systems. It is unusual in being based on a combination of λ-calculus and finite state machine notions, rather than the more usual propositional logic, or flat finite-state-machine models. The use of a strict, purely functional programming notation allows the construction of a strong cost model for expressions, which can then be embedded into a simple cost model for processes.

In this chapter, we introduce Hume, describe the Hume Abstract Machine implementation, and show how a high-level cost model can be constructed that relates costs from the abstract machine to Hume source programs. We illustrate our approach with an example adapted from the literature: a simple vending machine controller.

1 Introduction

Hume is a functionally-based research language aimed at applications requiring bounded time and space behaviour, such as real-time embedded systems. Historically, the majority of embedded systems were implemented using low-level notations, often machine code. This was dictated by, firstly, the need to interface to devices at a very low level of abstraction; and, secondly, the need to minimise overall system costs by producing code that was highly efficient in both time and space usage. Since embedded applications were historically both small and simple, they could be (relatively) easily re-implemented if a change to a new architecture proved necessary. Encouraged by major improvements in the price/performance ratio of low- and mid-range devices, the demand for ever more complex applications, such as those found in mobile telephony, has, however, mandated a move to higher level languages such as C++, Ada or even perhaps Java. In this way, some precise low-level control has been sacrificed in favour of advantages of portability, speed of coding and code reuse. This trend is projected to continue with the introduction of Systems-on-a-(Programmable)-Chip. The challenge that must be met in designing future languages for embedded systems work is to preserve the essential properties of costability and low-level interfacing whilst providing as high-level a programming environment as possible.

Hume has three main research objectives: firstly, to explore the tension between expressibility and costability in order to determine how much abstraction can be included in a language design without losing strong formal properties of cost; secondly, to act as a "virtual laboratory" to allow the construction of new, advanced cost models and analyses in a relatively constrained setting; and thirdly to explore whether functional programming languages can plausibly be used to program real-time embedded systems.

1.1 Properties of a Real-Time Language

We can identify a number of essential or desirable properties for a language that is aimed at real-time embedded systems [18, 30].

- *determinacy* – the language should allow the construction of determinate systems, by which we mean that under identical environmental constraints, all executions of the system should be *observationally equivalent*;
- *bounded time/space* – the language must allow the construction of systems whose resource costs are statically bounded – so ensuring that *hard real-time* and *real-space* constraints can be met;
- *asynchronicity* – the language must allow the construction of systems that are capable of responding to inputs as they are received without imposing total ordering on environmental or internal interactions;
- *concurrency* – the language must allow the construction of systems as communicating units of independent computation;
- *correctness* – the language must allow a high degree of confidence that constructed systems meet their formal requirements [1].

Since functional languages have strong properties of determinacy and correctness, they are potentially good fits to the real-time systems domain provided it is possible to incorporate the remaining properties of concurrency, asynchronicity (where this is required), and above all boundedness. We will now explore this fit in the context of existing functional language designs.

1.2 Functional Languages and Real-Time Systems

Real-time systems can be categorised as being either *hard real-time* or *soft real-time*. Soft real-time has been described as a situation where "nothing really serious happens if a time constraint is not met" [4]. Conversely, hard real-time is characterised by situations of systems failure, mission loss, and even personal

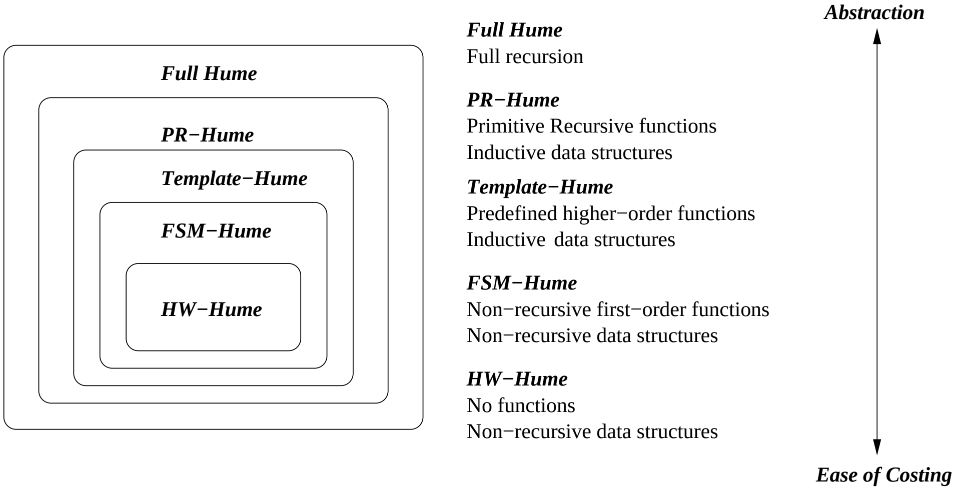
Full Hume
PR−Hume
Template−Hume
FSM−Hume
HW−Hume
Full Hume
Full recursion
PR−Hume
Primitive Recursive functions
Inductive data structures
Template−Hume
Predefined higher−order functions
Inductive data structures
FSM−Hume
Non−recursive first−order functions
Non−recursive data structures
HW−Hume
No functions
Non−recursive data structures
Abstraction
Ease of Costing

Fig. 1. Expressibility versus Costability in the Hume Design

injury or death. Since issues of cost and performance conflict with the high-level expressibility that is the general goal of most functional language designs, the majority of functional languages that have even considered real-time issues to date have focused on soft real-time rather than hard real-time. In soft real-time systems, time boundedness is, of course, non-critical; however, space boundedness may still be essential if memory overflows etc. are to be avoided.

The most widely used soft real-time functional language is the impure, strict language Erlang [5]. Erlang has been used by Ericsson to construct a number of successful telecommunications and related applications [3], including the Ericsson AXD301 ATM switch [8] and the soft real-time distributed database management system, Mnesia [50]. In Erlang, concurrent processes are constructed using explicit spawn operations, with communication occurring through explicit send and receive operations to nominated processes. While Erlang therefore supports asynchronicity, there is thus no strong notion of process network.

Another functional language targetting soft real-time systems is the experimental Embedded Gofer language [49]. Embedded Gofer is a strongly-typed purely functional programming language based on the standard non-strict functional language Haskell. Embedded Gofer has a two-level structure, where concurrent asynchronously communicating processes are defined using a monadic notation. These processes are then defined in terms of purely functional computations. As part of the monadic process level, Embedded Gofer includes explicit register access, process creation and communication primitives. This gives a direct, but relatively low-level approach to programming soft real-time systems, similar to that adopted by existing imperative languages, but with type guarantees providing separation between the two programming levels. Since Embedded Gofer has a non-strict evaluation model, however, it is not possible to provide strong behavioural guarantees on either time or space behaviour, as we require for bounded resource programming.

program ::=	decl_1 **;** ... **;** decl_n	$n \geq 1$
decl ::=	*boxdecl* $\mid$ *vardecl* $\mid$ *fundecl* $\mid$ *datadecl* $\mid$ *wiredecl*	
boxdecl ::=	**box** boxid *ins* *outs* **fair/unfair** *matches*	
ins/outs ::=	**(** wireid_1 **,** ... **,** wireid_n **)**	$n \geq 0$
matches ::=	match_1 **;** ... **;** match_n	$n \geq 1$
match ::=	pat_1 ... pat_n $\rightarrow$ *exp*	$n \geq 1$
pat ::=	*int* $\mid$ *float* $\mid$ *char* $\mid$ *bool* $\mid$ *string* $\mid$ varid $\mid$ _ $\mid$ *****	
	$\mid$ conid varid_1 ... varid_n	$n \geq 0$
	$\mid$ **(** pat_1 **,** ... **,** pat_n **)**	$n \geq 2$
	$\mid$ $\langle\langle$ pat_1 **,** ... **,** pat_n $\rangle\rangle$	$n \geq 0$
exp ::=	*int* $\mid$ *float* $\mid$ *char* $\mid$ *bool* $\mid$ *string* $\mid$ varid $\mid$ *****	
	$\mid$ funid exp_1 ... exp_n	$n \geq 1$
	$\mid$ varid exp_1 ... exp_n	$n \geq 1$
	$\mid$ primid exp_1 ... exp_n	$0 < n \leq 3$
	$\mid$ conid exp_1 ... exp_n	$n \geq 0$
	$\mid$ **(** exp_1 **,** ... **,** exp_n **)**	$n \geq 2$
	$\mid$ $\langle\langle$ exp_1 **,** ... **,** exp_n $\rangle\rangle$	$n \geq 0$
	$\mid$ **if** exp_1 **then** exp_2 **else** exp_3	
	$\mid$ **case** *exp* **of** *matches*	
	$\mid$ *exp* **within** *constraint*	
	$\mid$ *exp* **as** τ	
	$\mid$ *exp* **::** τ	
	$\mid$ **let** $\mathit{vardecl}_1$ **;** ... **;** $\mathit{vardecl}_n$ **in** *exp*	$n \geq 1$
constraint ::=	timeconst $\mid$ spaceconst	
vardecl ::=	varid **=** *exp*	
fundecl ::=	funid *matches*	
datadecl ::=	**data** typeid tvarid_1 ... tvarid_n **=** *constrs*	$n \geq 0$
constrs ::=	constr_1 **|** ... **|** constr_n	$n \geq 1$
constr ::=	conid τ_1 ... τ_n	$n \geq 0$
τ ::=	typeid $\tau_1 \ldots \tau_n$ $\mid$ tvarid $\mid$ $\tau_1 \rightarrow \tau_2$	$n \geq 0$
wiredecl ::=	**wire** port_1 **to** port_2	
port ::=	boxid **.** wireid $\mid$ deviceid	

Fig. 2. Hume Abstract Syntax (Main Constructs)

A similar approach to Embedded Gofer has also been taken by Fijma and Udink, who introduced special language constructs in a purely functional language to control a robot arm [16]; and the Timber language also uses *monadic* constructs for specifying strong real-time properties [33].

Finally, a number of functional languages have been designed for programming reactive systems, without considering real-time properties. Such languages will typically incorporate concurrent process creation, inter-process communication and synchronisation constructs. Examples include Concurrent Clean [32], Concurrent ML [42], Concurrent Haskell [40] and Eden [11]. One recent example is Frob (Functional Robotics) [37], which provides monadic support for timed events, tasks and behaviours, and which has been used successfully on the Yale robotics course. Frob is primarily intended to explore issues of high-level expressibility, rather than control systems, real-time systems or bounded space.

1.3 The Hume Design Philosophy

Like many of the reactive functional languages described above, and also like Embedded Gofer, Hume takes a two-level approach to language design, where a purely functional expression layer is embedded within a process layer that describes communicating processes. Where Embedded Gofer uses monads to encapsulate processes, Eden uses a process construct within a functional expression, and Concurrent ML uses side-effecting process creation and communication constructs, Hume makes the separation more explicit by introducing a syntactically distinct process notation, and uses implicit communication.

Costability is, of course, key to the Hume design. Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework either directly or by altering the language to a greater or lesser extent as with e.g. RTSj [9], our approach is to design Hume in such a way that we are *certain* that formal cost models and the associated correctness proofs can be constructed for all Hume language constructs. In order to provide structure to these notions of cost, we envisage a series of overlapping language subsets as shown in Figure 1, where each superset adds expressibility to the expression layer, but either loses some aspect of decidability over our formal properties or increases the difficulty of providing formal correctness/cost models. By choosing an appropriate language level, the programmer can obtain the required balance between expressibility and costability.

2 Boxes and Coordination

In order to support concurrency, Hume requires both computation and coordination constructs. The fundamental unit of computation in Hume is the *box*, which defines a finite mapping from inputs to outputs in a functional manner. Boxes are *wired* into (static) networks of concurrent processes using wiring directives. Each box introduces one process. This section introduces such notions informally.

Figure 2 shows the abstract syntax of Hume. We have chosen to use a rule-based approach to our language design, with a fairly conventional purely-functional

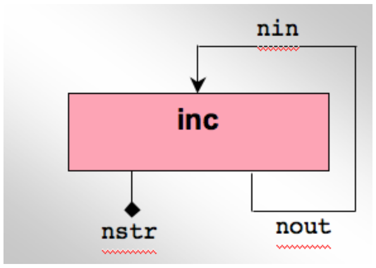

Fig. 3. System diagram for a simple incrementing box

expression notation embedded in an asynchronous process model. This simplifies both correctness proofs and the construction of cost models at the expression level. There are four unconventional expression forms: $\langle\langle\ \ldots\ \rangle\rangle$ is a vector pattern or expression; *exp* **within** *constraint* expresses a checkable constraint on time- or space-usage; *exp* **as** τ indicates a dynamic coercion to the specified type τ; and ***** is used to define asynchronous programs, as described in Section 2.3.

Boxes are abstractions of processes that correspond to (usually finite) state machines. A single Hume box comprises a set of pattern-directed rules, rewriting a set of inputs to a set of outputs, plus appropriate exception handlers and type information. The left-hand-side (pattern part) of each rule defines the situations in which that rule may be active. The right-hand-side of each rule is an expression specifying the results of the box when the rule is activated and matches the corresponding pattern. A box becomes active when any of its rules may match the inputs that have been provided. For example, we can define a box `inc` that simultaneously increments its input `nin` and outputs it as a fixed-width string as follows. Figure 3 shows a system diagram of this box.

```
box inc
in  (nin :: int 32)
out (nout ::int 32, nstr::string 11)
match
 n -> (n+1, n as string 10 ++ "\n");
```

We first specify the types of the inputs (the singleton `nin`, an integer) and outputs (the pair `nout`, an integer, and `nstr`, a string) to the box. We then specify the single pattern-matching rule that takes a single integer value `n` and produces two outputs: the incremented value of `n`; and the original input value converted to a fixed-width string using the `as` construct. The Hume design requires the use of explicit type-coercion in order to make clear that time and space costs may be incurred at this point in the program.

2.1 Wiring

Boxes are connected using wiring declarations to form a static process network, with each wire mapping a specific box output to a specific input. Each box output must be connected to precisely one input, and each input must have precisely one output connected to it. In addition to the usual wires connecting boxes, inputs/outputs may be wired from/to external devices, such as I/O streams or ports attached to hardware devices. It is also possible to specify the initial value that appears on a wire. This is typically used to seed computations, such as wires carrying explicit state parameters, where an output wire from a box is wired to an input wire from the same box. For example, we could wire the `inc` box as follows, where `inc.nin`/`inc.nout` form a pair of wires carrying an explicit state that is initialised to 0, and `inc.nstr` is connected to the standard output stream.

```
wire inc.nout to inc.nin initially 0;
wire inc.nstr to output;

stream output to "std_out";
```

2.2 Coordination

Having considered how a static process network can be constructed from a set of boxes, we now turn our attention to how boxes are executed and scheduled. The basic box execution cycle in Hume is:

1. check input availability for all box inputs and latch the input values;
2. match box inputs against the box rules in turn;
3. consume all box inputs;
4. bind variables to input values and evaluate the RHS of the selected rule;
5. write box outputs to the corresponding wires.

A key issue is the management of input and output values on wires. As we have seen, in the Hume model, there is a one-to-one correspondance between input- and output-wires. Each of these wires is *single-buffered*. Since we require data types to convey bounded size information , this ensures that communication buffers are also of bounded size. It also avoids the synchronisation problems that can occur if no buffering is used. In particular, a box may write an output to one of its own inputs, so creating an explicit representation of state, as shown in the `inc` example above.

Values for all available inputs are latched atomically, but are not removed from the wire buffer (*consumed*) until a rule is matched. Consuming an input removes the lock on the wire buffer, resetting the availability for that input. Outputs are written atomically: if any output cannot be written to its buffer because a previous value has not yet been consumed, the box blocks. This reduces concurrency by preventing boxes from proceeding if their inputs could be made available but the producer is blocked on some other output, so potentially introducing deadlock/livelock in some situtations. However, it also improves strong notions of *causality* [43]: if a value has appeared as input on a wire the box that produced that input has certainly generated all of its outputs.

Once a box execution cycle has completed and all outputs have been written to the corresponding wire buffers, the box becomes available for execution in the next scheduling cycle as described in Section 2.4. This improves concurrency, by avoiding unnecessary synchronisation. Note that individual Hume boxes will never terminate. Program termination occurs when no box is runnable and no external input can become available in future. This reflects the requirements of the embedded systems domain that we are targetting: programs should not normally terminate, but should be available to react to any external inputs that may become available at any point during their execution.

2.3 Asynchronous Coordination Constructs

So far, we have described an essentially synchronous coordination model. However, many real-time applications benefit from asynchronicity in its various forms. One of the interesting features of Hume is that it goes beyond the usual synchronous programming models such as Lustre [13], Signal [17] or Esterel [10], whilst maintaining the strong cost properties that are required for embedded systems programming. The two primary coordination constructs that are used to introduce asynchronous coordination are to *ignore* certain inputs/outputs and to introduce *fair matching*. In order to deal with these asynchronous constructs, it is necessary to alter the basic box execution cycle as follows (changes are italicised):

1. check input availability *against possible matches* and latch *available* input values;
2. match *available* inputs against box rules in turn;
3. consume *those inputs that have been matched and which are not ignored in the selected rule*;
4. bind variables to input values and evaluate the RHS of the selected rule;
5. write *non-ignored* outputs to the corresponding wires;
6. *reorder match rules according to the fairness criteria*.

Note that: i) inputs are now consumed after rules have been selected rather than before; ii) only some inputs/outputs may be involved in a given box cycle, rather than all inputs/outputs being required; and iii) rules may be reordered if the box is engaged in fair matching. This new model in which inputs can be ignored in certain patterns or in certain output positions can be considered to be similar to *non-strictness* at the box level.

We use the accepted notion of *fairness* whereby each rule will be used equally often given a stream of inputs that match all rules [2]. *Channel fairness* [2] is not enforced, however: it is entirely possible, for example, for a programmer to write a sequence of rules that will treat the input from different sources unfairly. It is the programmer's responsibility to ensure that channel fairness is maintained, if required.

For example, a fair merge operation, which selects values alternately from its two inputs, can be defined as:

```
box merge
in  ( xs  :: int 32, ys :: int 32)
out ( xys :: int 32)
fair
  (x, *) -> x
| (*, y) -> y
;
```

The *-pattern indicates that the corresponding input position should be ignored, that is the *-pattern matches *any* input, without consuming it. Note the difference between *-patterns and wildcard/variable patterns: in the latter cases, successful matching will mean that the corresponding input value (and all of that value) is removed from the input buffer. * can be used as an expression. In this case no output is produced on the corresponding wire, and consequently the box cannot be blocked on that output.

Fair merging is an example of an operation that cannot be expressed easily in a single-layer purely functional notation, since it is non-deterministic at the box level. There have been several attempts in the literature to resolve this problem, including *hiatons* [48] (empty values similar in effect to our * notation, but which incur cost even if no value is present on an input), computations over *sets* of values [23], the introduction of *non-deterministic monads* [28], and the use of explicit concurrency in the Eden language [11]. We argue that the two-layer approach used in Hume offers a simple and clean solution to this problem, by properly encapsulating the notion of a process as a separate construct not embedded into the single functional layer. In this way, all underlying properties of the functional language are preserved without the need for complex analyses to determine whether a computation might be non-deterministic, as is required with Eden [35].

Note that despite this local idea of non-determinacy in Hume (which is an essential part of the specification of this problem), *the system as a whole* is still deterministic. Since we use a least-recently used notion of fairness on rules, systems may be replayed from any known intermediate state, yielding identical results to those originally obtained.

2.4 Thread Scheduling

The prototype Hume Abstract Machine implementation maintains a vector of threads (*thread*), one per box, each with its own *thread state record*, containing

```
for $i = 1$ to $nThreads$ do
    $runnable := false$;
    for $j = 1$ to $thread[i].nRules$ do
        if $\neg$ $runnable$ then
            $runnable := true$;
            for $k = 1$ to $thread[i].nIns$ do
                $runnable \mathbin{\&}= thread[i].required[j,k] \Rightarrow thread[i].ins[k].available$
            endfor
        endif
    endfor
    if $runnable$ then $schedule$ ($thread[i]$) endif
endfor
```

Fig. 4. Hume Abstract Machine Thread Scheduling Algorithm

state information and links to input/output wires. Each wire comprises a pair of a value (*value*) and a validity flag (*available*). used to ensure correct locking between input and output threads. The flag is atomically set to *true* when an output is written to the wire, and is reset to *false* when an input is consumed.

Threads are scheduled under the control of a built-in scheduler, which currently implements round-robin scheduling. A thread 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 thread t, $thread[t].required[r,i]$ is true if input i is required to run rule r of that thread. Since wires are single-buffered, a thread will consequently 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 immediately before any output is written. No output will be written until all the input on all output wires has been consumed. The check ignores * output positions.

2.5 Hume Example: a Vending Machine

We will now illustrate Hume with a simple, but more realistic, example from the reactive systems literature, suggested to us at CEFP 2005 by Pieter Koopman: the control logic for a simple vending machine. A system diagram is shown in Figure 5. We will show Hume code only for the most important part of the system: the *control* box. This box responds to inputs from the keypad box and the cash holder box representing presses of a button (for tea, coffee, or a refund) or coins (nickels/dimes) being loaded into the cash box. In a real system, these boxes would probably be implemented as hardware components. If a drinks button (tea/coffee) is pressed, then the controller determines whether a sufficient value of coins has been deposited for the requested drink using the `do_dispense` function. If so, the vending unit is instructed to produce the requested drink.

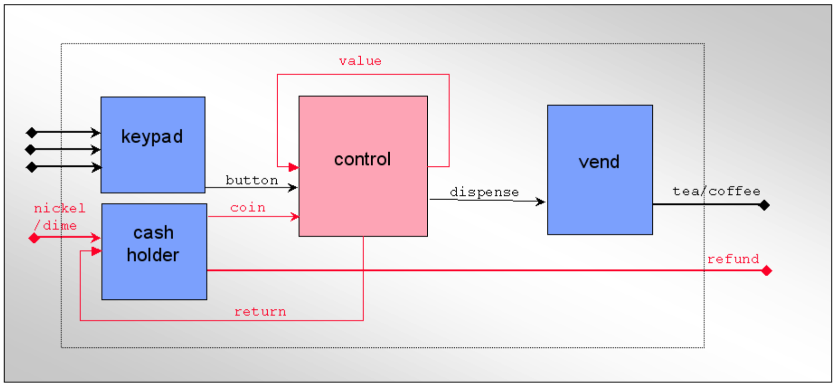

Fig. 5. Hume example: vending machine box diagram

Otherwise, the button press is ignored. If the cancel button is pressed, or the control unit does not respond within 30s of a button being pressed, then the cash box is instructed to refund the value of the input coins to the consumer.

First we define some basic types representing the value of coins held in the machine (`Cash`), the different types of coins (`Coins`), the drinks that can be dispensed (`Drinks`) and the buttons that can be pressed (`Buttons`). We also define the maximum value of coins that can be input by a single consumer (`MAX_VALUE`).

```
type Cash = int 8;

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

constant MAX_VALUE = 100;
```

Now we can define the `control` box itself. This box uses *asychronous* constructs to react to each of the two input possibilities: either an inserted coin or a button-press. As with the `inc` box, state is maintained explicity through a feedback wire: the `value'` output will be wired directly to the `value` input. For simplicity, the box uses unfair matching, which will prioritise coin inputs over simultaneous button presses. If the cancel button (`BCancel`) is pressed, no drink will be dispensed (shown by `*`), but the internal cash value will be reset to zero and the cash holder instructed to refund the current value of coins held (`value`) through the `return` wire. A timeout has the same effect as explicitly pressing the cancel button.

```
-- vending machine control box
```

```
box control
in  ( coin :: Coins, button :: Buttons, value :: Cash )
out ( dispense :: Drinks, value' :: Cash, return :: Cash )
match
    ( Nickel, *,       v )  ->  add_value v 5
  | ( Dime,   *,       v )  ->  add_value v 10
  | ( *,      BCoffee, v )  ->  do_dispense Coffee 10 v
  | ( *,      BTea,    v )  ->  do_dispense Tea 5 v
  | ( *,      BCancel, v )  ->  ( *, 0, v )
handle
  TimeOut ( *, *, v) -> ( *, 0, v );
```

The control box logic makes use of two auxiliary functions: `do_dispense` calculates whether sufficient coins have been deposited and instructs the `vend` box accordingly; and `add_value` increments the value held in the cash box by the value of the deposited coin. Note the use of `*` as a return value in the function definition: this is permitted only in positions which correspond to top-level outputs. Note also that the box corresponds to the FSM-Hume level of Section 1.3: it uses first-order non-recursive functions as part of its definition.

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

add_value v coin =
   let v' = v + coin in
   if v' > MAX_VALUE then ( *, v,  coin )
                     else ( *, v', *    );
```

Finally, we wire the control box to the other boxes shown in the system diagram. Note that the `button` wire is instructed to carry a 30s timeout, and the `value`/`value'` pair are wired together with an initial value of 0. This ensures that the system is properly initialised and can proceed when started.

```
wire cashbox.coin_in        to control.coin;
wire keypad.button_pressed  to control.button timeout 30s;
wire control.value'         to control.value initially 0;
wire control.dispense       to vend.dispense;
wire control.return         to cashbox.refund;
```

3 Hume Abstract Machine Design

We will now show how Hume programs may be compiled into an abstract machine representation which will allow concrete time and space cost information

name	interpretation
S	stack
H	heap
sp	stack pointer
hp	heap pointer
fp	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
EXNPC	exception program counter
ins	input buffers
outs	output buffers
nIns	number of inputs
nOuts	number of outputs
timeout	current timeout value
thandler	pc for timeout handler

Fig. 6. Thread-specific registers, constants and memory areas – the *thread state record*

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

Fig. 7. Ruleset-specific registers and constants

to be extracted. The goal of the prototype Hume Abstract Machine (pHAM) design is to provide a credible basis for research into bounded time and space computation, allowing formal cost models to be verified against a realistic implementation. An important part of this work is to provide a formal and precise translation scheme as given in Section 4. We have therefore defined the pHAM at a level which is similar to, but slightly more abstract than, the JVM [29], giving a formalised description of the abstract machine in terms of a simple abstract register language which can be easily mapped to machine code. Absolute space- and time-performance (while an important long-term objective for Hume) is thus less important in this initial design than predictability, simplicity and ease of implementation.

3.1 Outline Design

The prototype Hume Abstract Machine is loosely based on the design of the classical G-Machine [6] or SECD-Machine [27]. with extensions to manage concurrency and asynchronicity. Each Hume box is implemented as a thread with its own dynamic stack (S) and heap (H) and associated stack and heap pointers (*sp* and *hp*). These and the other items that form part of the individual *thread state record* are shown in Figure 6. Each function and box has an associated *ruleset* (Figure 7). The ruleset is used for two purposes: it gives the address of the next rule to try if matching fails; and it is used to reorder rules if fair matching is specified. The box ruleset is specified as the *base* field of the thread state record. Function rulesets are set as part of a function call.

The motivation for a separate stack for each thread is to maintain independence between thread states. Similarly, separate heaps allow a simple, real-time model of garbage collection where the *entire heap* for a box becomes garbage each time a thread completes, and all heaps can therefore be allocated from the same common dynamic memory area. The sizes of all stack and heap spaces are fixed at compile-time using the static analysis described in Section 5, and in principle, small pointer ranges (e.g. 8 bits) can be used in either case. The main disadvantage of our present design is that we cannot use physically- or virtually-shared heap to communicate arguments and results between threads. Rather, such values are copied between heaps at the beginning and end of thread execution as explicit values within wire buffers. In effect, we have a simple copying garbage collector for which liveness can be trivially determined.

There is an analogy with the *working copies* of global variables that may be obtained by implementations of the JVM [29]. However, variable accesses in the JVM may occur at any point during thread execution, not only at the beginning/end as in the pHAM. Moreover, unlike the pHAM, which is stateless, the JVM maintains a virtually shared heap containing *master copies* of each variable. Our design is thus closer to that of Eden [11]: a reactive functional language based on Haskell.

The pHAM design uses a pure stack calling convention. Function arguments are followed by a three-item subframe containing the return address, a pointer to the previous ruleset, and the previous frame pointer. In the rules that follow, the size of this subframe is given by the constant $\mathcal{S}_{frame}$. The local frame pointer *fp* points immediately after this subframe, to the address of the first local variable. For consistency, the same layout is used at the outer thread level. In this case, the box inputs are stored in the argument position, and the return address item is redundant. All values on the stack other than the saved return address, ruleset and frame pointer are local heap pointers (i.e. they are *boxed* [38]). Moreover, in the current design there is no separate *basic value stack* to handle scalar values as in some versions of the G-Machine [38], STG-Machine [39] etc. nor are scalars and heap objects mixed on the stack as in the JVM [29].

3.2 The Hume Abstract Machine Instructions

The abstract machine instructions implement the abstract machine design described above. An operational description of these instructions is given in Figures 8–13. We use a number of auxiliary definitions: *copy* creates a copy of a wire value in the appropriate heap/wire; *getchar* reads a character from the specified stream; *putvalue* writes a representation of its value argument to the specified stream; and *reschedule* terminates the current box execution, passing control to the abstract machine scheduler. $\mathcal{H}_{con}$, $\mathcal{H}_{int32}$ etc. are constants defining the sizes

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

Fig. 8. Hume Abstract Machine Instructions (Heap and Stack Manipulation)

of heap objects. A number of pseudo-instructions: **Box**, **Stream**, **Wire**, **Label**, **Function**, **Rule** and **Require** are also used to provide information about program structure that is exploited by the abstract machine implementation (Figure 11).

Heap Object Creation and Stack Manipulation (Figure 8). Tagged objects are created in the heap, and pointers to the new object stored on the top of the stack. For scalar values (booleans, characters, integers, floats and strings – **MkInt32** etc.), the actual value is taken directly from the instruction stream. For strings, this value is a pointer into a global shared string table). The instruc-

```
Goto $l$           $pc := l$
If $l$             if$S[sp-1] = true$ then $pc := l$ else $++pc$ endif; $--sp$

Call $f$           $S[sp] := pc+1; S[sp+1] := rs; ++sp;$
                   $rs := f.ruleset; rs.rp := 0; pc := rs.rules[0]$

Return $l$         $rs := S[fp-2]; pc := S[fp-3]; sp' := fp - \mathcal{S}_{frame};$
                   $fp := S[fp-1]; S[sp'] := S[sp-1]; sp := sp'+1$

CallPrim1 $p$      $S[sp] := p\ (S[sp]); ++pc$
CallPrim2 $p$      $S[sp-1] := p\ (S[sp])\ (S[sp-1]); --sp; ++pc$
CallPrim3 $p$      $S[sp-1] := p\ (S[sp])\ (S[sp-1])\ (S[sp-2]); sp := sp-2; ++pc$

CallVar $v$ $x$    let $H[v]$ be Fun $f\ m\ n\ a_1 \ldots\ a_m$ in
                   if $m + x \geq n$ then
                     for $i\ =\ 1$ to $m$ do
                       $S[sp+i-1] := a_i$;
                     $sp := sp + m$;
                     Call $f$;
                   else
                    $H[hp] :=$ Fun $f\ (m+x)\ n\ a_1\ \ldots\ a_m\ S[sp-1]\ \ldots\ S[sp-x-1]$;
                     $sp := sp - x;\ S[sp-1] := hp;\ hp := hp + \mathcal{H}_{fun} + m + x;\ ++pc$

AP $n$             $--\ sp;$ CallVar$(S[sp])n$;
```

Fig. 9. Hume Abstract Machine Instructions (Control)

tion **MkNone** creates a special value **None**, which is tested in the **CheckOutputs/Write** instructions. Finally, **MkCon** builds user-defined data structures of a given size, **MkTuple** builds tuples and **MkVector** builds vectors.

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

Control Operations (Figure 9). The Hume abstract machine control instructions are shown in Figure 9. **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

```
MatchRule          $mp := fp - \mathcal{S}_{frame} + 1; inp := 0; pc := rs.rules[rp]; {+}{+}rp$

MatchNone          ${-}{-}mp; {+}{+}inp; {+}{+}pc$
MatchAvailable     if $\neg\ ins[inp].available$ then $pc := rules[rp]$ endif; $inp := inp + 1$

MatchBool $b$        ${-}{-}mp;$ if $H[S[mp]] \neq$ Bool $b$ then $pc := rules[rp]$ endif
MatchChar $c$        ${-}{-}mp;$ if $H[S[mp]] \neq$ Char $c$ then $pc := rules[rp]$ endif
MatchString $s$      ${-}{-}mp;$ if $H[S[mp]] \neq$ String $s$ then $pc := rules[rp]$ endif
MatchInt32 $i$       ${-}{-}mp;$ if $H[S[mp]] \neq$ Int32 $i$ then $pc := rules[rp]$ endif
MatchFloat32 $f$     ${-}{-}mp;$ if $H[S[mp]] \neq$ Float32 $f$ then $pc := rules[rp]$ endif
...
MatchCon $c\ n$      ${-}{-}mp;$ if $H[S[mp]] \neq$ Con $c\ n$ then $pc := rules[rp]$ endif
MatchTuple $n$       ${-}{-}mp;$ if $H[S[mp]] \neq$ Tuple $n$ then $pc := rules[rp]$ endif
MatchVector $s$      ${-}{-}mp;$ if $H[S[mp]] \neq$ Vector $s$ then $pc := rules[rp]$ endif

Unpack             let offset =
                     if $H[S[{-}{-}sp]] =$ Tuple $n$ then 2
                     else if $H[S[sp]] =$ Con $c\ n$ then 3 in
                     else if $H[S[sp]] =$ Vector $n$ then 2 in
                       for $i = 0$ to $n - 1$ do $S[sp{+}{+}] := H[hp + \text{offset} + i]$; endfor;
                   ${+}{+}pc$

StartMatches       $pc := base.rules[0]$

Reorder            let $n = rs.nRules - 1; r = rs.rules[rp]$ in
                       for $i = rp$ to $n$ do
                           $rs.rules[i] := rs.rules[i + 1]$
                       endfor;
                   $rs.rules[n] := r; {+}{+}pc$
```

Fig. 10. Hume Abstract Machine Instructions (Pattern Matching)

specified function, saves the current ruleset on the stack, and updates the ruleset. **CallPrim1/2/3** call primitive (built-in) functions with the corresponding numbers of arguments.

Pattern matching (Figure 10). We use a set of high level pattern matching instructions rather than compiling into a series of case matches as with e.g. the STG-Machine [39]. Thread matching is initiated by the **StartMatches** instruction, which sets the program counter to the first rule in the base ruleset. Identical matching operations are used both for box inputs and for function arguments. The operations are divided into three sets: the **MatchRule** operation which initialises the matching for a rule; the **MatchAvailable** and **MatchNone** operations which check box input availability (**MatchNone** for *****-patterns); and the value matching operations such as **MatchInt32** or **MatchCon**, which use

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

Fig. 11. Hume Abstract Machine Pseudo-Instructions

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

Fig. 12. Hume Abstract Machine Instructions (Exceptions)

the current match pointer, *mp*. Nested matching is achieved by unpacking the arguments onto the stack using **Unpack**. Finally rules may be reordered if fair matching is required using **Reorder**.

Exceptions (Figure 12). Exceptions are raised by the **Raise** instruction, which constructs the relevant exception value and then transfer control to the box's exception handler (EXNPC). **within**-expressions are managed by three instructions, which manipulate *timeout* and *thandler*: provided the new timeout t is earlier than the current timeout, *timeout*, the **Within** instruction will stack the previous timeout value, together with the timeout handler, *thandler*. If a timeout occurs, then control will be transferred to the timeout handler, whose first action will be to use a **RaiseWithin** instruction to disable the timeout, by setting the timeout so that it will never occur. Finally, if the expression doesn't trigger a timeout, then **DoneWithin** will restore the previous *timeout* and *thandler* values from the stack. Similar instructions (not shown here) are used to handle space restrictions.

Thread input/output and rescheduling operations (Figure 13). Thread input and output on wires is handled by two sets of operations. The **CopyInput** instruction copies the specified input from the input wire into the heap and places it on the top of the stack prior to matching. If matching is successful, input is *consumed* using the **Consume** operation, which resets the availability flag for

```
CopyInput $n$      $S[sp] := copy\ (ins[n].value); ++sp; ++pc$
Consume $n$        $ins[n].available := false; ++pc$

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

Write $n$          if $H[S[--sp]]\ \neq$ None then
                       $outs[n].value := copy(H[S[sp]]); outs[n].available := true;$
                   endif; $++pc$

Input $s$          let $c = getchar\ s$ in
                       $H[hp] :=$ Char $c; S[sp] := hp; ++sp;$
                       $hp := hp := hp + \mathcal{H}_{char}; ++pc$

Output $s$         $putvalue(s, S[sp-1]); --sp; ++pc$

Schedule           $reschedule$
```

Fig. 13. Hume Abstract Machine Instructions (Threads)

the appropriate input wire, thereby permitting subsequent **Write** instructions to succeed for that wire.

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

Control is returned to the scheduler either when a thread blocks, either as a consequence of being unable to write some output during the **CheckOutputs** operation, or explicitly when a thread terminates as a consequence of executing the **Schedule** operation. In either case, the scheduler will select a new runnable thread to execute. If there is no runnable thread, then in the current implementation the system will terminate. In a distributed system, it would be necessary to check for global termination, including outstanding communications that could awaken some thread.

Finally, two operations are provided to manage stream and device input/output. A special I/O thread is attached to each stream/device by the **Stream** pseudo-instruction (Figure 11). Executing the **Input** operation blocks this thread if no input is available, but otherwise reads input into the thread's heap. The **Output** operation simply writes the top stack value to the appropriate device. Normal wire operations are used to interface other threads to these special I/O threads. For simplicity, we only show character-level I/O, but more complex I/O can also be managed in a similar way.

4 Compilation Scheme

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

Figures 14–18 outline rules for compiling Hume abstract syntax forms into the abstract machine instructions described in Section 3, as a formal compilation scheme similar to that for the G-machine [6]. These rules have been used to construct a compiler from Hume source code to the pHAM, whose main component is a 500-line Haskell module translating abstract syntax to pHAM instructions.

The compilation scheme makes extensive use of a simple sequence notation: $\langle\ i_1,\ \ldots,\ i_n\ \rangle$ 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 definitions taken from a declaration sequence – the *depth* of these new bindings is 0, whilst the depth of existing variable bindings in the environment is incremented by 1; *bindVars* does the same for a sequence of patterns; and *labels* generates new labels for a set of function/box rules. Note that where labels *lt*, *ln*, *lx* etc. are used, these are assumed to be unique in the obvious way: there is at most one **Label** pseudo-instruction for each label in the translated program. Labels for boxes and function blocks are derived in a standard way from the (unique) name of the box or function. Finally, priming (e.g. ρ') has no semantic significance as in mathematics: it is used here for naming purposes only.

The rules are structured by abstract syntax class. The rules for translating expressions ($\mathcal{C}_E$ etc. – Figures 14–15) are generally straightforward, but note that function frames are created to deal with *let*-expressions and other similar structures, which then exploit the function calling mechanism. This allows the creation of local stack frames. It would obviously be possible to eliminate the

$$
\begin{array}{lcl}
\mathcal{C}_E\ \rho\ (c\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e_1 \mathbin{+\!\!+} \langle\ \text{MkCon}\ c\ n\ \rangle \\
\mathcal{C}_E\ \rho\ (p\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e_1 \mathbin{+\!\!+} \langle\ \text{CallPrim}n\ p\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (f\ e_1\ \ldots\ e_n) & = & \textbf{let}\ a\ = \mathit{arity}\ f\ \textbf{in} \\
 & & \mathcal{C}_E\ \rho\ e_n \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e_1 \mathbin{+\!\!+} \\
 & & \quad \textbf{if}\ n = a\ \textbf{then}\ \ \langle\ \text{Call}\ f, \text{Slide}\ n\ \rangle \\
 & & \quad \textbf{else if}\ n < a\ \textbf{then}\ \ \langle\ \text{MkFun}\ \rangle \\
 & & \quad \textbf{else}\ \ \langle\ \text{Call}\ f,\ \text{Slide}\ n,\ \text{AP}\ (n-a)\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (v\ e_1\ \ldots\ e_n) & = & \mathcal{C}_E\ \rho\ e_n \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e_1 \mathbin{+\!\!+} \\
 & & \quad \langle\ \text{CallVar}\ v\ n, \text{SlideVar}\ v\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (i) & = & \langle\ \text{MkInt32}\ i\ \rangle \\
\ldots & & \\
\mathcal{C}_E\ \rho\ (*) & = & \langle\ \text{MkNone}\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (\ e_1\ ,\ \ldots\ ,\ e_n\ ) & = & \mathcal{C}_E\ \rho\ e_n \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e_1 \mathbin{+\!\!+} \langle\ \text{MkTuple}\ n\ \rangle \\
\mathcal{C}_E\ \rho\ \langle\ \langle\ e_1\ ,\ \ldots\ ,\ e_n\ \rangle\ \rangle & = & \mathcal{C}_E\ \rho\ e_n \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e_1 \mathbin{+\!\!+} \langle\ \text{MkVector}\ n\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (\mathit{var}) & = & \langle\ \text{PushVar}\ n\ \rangle
\end{array}
$$

Fig. 14. Compilation Rules for Expressions (1)

function call for *let*-expressions provided the stack frame was properly set up in order to allow access to non-local definitions. Note also the three cases for function application: respectively corresponding to the usual first-order case, to under-application of a function (where a closure will be created using **MkFun**), and to over-application of a function (where the closure will be applied to the additional arguments using **Ap**).

In order to avoid the increase in closure sizes that would result from lambda-lifting [22], we instead use a *static link pointer* approach, where each new stack frame is linked at runtime to the frame corresponding to the function that statically encloses the current function body. This gives a space and time cost model that can be more easily related to the source, since we will not transform value definitions into functions as a result of lifting free variables. It also reduces the number and size of functional closures that must be created. However, it does increase the (fixed) size for each stack frame, since a static link pointer to the enclosing scope must be stored in each frame. In order to exploit this approach, we introduce variants of the **PushVar**, **SlidaVar**, **CallVar** instructions that use the static link pointer to locate the variable definition from the correct statically linked frame. For simplicity, we have omitted these instructions here.

The rules for translating box and function declarations are shown in Figure 16. 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

$$\begin{array}{lcl}
\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\ (labels\ lc)\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (\textbf{let}\ d_1\ \ldots\ d_n\ \textbf{in}\ e) & = & \textbf{let}\ \rho' = 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\ (\ e\ \textbf{as}\ \tau) & = & \mathcal{C}_E\ \rho\ e +\!\!+ \langle\ \text{CallPrim1}\ Coerce-\tau\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (\ e\ \textbf{within}\ c) & = & \langle\ \text{Within}\ lw\ c\ \rangle +\!\!+ \langle\ \mathcal{C}_E\ \rho\ e +\!\!+ \langle\ \text{Goto}\ ln\ \rangle +\!\!+ \\
 & & \langle\ \text{Label}\ lw,\ \text{RaiseWithin}\ \rangle +\!\!+ \\
 & & \langle\ \text{MkTuple 0},\ \text{Raise}\ Timeout/HeapOverflow\ \rangle +\!\!+ \\
 & & \langle\ \text{Label}\ ln,\ \text{DoneWithin}\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (\textbf{raise}\ \text{exnid}\ e) & = & \mathcal{C}_E\ \rho\ e +\!\!+ \langle\ \text{Raise exnid}\ \rangle \\
 & & \\
\mathcal{C}_E\ \rho\ (e\ \textbf{::}\ \tau) & = & \mathcal{C}_E\ \rho\ e \\
 & & \\
 & & \\
\mathcal{C}_{Case}\ \rho\ \langle\ r_1,\ldots,r_m\ \rangle & = & \textbf{let}\ n = 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\ (v=e) & = & \mathcal{C}_E\ \rho\ e +\!\!+ \langle\ \text{MakeVar}\ n\ \rangle
\end{array}$$

Fig. 15. Compilation Rules for Expressions (2)

case of box declarations, it is also necessary to copy inputs to the stack using **CopyInput** instructions and to deal with fair matching.

Box bodies are compiled using $\mathcal{C}_R/\mathcal{C}_{R'}$ (Figure 17). These rules compile matches for the outer level patterns using $\mathcal{C}_P$, then compile inner pattern matches using $\mathcal{C}_A$, before introducing **Consume** instructions for non-***** input positions. The RHS can now be compiled. If more than one result is to be produced, the tuple of outputs is unpacked onto the stack. A **CheckOutputs** is inserted to verify that the outputs can be written using appropriate **Write** instructions. Finally, a **Reorder** is inserted if needed to deal with fair matching, and a **Schedule** returns control to the scheduler. The compilation of function/handler bodies using $\mathcal{C}_F/\mathcal{C}_{F'}$ is similar, except that $\mathcal{C}_{P'}$ is used rather than $\mathcal{C}_P$, there is no need

$$\begin{array}{l}
\mathcal{C}_D\ \rho\ (\textbf{box}\ b\ \mathit{ins}\ \mathit{outs}\ \textbf{fair}\ \mathit{rs}\ \textbf{handle}\ \mathit{xs}) \ = \ \mathcal{C}_B\ \rho\ \mathit{true}\ b\ \mathit{ins}\ \mathit{outs}\ \mathit{rs} \\
\mathcal{C}_D\ \rho\ (\textbf{box}\ b\ \mathit{ins}\ \mathit{outs}\ \textbf{unfair}\ \mathit{rs}\ \textbf{handle}\ \mathit{xs}) \ = \ \mathcal{C}_B\ \rho\ \mathit{false}\ b\ \mathit{ins}\ \mathit{outs}\ \mathit{rs} \\
\\
\mathcal{C}_D\ \rho\ (v\ =\ e) \ = \ \langle\ \text{Label}\ v,\ \text{CreateFrame}\ 0\ \rangle \mathbin{+\!\!+} \mathcal{C}_E\ \rho\ e \mathbin{+\!\!+} \langle\ \text{Return}\ \rangle \\
\\
\mathcal{C}_D\ \rho\ (f\ =\ p_1\ \rightarrow\ e_1\ ;\ \ldots\ p_n\ ;\ \rightarrow\ e_n) = \\
\qquad \textbf{let}\ n\ =\ \mathit{maxVars}\ \langle\ p_1,\ \ldots,\ p_n\ \rangle\ \textbf{in} \\
\qquad \langle\ \text{Label}\ f,\ \text{CreateFrame}\ n\ \rangle \mathbin{+\!\!+} \\
\qquad \mathcal{C}_F\ \rho\ \langle\ p_1\ \rightarrow\ e_1\ \ldots\ p_n\ \rightarrow\ e_n\ \rangle \mathbin{+\!\!+} \\
\qquad \langle\ \text{Function}\ f\ (\mathit{labels}\ f)\ \rangle \\
\\
\mathcal{C}_D\ \rho\ (\textbf{wire}\ p_1\ \textbf{to}\ p_2) = \ \langle\ \text{Wire}\ \ldots\ \rangle \\
\\
\mathcal{C}_D\ \rho\ (\textbf{data}\ \ t\ v_1\ \ldots\ v_n\ ) = \ \langle\ \ \rangle \\
\\
\\
\mathcal{C}_B\ \rho\ f\ b\ (\mathit{in}_1, \ldots, \mathit{in}_i)\ (\mathit{out}_1, \ldots, \mathit{out}_m)\ \mathit{rs} \ = \\
\qquad \textbf{let}\ n\ =\ \mathit{maxVars}\ \mathit{rs}\ \textbf{in} \\
\qquad \langle\ \text{Label}\ b\ \rangle \mathbin{+\!\!+} \\
\qquad \langle\ \text{CopyInput}\ (i-1),\ \ldots,\ \text{CopyInput}\ 0\ \rangle \mathbin{+\!\!+} \\
\qquad \langle\ \text{Push}\ 2,\ \text{CreateFrame}\ n\ \rangle \mathbin{+\!\!+} \\
\qquad (\textbf{if}\ f\ \textbf{then}\ \langle\ \text{StartMatches}\ \rangle\ \textbf{else}\ \langle\ \rangle) \ \mathbin{+\!\!+} \mathcal{C}_R\ \rho\ f\ m\ \mathit{rs} \ \mathbin{+\!\!+} \\
\qquad \langle\ \text{Box}\ b\ \ldots\ \rangle
\end{array}$$

Fig. 16. Compilation Rules for Declarations and Box Bodies

to deal with box inputs/outputs or fair matching, and a **Return** rather than **Schedule** is inserted at the end of each compiled rule.

Finally patterns are compiled using $\mathcal{C}_P/\mathcal{C}_{P'}$ (Figure 18), 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 constructor is matched, and then if the match is successful, the matched object is deconstructed on the stack to allow its inner components to be matched against the inner patterns. These nested patterns are compiled using $\mathcal{C}_A$ and $\mathcal{C}_{A'}$. $\mathcal{C}_{A'}$ inserts either **CopyArg** and **Unpack** instructions to decompose function/box arguments, or **Copy** and **Unpack** instructions to deal with nested pattern matches, where it is only necessary to replicate arguments that are already in the local stack frame.

4.1 Compilation Example: the Vending Machine

Figure 19 shows the pHAM instructions for the vending machine example of Section 2.5. First, in the preamble to the box, the inputs are copied into the local heap using `CopyInput` instructions and a dummy stack frame is constructed to

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

Fig. 17. Compilation Rules for Rule Matches and Functions

$$\begin{array}{lcl}
\mathcal{C}_P\ (*) & = & \langle\ \text{MatchNone}\ \rangle \\
\mathcal{C}_P\ (p) & = & \langle\ \text{MatchAvailable}\ \rangle \mathbin{+\!\!+} \mathcal{C}_{P'}\ p \\
 & & \\
\mathcal{C}_{P'}\ (i) & = & \langle\ \text{MatchInt32 } i\ \rangle \\
\ldots & & \\
\mathcal{C}_{P'}\ (c\ p_1\ \ldots\ p_n) & = & \langle\ \text{MatchCon } c\ n\ \rangle \\
\mathcal{C}_{P'}\ (\ p_1\ \ldots\ p_n\ ) & = & \langle\ \text{MatchTuple } n\ \rangle \\
\mathcal{C}_{P'}\ \langle\langle\ p_1\ \ldots\ p_n\ \rangle\rangle & = & \langle\ \text{MatchVector } n\ \rangle \\
\mathcal{C}_{P'}\ (\mathit{var}) & = & \langle\ \text{MatchVar } \mathit{var}\ \rangle \\
\mathcal{C}_{P'}\ \_ & = & \langle\ \text{MatchAny}\ \rangle \\
 & & \\
\mathcal{C}_A\ (c\ p_1\ \ldots\ p_n) & = & \mathcal{C}_{A'}\ \text{CopyArg}\ \langle\ p_1, \ldots,\ p_n\ \rangle \\
\mathcal{C}_A\ (\ p_1,\ \ldots,\ p_n\ ) & = & \mathcal{C}_{A'}\ \text{CopyArg}\ \langle\ p_1, \ldots,\ p_n\ \rangle \\
\mathcal{C}_A\ (x\ p) & = & \mathcal{C}_{A'}\ \text{CopyArg}\ \langle\ p\ \rangle \\
\mathcal{C}_A\ p & = & \langle\ \rangle \\
 & & \\
\mathcal{C}_{A'}\ i\ \langle\ p_1, \ldots,\ p_n\ \rangle & = & \langle\ i, \text{Unpack}\ \rangle \mathbin{+\!\!+} \\
 & & \mathcal{C}_{A'}\ \text{Copy}\ p_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_{A'}\ \text{Copy}\ p_n \mathbin{+\!\!+} \\
 & & \mathcal{C}_{P'}\ p_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}_{P'}\ p_n
\end{array}$$

Fig. 18. Compilation Rules for Patterns

hold values of the matched variables. Each rule is then compiled separately. For illustration, we show instructions corresponding to the first and third rules only. The LHS of each rule is bracketed in a pair of `MatchRule`. . .`MatchedRule` instructions, where individual `MatchX` instructions perform the matching of inputs against patterns. `MatchAvailable` is used to ensure that the input is available before matching a concrete value or variable; `MatchNone` corresponds to a `*`-pattern, and is used to indicate that the particular input need not be matched. Any match failure transfers control to label corresponding to the next rule. Finally, the inputs that have been matched are consumed.

The expression corresponding to RHS of the rule is compiled in a fairly conventional manner with arguments pushed on the stack before each function call, and the stack cleaned up using a `Slide`)instruction and the result unpacked from a tuple onto the stack using an `Unpack` instruction.

Finally, it is necessary to check that the thread does not need to block (this is done using `CheckOutputs`); if not, then each of the results is written to the correct wire using a `Write` instruction, and finally control is returned to the scheduler using a `Schedule` instruction.

5 Modelling Space Costs

A major goal of our research is to provide good cost models for Hume programs. We have already defined a simple space cost model for FSM-Hume that predicts upper bound stack and heap space limits for the pHAM [19]. We will reprise and extend this model here to illustrate how Hume aids the construction of cost models for resource-bounded programming.

The stack and heap requirements for the boxes and wires represent the only dynamically variable memory requirements for the pHAM implementation: all other memory costs can be fixed at compile-time based on the number of wires, boxes, functions and the sizes of static strings. In the absence of recursion, we can provide precise static memory bounds on rule evaluation. Predicting the stack and heap requirements for an FSM-Hume program thus provides complete static information about system memory requirements.

5.1 Memory Costs in the pHAM

Figure 20 defines values in the pHAM for all the constants $\mathcal{H}_{int32}$ etc. used above. In the prototype implementation, all heap cells are *boxed* [39] with tags distinguishing different kinds of objects. Furthermore, tuple structures require *size* fields, and data constructors also require a *constructor tag* field. All data objects in a structure are referenced by pointer. For simplicity each field is constrained to occupy one word of memory. Clearly, it would be easy to considerably reduce heap usage using a more compact representation such as that used by the state-of-the-art STG-Machine [39]. For now, we are, however, primarily

```
Label "control"
CopyInput 2    # latch inputs
CopyInput 1
CopyInput 0
Push 3         # new frame
CreateFrame 1  # one matched var.

Label "control_0"                       Label "control_2"
MatchRule      # LHS of rule 1          MatchRule      # LHS of rule 3
MatchAvailable # coin = Nickel?         MatchNone      # match *
MatchCon "Nickel" 0                     MatchAvailable # button = Coffee?
MatchNone      # match *                MatchCon "BCoffee" 0
MatchAvailable # match v                MatchAvailable # match v
MatchVar 0                              MatchVar 0
Consume 0      # consume coin           Consume 1      # consume button
Consume 2      # consume value          Consume 2      # consume value
MatchedRule    # end of LHS             MatchedRule    # end of LHS

# add_value v 5                         # do_dispense Coffee 10 v
MkInt 5        # push 5                 PushVar 0      # push v
PushVar 0      # push v                 MkInt 10       # push 10
Call "f_add_value"                      MkCon "Coffee" 0 # push Coffee
Slide 2        # pop 2 args.            Call "f_do_dispense"
Unpack         # unpack result          Slide 3        # pop 3 args.
                                        Unpack         # unpack result

CheckOutputs   # OK to write?           CheckOutputs   # OK to write?
Write 0        # write dispense         Write 0        # write dispense
Write 1        # write value'           Write 1        # write value'
Write 2        # write return           Write 2        # write return
Schedule       # end of rule 1          Schedule       # end of rule 3 ...

...

...

Wire "control" 0 "cashbox" 0 3 0 NullT
Wire "control" 1 "keypad" 1 3 0 NullT
Wire "control" 2 "control" 1 2 0 NullT
Wire "vend" 0 "control" 0 3 0 NullT
Wire "cashbox" 1 "control" 2 2 0 NullT
```

Fig. 19. pHAM instructions for the Vending Machine Example

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 described here, except by reducing absolute costs of both model and implementation.

constant	value (words)
$\mathcal{H}_{con}$	3
$\mathcal{H}_{tuple}$	2
$\mathcal{H}_{bool}$	2
$\mathcal{H}_{char}$	2
$\mathcal{H}_{int32}$	2

constant	value (words)
$\mathcal{H}_{float32}$	2
$\mathcal{H}_{none}$	1
$\mathcal{H}_{exn}$	1
$\mathcal{H}_{fun}$	4
...	...
$\mathcal{S}_{frame}$	4

Fig. 20. Sizes from the prototype Hume Abstract Machine

Instruction	stack	heap
MkBool b	*1*	*2*
MkInt32 n	*1*	*2*
MkFloat32 f	*1*	*2*
MkNone	*1*	*1*
MkCon c n	*1*	$n+3$
MkTuple n	*1*	$n+2$
MkVector n	*1*	$n+2$
MkFun f m n	*1*	$m+4$
Push n	n	*0*
Pop n	$-n$	*0*
Slide n	$-n$	*0*
Copy n	*1*	*0*
CopyArg n	*1*	*0*
CreateFrame n	$n+3$	*0*
PushVar n	*1*	*0*
MakeVar n	*0*	*0*

Instruction	stack	heap
MatchRule	*0*	*0*
MatchAvailable	*0*	*0*
MatchNone	*0*	*0*
MatchBool	*0*	*0*
MatchInt32	*0*	*0*
MatchFloat32	*0*	*0*
MatchCon	*0*	*0*
MatchTuple	*0*	*0*
MatchVector	*0*	*0*
Unpack *n-tuple*	n	*0*
Unpack *n-vector*	n	*0*
Unpack *constr(c,n)*	n	*0*
StartMatches	*0*	*0*
Reorder	*0*	*0*
CopyInput n τ	*0*	*sizeof*(τ)
Consume	*0*	*0*
CheckOutputs	*0*	*0*
Write	*0*	*0*
Input	*1*	*2*
Output	*-1*	*0*
Schedule	*0*	*0*

Fig. 21. Memory Costs for each pHAM instruction (first-order constructs)

We can now define concrete costs for each of the first-order pHAM instructions from Section 3 as shown in Figure 21. This effectively gives a small-step operational semantics for cost based on the translation of FSM-Hume source programs to pHAM instructions that was defined earlier.

5.2 Space Cost Rules

Figures 22–23 give stack- and heap-space cost rules for a representative subset of FSM-Hume expressions, based on an operational interpretation of the Hume

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

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

...

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

$$(3)\quad \frac{\forall i.\ 1\leq i\leq n,\ \ \mathrm{E}\ \overset{\text{space}}{\vdash}\ \mathit{exp}_i\ \Rightarrow\ h_i,s_i}{\mathrm{E}\ \overset{\text{space}}{\vdash}\ \mathrm{conid}\ \mathit{exp}_1\ \ldots\ \mathit{exp}_n\ \Rightarrow\ \sum_{i=1}^{n} h_i+n+\mathcal{H}_{con},\ \underset{i=1}{\overset{n}{\mathit{max}}}\ (s_i+(i-1))}$$

$$(4)\quad \frac{\forall i.\ 1\leq i\leq n,\ \mathrm{E}\ \overset{\text{space}}{\vdash}\ \mathit{exp}_i\ \Rightarrow\ h_i,s_i}{\mathrm{E}\ \overset{\text{space}}{\vdash}\ (\ \mathit{exp}_1,\ \ldots,\ \mathit{exp}_n\ )\ \Rightarrow\ \sum_{i=1}^{n} h_i+n+\mathcal{H}_{tuple},\ \underset{i=1}{\overset{n}{\mathit{max}}}\ (s_i+(i-1))}$$

$$(5)\quad \frac{\forall i.\ 1\leq i\leq n,\ \mathrm{E}\ \overset{\text{space}}{\vdash}\ \mathit{exp}_i\ \Rightarrow\ h_i,s_i}{\mathrm{E}\ \overset{\text{space}}{\vdash}\ \langle\langle\ \mathit{exp}_1,\ \ldots,\ \mathit{exp}_n\ \rangle\rangle\ \Rightarrow\ \sum_{i=1}^{n} h_i+n+\mathcal{H}_{vector},\ \underset{i=1}{\overset{n}{\mathit{max}}}\ (s_i+(i-1))}$$

Fig. 22. Space cost axioms for expressions (1)

abstract machine implementation. Heap and stack costs are each integer values of type Cost, labelled h and s, respectively. Each rule produces a pair of such values representing an independent upper bound on the stack and heap usage. The result is produced in the context of an environment, E, that maps function names to the space (heap and stack) requirements associated with executing the body of the function. This environment is derived from the top-level program declarations plus standard prelude definitions. Rules for building the environment are omitted here, except for local declarations, but can be trivially constructed.

The heap cost of a standard integer is given by $\mathcal{H}_{int32}$ (rule 1), with other scalar values costed similarly. The cost of a function application is the cost of evaluating the body of the function plus the cost of each argument (rule 2). Each evaluated argument is pushed on the stack before the function is applied, and this must

$$(6)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp}_1 \Rightarrow h_1, s_1 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp}_2 \Rightarrow h_2, s_2 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp}_3 \Rightarrow h_3, s_3}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{if } \mathit{exp}_1 \texttt{ then } \mathit{exp}_2 \texttt{ else } \mathit{exp}_3 \Rightarrow h_1 + \mathit{max}(h_2, h_3), \mathit{max}(s_1, s_2, s_3)}$$

$$(7)\quad \frac{\mathrm{E} \overset{\text{decl}}{\vdash} \text{decls} \Rightarrow h_d, s_d, s'_d, \text{E'} \qquad \text{E'} \overset{\text{space}}{\vdash} \mathit{exp} \Rightarrow h_e, s_e}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{let } \text{decls} \texttt{ in } \mathit{exp} \Rightarrow h_d + h_e, \mathit{max}(s_d, s'_d + s_e)}$$

$$(8)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \Rightarrow h_e, s_e \qquad \mathrm{E} \models \mathit{matches} \Rightarrow h_m, s_m}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{case } \mathit{exp} \texttt{ of } \mathit{matches} \Rightarrow h_e + h_m, \mathit{max}(s_e, s_m)}$$

$$(9)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \Rightarrow h_e, s_e}{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \texttt{ within } h, s \Rightarrow \mathit{min}(h_e, h), \mathit{min}(s_e, s)}$$

$$(10)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \Rightarrow h_e, s_e \qquad \mathit{coerce}\ \tau \;=\; (h_c, s_c)}{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \texttt{ as } \tau \Rightarrow h_e + h_c, \mathit{max}(s_e, s_c)}$$

$$(11)\quad \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \Rightarrow h_e, s_e}{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp} \texttt{ :: } \tau \Rightarrow h_e, s_e)}$$

Fig. 23. Space cost axioms for expressions (2)

be taken into account when calculating the maximum stack usage. The cost of building a new data constructor value such as a user-defined constructed type (rule 3) is similar to a function application, except that pointers to the arguments must be stored in the newly created closure (one word per argument), and fixed costs $\mathcal{H}_{con}$ are added to represent the costs of tag and size fields. Tuples and vectors are costed analogously to constructors (rules 4 and 5).

The heap usage of a conditional (rule 6) is the heap required by the condition part plus the maximum heap used by either branch. The maximum stack requirement is simply the maximum required by the condition and either branch. Case expressions (omitted) are costed analogously. The cost of a let-expression (rule 7) is the space required to evaluate the value definitions (including the stack required to store the result of each new value definition) plus the cost of the enclosed expression. The local declarations are used to derive a quadruple comprising total heap usage, maximum stack required to evaluate any value definition, a count of the value definitions in the declaration sequence (used to calculate the size of the stack frame for the local declaratons), and an environment mapping function names to heap and stack usage. The body of the let-expression is costed in the context of this extended environment. The cost of a case=expression (rule 8) is derived from the maximum heap used by any RHS or the matched expression e, and the greatest stack used by any individual

```
Label "f_do_dispense"
CreateFrame 3    # 3 local vars.

Label "f_do_dispense_0"           Label "t"
MatchRule                         MkNone          # *
MatchVar 0      # bind drink      PushVar 1       # cost
MatchVar 1      # bind cost       PushVar 2       # v
MatchVar 2      # bind v          CallPrim "-"    # v-cost
MatchedRule                       PushVar 0       # drink
                                  MkTuple 3       # make result
PushVar 1       # cost            Return
PushVar 2       # v
CallPrim ">="   # v >= cost?
If "t"          # branch if so

MkNone          # *
PushVar 2       # v
MkNone          # *
MkTuple 3       # (*,v,*)
Return

Function "f_do_dispense" "f_do_dispense_0"
```

Fig. 24. pHAM instructions for the do_dispense function

match or the matched expression e. Note that in calculating the stack usage, it is necessary to account both for the stack frame which is needed to record the local variable definitions in each match, and for any structured values which are unpacked onto the stack during matching.

Finally, the rule for within-expressions (rule 9) restricts the costs to the smaller of the space that is estimated to be used by the sub-expression or the specified restriction; as-expressions must account for any space used by the coercion (rule 10); but type restrictions have no cost component (rule 11).

5.3 Costing Example: the Vending Machine

We illustrate the cost analysis by showing how stack and heap limits can be derived structurally for the `do_dispense` function taken from the vending machine example.

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

Figure 24 shows the pHAM bytecode for this function, which follows the translation of Section 4. The costs for each expression are shown below, where $e : \ s/h$

is the relative change in stack (s) and heap (h) incurred by evaluating expression e:

$$\dfrac{\dfrac{v:\ 1/0 \quad \mathit{cost}:\ 1/0}{v \geq \mathit{cost}:\ 2/2} \quad \dfrac{*:\ 1/1 \quad \dfrac{v:\ 1/0 \quad \mathit{cost}:\ 1/0}{v - \mathit{cost}:\ 2/2} \quad \mathit{drink}:\ 1/0}{(\mathit{drink}, v - \mathit{cost}, *):\ 3/8} \quad (*, v, *):\ 3/7}{\mathit{if}\ v \geq \mathit{cost}\ \mathit{then}\ (\mathit{drink}, v - \mathit{cost}, *)\ \mathit{else}\ (*, v, *):\ \mathit{max}(3,3)/2 + \mathit{max}(7,8)}$$

```
$ phamc coffee.hume
Prototype Hume Abstract Machine Compiler 0.8t
importing coffee.hume
Function costs:

do_dispense: stack = 16; heap = 10
add_value: stack = 23; heap = 13
...

Box costs:
Box control: stack = 31, heap = 23
...
```

Results from a sample execution of the program on a 1.67GHz PowerPC G4 (Apple Macintosh Powerbook) show that stack and heap costs are consistent with the limits given by the analysis: 26 words of stack used versus a limit of 31 word; and 20 words of heap versus a limit of 23 words. The differences are due to the need to reserve heap memory for all wires in calculating an upper bound limit for heap usage, even though, in the actual execution path, only one of `coin` or `button` is active at any time; and to the existence of program paths that require additional stack, but which are not explored in the test examples.

The `control` box takes a maximum of 630μs to execute, with maximum response time on any of the control wires being 0.186ms. These figures are consistent with genuine real-time requirements.

```
Box Statistics:

control: CALLS = 14, MAXTIME = 0.00063s, MAXHP = 20, MAXSP = 26
...

Wire Statistics:

control.0: MAX DELAY = 0.045ms, MAXHP = 3
control.1: MAX DELAY = 0.082ms, MAXHP = 3
control.2: MAX DELAY = 0.186ms, MAXHP = 2
```

```
Memory usage for boxes:

control: Heap 20(23) Stack 26(31) -- wires Heap 8(8) Stack 0(0)
...

Total heap usage: 72 (87 est)
Total stack usage: 49 (57 est)
```

6 Other Related Work

As discussed in Section 1.2, because of the difficulty of constructing strong bounded cost models, functional languages have not historically been applied to hard real-time systems. Hume is therefore highly novel in its approach. There has, however, been much recent theoretical interest both in the problems associated with costing functional languages [41, 25, 12, 46, 36] and in bounding space/time usage [24, 45, 21], including work on automatically generating heap bounded functional programs [44].

Synchronous dataflow languages such as Lustre [13] or Signal [17] have been widely used for programming hard real-time systems. Like Hume, such designs tend to separate computation from communication. Compared with Hume, however, such notations tend to lack expressibility. For example, they will usually provide only first order functions and flat data structures, compared with the higher-order functions and recursive data structures that Hume supports. They also generally eschew constructs such as exception handling, and are also, of course, restricted to synchronous communication rather than the asynchronous approach described here. The advantage of these notations lies in providing a powerful and simple model of time costs, by eliminating the timing complexities associated with asynchronicity. One interesting new design that attempts to marry the advantages of synchronous and asynchronous languages is Lucid Synchrone [15]. This language combines both synchronous and asynchronous communication within a single language framework.

Cost issues are relevant not only for programming real-time systems, but also for producing hardware designs, or even hardware/software co-designs. There have been several functionally-based notations for hardware design [20, 14, 26, 34, 31], similar in scope to the HW-Hume level described here, but usually including hardware-specific issues such as on-chip circuit layout.

Finally, there has, of course, been much work on applying conventional language technology to the problems of hard real-time, bounded space programming. Amongst recent language designs, two extreme approaches are SPARK Ada [7] and the real-time specification for Java (RTSJ) [9]. SPARK Ada aims to ensure strong formal properties by eliminating difficult-to-model behaviours from the general-purpose language Ada. However, this excludes highly desirable features such as concurrency and asynchronous communication.

A contrasting approach is taken by RTSj, which provides specialised runtime and library support for real-time systems work, including manually controlled region-based dynamic memory allocation, but which makes no absolute performance guarantees. One major limitation of the approach is that standard Java libraries cannot normally be used, since the usual object allocation and garbage collection mechanisms are not supported. Thus, SPARK Ada provides a minimal, highly controlled environment for real-time programming emphasising *correctness by construction* [1], whilst Real-Time Java provides a much more expressible, but less controlled environment, without formal guarantees. Our objective with the Hume design is to maintain strong formal correctness whilst also providing high levels of expressibility that match the demands of real-time systems programming.

7 Expressibility versus Costability

In Section 5 we have shown that formal space cost models can be easily constructed for FSM-Hume. Although we have not yet proven the formal correctness of these models, we have verified the accuracy of these cost models against the pHAM implementation and shown that such models have practical application in the pHAM compiler. Moreover the pHAM implementation technology is competitive in both time and space usage with similar technologies proposed for embedded Java applications, for example. We anticipate that it should be possible to construct equally usable time-cost models using a similar approach, but that these must be tailored to specific processor architectures in order to obtain usable real-time guarantees.

We have obtained accurate cost models by sacrificing expressibility: FSM-Hume is devoid of recursive or higher-order functions and possesses only non-recursive data structure such as tuples. Features such as asynchronous concurrency and box iteration do compensate to some extent and will allow the construction of moderately complex programs. Our more theoretical work [41] suggests that we should be able to incorporate higher-order functions into FSM-Hume (to form Template-Hume) while still being able to construct good quality cost models. We have also constructed prototype cost models that include automatic cost analysis of primitive recursive function definitions, and have obtained results that are equivalent to hand analysis for some simple recursive functions [47]. Our hope is that this work will allow us to construct PR-Hume, incorporating primitive recursive functions and a range of recursive data structures such as lists. Such a result would show that it is possible to combine both a very high level of expressibility and accurate cost models.

8 Conclusions and Further Work

This paper has introduced Hume, a concurrent functionally-based language aimed at resource-limited systems such as real-time embedded systems. Hume is novel in being built on a combination of finite-state-machine and λ-calculus concepts. It is also novel in aiming to provide a high level of programming abstraction whilst maintaining good formal properties, including bounded time and space behaviour and provably correct rule-based translation. We achieve this by synthesising recent advances in theoretical computer science into a coherent pragmatic framework. In this paper, we have formally described the Hume abstract machine implementation, shown how this is related to Hume source programs through a formal set of translation rules, and finally shown how a source-level cost model can consequently be constructed for Hume. We believe that this work helps to open up the in-principle use of functional programming techniques in resource-constrained situations.

While we have not yet optimised our implementations, Hume has been designed to allow good compiler optimisations to be exploited – time performance *without* optimisation is roughly 10 times that for Sun's embedded KVM Java Virtual Machine or about 50% of that of optimised C++, and dynamic space usage is both guaranteed to be bounded and a fraction of that required by either Java or C++. For example, we have constructed a complete implementation for a Renesas M32C bare development board using less than 16KB RAM, including all runtime, operating system, user code and library support.

A number of important limitations remain to be addressed:

1. space and time cost models must be defined for additional Hume levels and language features, including higher-order functions (Template-Hume) and (primitive) recursion (PR-Hume);
2. these cost models must be used to construct high-quality static analyses;
3. we must incorporate interrupt handling and some other low-level features into our design and implementation;
4. more sophisticated scheduling algorithms could improve performance, however, these must be balanced with the need to maintain correctness; and finally
5. no attempt is made to avoid deadlock situations: a suitable model checker must be desgned and implemented.

Acknowledgements

This work has been generously supported by EU Framework VI grant IST-2004-510255 (EmBounded) and by EPSRC Grant EPC/0001346.

References

1. P. Amey. Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, pages 24–28, March 2002.
2. K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer Verlag, 1997. 2nd Edition.
3. J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *Proc. INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.
4. J. Armstrong. The Development of Erlang. In *Proc. 1997 ACM Intl. Conf. on Functl. Prog. (ICFP '97)*, pages 196–203, Amsterdam, The Netherlands, 1997.
5. J. Armstrong, S.R. Virding, and M.C. Williams. *Concurrent Programming in Erlang.* Prentice-Hall, 1993.
6. L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, 1987.
7. J.G.P. Barnes. *High Integrity Ada: the Spark Approach.* Addison-Wesley, 1997.
8. S. Blau and J. Rooth. AXD-301: a New Generation ATM Switching System. *Ericsson Review*, 1, 1998.
9. G. Bollela and et al. *The Real-Time Specification for Java.* Addison-Wesley, 2000.
10. F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.
11. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pe na. The Eden Coordination Model for Distributed Memory Systems. In *Proc. High-Level Parallel Programming Models and Supportive Environments (HIPS '97), Springer-Verlag LNCS 1123.* Springer-Verlag, 1997.
12. R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, Dept. of Comp. Sci., Univ. of Edinburgh, 1987.
13. P. Caspi, D. Pilaud, N. Halbwachs, and J. Place. Lustre: a Declarative Language for Programming Synchronous Systems. In *Proc. POPL '87 – 1987 Symposium on Principles of Programming Languages, München, Germany*, pages 178–88, January 1987.
14. K. Claessen and M. Sheeran. A Tutorial on Lava: a Hardware Description and Verification System, unpublished report, chalmers university of technology, sweden. August 2000.
15. J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *Proc. ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey City, New Jersey, USA, September 2005.
16. D.H. Fijma and R.T. Udink. A Case Study in Functional Real-Time Programming. Technical report, Dept. of Computer Science, Univ. of Twente, The Netherlands, 1991. Memoranda Informatica 91-62.
17. T. Gautier, P. Le Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Proc. Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA '87), Springer-Verlag LNCS 274*, pages 257–277, 1987.
18. K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. 2003 Conf. on Generative Programming and Component Engineering (GPCE '03), Springer-Verlag LNCS 2830*, pages 37–56, 2003.
19. K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

20. J. Hawkins and A.E. Abdallah. Behavioural Synthesis of a Parallel Hardware JPEG Decoder from a Functional Specification. In *Proc. EuroPar 2002, Paderborn, Germany*, pages 615–619. Springer-Verlag LNCS 2400, August 2002.
21. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, LA, USA, January 2003. ACM Press.
22. R.J.M. Hughes. The Design and Implementation of Programming Languages, DPhil Thesis, Programming Research Group, Oxford. July 1983.
23. R.J.M. Hughes and J.T. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, pages 308–328, London, UK, 1990. Springer-Verlag.
24. R.J.M. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: towards Embedded ML Programming. In *ICFP'99 — International Conference on Functional Programming*, pages 70–81, Paris, France, September 1999. ACM Press.
25. R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL'96 — Symposium on Principles of Programming Languages*, pages 410–423, St. Petersburg Beach, Florida, January 1996. ACM.
26. J. Launchbury J. Matthews and B. Cook. Microprocessor Specification in Hawk. In *Proc. International Conference on Computer Languages*, pages 90–101, 1998.
27. P. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, Jan 1964.
28. S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In ACM, editor, *Proc. POPL '95 — 1995 Symposium on Principles of Programming Languages: San Francisco, California*, pages 333–343, New York, NY, USA, 1995. ACM Press.
29. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, April 1999.
30. J. McDermid. *Engineering Safety-Critical Systems*, pages 217–245. Cambridge University Press, 1996.
31. A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS '01)*, pages 236–251, 2001.
32. E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *Proc. Parallel Architectures and Languages Europe (PARLE '91)*, pages 202–219. Springer-Verlag LNCS 505, 1991.
33. J. Nordlander, M. Carlsson, and M. Jones. Programming with Time-Constrained Reactions (unpublished report). `http://www.cse.ogi.edu/pacsoft/projects/Timber/publications.htm`. 2006.
34. J.T. O'Donnell. The Hydra Hardware Description Language. In *Domain-Specific Program Generation*. Spinger-Verlag LNCS 3016, 2004.
35. R. Peña and C. Segura. A polynomial cost non-determinism analysis. In *Proc. IFL'01 – 13th International Workshop on Implementation of Functional Languages, Spinger-Verlag LNCS 2312*, pages 121–137, 2002.
36. R. Peña and C. Segura. A First-Order Functl. Lang. for Reasoning about Heap Consumption. In *Draft Proc. International Workshop on Implementation and Application of Functional Languages (IFL '04)*, 2004.
37. J. Peterson, P. Hudak, and C. Elliot. Lambda in Motion: Controlling Robots with Haskell. In *Proc. ACM Conference on Practical Applications of Declarative Languages (PADL '99)*, September 1999.

38. S. L. Peyton Jones and D. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall, 1992.
39. S.L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware:the Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
40. S.L. Peyton Jones, A.D. Gordon, and S.O. Finne. Concurrent Haskell. In *Proc. POPL'96 — ACM Symp. on Principles of Programming Languages*, pages 295–308, January 1996.
41. A.J. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. A Sized Time System for a Parallel Functional Language (Revised). In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, pages 1–16. Springer-Verlag LNCS 2670, 2003.
42. J.H. Reppy. CML: a Higher-Order Concurrent Language. In *Proc. 1991 ACM Conf. on Prog. Lang. Design and Impl. (PLDI '91)*, pages 293–305, June 1991.
43. R. Schwarz. Causality in distributed systems. In *Proc. EW 5: 5th ACM SIGOPS European workshop*, pages 1–5, New York, NY, USA, 1992. ACM Press.
44. W. Taha, S. Ellner, and H. Xi. Generating Heap-Bounded Programs in a Functional Setting. In *Proc. ACM International Conference on Embedded Software (EMSOFT '03)*, pages 340–355, 2003.
45. M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1 February 1997.
46. D.A. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Functl. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, December 1995.
47. P.B. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, 2006. in preparation.
48. W.W. Wadge and E.A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press, 1985.
49. M. Wallace and C. Runciman. Extending a Functional Programming System for Embedded Applications. *Software: Practice & Experience*, 25(1), January 1995.
50. C. Wikström and H. Nilsson. Mnesia — an Industrial Database with Transactions, Distribution and a Logical Query Language. In *Proc. 1996 International Symposium on Cooperative Database Systems for Advanced Applications*, 1996.