Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach

KEVIN HAMMOND
University of St Andrews
GREG J. MICHAELSON
Heriot-Watt University
and
PEDRO B. VASCONCELOS
Universidade do Porto

Software engineering involves matching abstract software requirements to concrete implementations. Programming at a high-level of abstraction improves confidence in the correctness of functional requirements and reduces the cost of constructing software, but loses confidence in the correctness of behavioural requirements. This is especially serious for resource-constrained systems, such as real-time embedded or control systems, where tight runtime bounds must be maintained on both space and time behaviour.

This paper describes Hume: a novel programming language based on a combination of finite state machine and recursive programming constructs. Hume is unusual in being structured as a series of levels, each of which exposes different machine properties. The paper provides a series of determinate cost models for the different Hume levels, and shows how Hume programs can be refactored to lower levels. In this way, appropriate programming abstractions can be chosen, without compromising on program properties such as costability.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: —*Real-time and embedded systems*; C.4 [**Performance of Systems**]: —*Modeling techniques*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*; *Formal methods*; *Reliability*; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Recursion*; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; *Concurrent, distributed, and parallel languages*; *Specialized application languages*; D.3.4 [**Programming**

This work is generously supported by grants GR/R 70545 and EP/C 001346/1 from the UK's Engineering and Physical Sciences Research Council and by EU Framework VI grant IST-510255 (EmBounded).
We would like to thank Steffen Jost for his comments on an earlier draft of this paper.
Authors' addresses: K. 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`; G.J. Michaelson, School of Mathematics and Computer Science, Heriot-Watt University, Edinburgh, Scotland. `Email: greg@macs.hw.ac.uk`, `Tel: +44-131-451-3422`, `Fax: +44-131-451-3327`; P.B. Vasconcelos, Departamento de Ciência de Computadores, Universidade do Porto, Porto, Portugal. `Email: pbv@dcc.fc.up.pt`.

Languages]: Processors—*Memory management (garbage collection)*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*; *Program analysis*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Control primitives*; *Functional constructs*; *Program and recursion schemes*

General Terms: Languages

Additional Key Words and Phrases: Resource-bounding

1. INTRODUCTION

Programming language design invariably maintains a tension between the level of programming abstraction and the ease with which language constructs can be related to their hardware implementation. More abstraction allows greater intrinsic software complexity to be handled with the same programming effort, often through reductions in code size[1]. Alternatively, less effort may be expended in dealing with the same intrinsic programming complexity. Where the reduction in effort involves a smaller software development team, productivity gains may even be super-linear[2] due to reduced communication and other overheads. However, high levels of programming abstraction create a large *semantic gap* between language and machine. This gap is normally observed either as poor performance and/or as difficulty in reasoning about actual machine behaviour from the program source.

In this paper, we consider how careful language design can help bridge this semantic gap in the domain of resource-bounded systems, for example in real-time embedded systems, a domain where performance is crucial and transparency is be essential, and so where the semantic gap is larger than in most general purpose settings. We contend that bridging this gap is vital if we are to reduce the costs of developing embedded software, to simultaneously reduce times to market and to allow the development of the more complex software that the market demands without losing important behavioural properties. In particular, we show how software may be constructed at an appropriate level of abstraction for the problem to be solved, yet whilst maintaining strong formal bounds on space usage, as is required for resource-constrained systems. We illustrate our approach with reference to a simple, but representative, example, the logic for a traffic light controller.

1.1 Essential Language Properties in the Real-Time Embedded System Domain

It is possible to identify a number of essential or desirable properties for a language that is aimed at real-time embedded systems [McDermid 1996].

—*determinacy* – the language should allow the construction of determinate systems, so that under identical environmental constraints, all executions of the system should be *observationally equivalent*;

[1]For example, Dazzle comprises 10,000 lines of Haskell code, where the functionally similar Elvira system comprises 220,000 lines of Java [Schrage et al. 2005]; and Wiger reports code size reductions of 4-10 when using Erlang rather than C++ [Wiger 2001].

[2]For example, Ericsson successfully replaced a 1000-strong team of C++ programmers with 6 Erlang programmers [Armstrong 2005].

	Java	C/C++	RTSj	Spark Ada	Esterel	Lustre	Erlang	Hume
general-purpose	*y*	*y*	*n*	*n*	*n*	*n*	*y*	*n*
determinacy	n^a	n^a	?	*y*	*y*	*y*	*y*	*y*
hard real-time	*n*	n^a	*y*	*y*	*y*	*y*	*n*	*y*
boundedness	*n*	n^a	*y*?	*y*	*y*	*y*	*n*	*y*
asynchronicity	*y*	n^b	*y*?	*y*	*y*	*n*	*y*	*y*
concurrency	*y*	n^b	*y*?	*y*	*y*	*n*	*y*	*y*
functional correctness	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*y*

a. possible by restricting use of language features.

b. not part of language design, available through (non-portable) libraries.

Fig. 1. Suitability of Languages for Real-Time Embedded Systems

—*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;

—*functional correctness* – the language must support a high degree of confidence that constructed systems meet their formal requirements [Amey 2002].

Adapting general purpose languages for the real-time embedded systems domain often leads to a poor fit between the language features and the implementation requirements (as seen in Figure 1). *Domain-specific* language designs such as Hume, however, allow low-level system requirements to guide the design of the high-level language features we desire.

1.2 Overview of the Hume Design

Our goal with the Hume design is to allow as high a level of program abstraction as possible whilst still maintaining the properties described above. For example, Hume provides exception handling, automatic memory management, higher-order functions, polymorphism and recursion whilst maintaining transparent time and space costing together with bounded time and space usage.

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 [Bollela and et al. 2000]), our approach is to design Hume in such a way that we are certain that formal models and proofs can be constructed. We identify a series of overlapping Hume language levels shown in Figure 2, where each level adds expressibility to the expression semantics, but either loses some desirable behavioural property or increases the technical difficulty of providing formal correctness/cost models[3].

[3]In earlier papers, we have identified a level termed HO-Hume, providing a fixed repertoire of com-

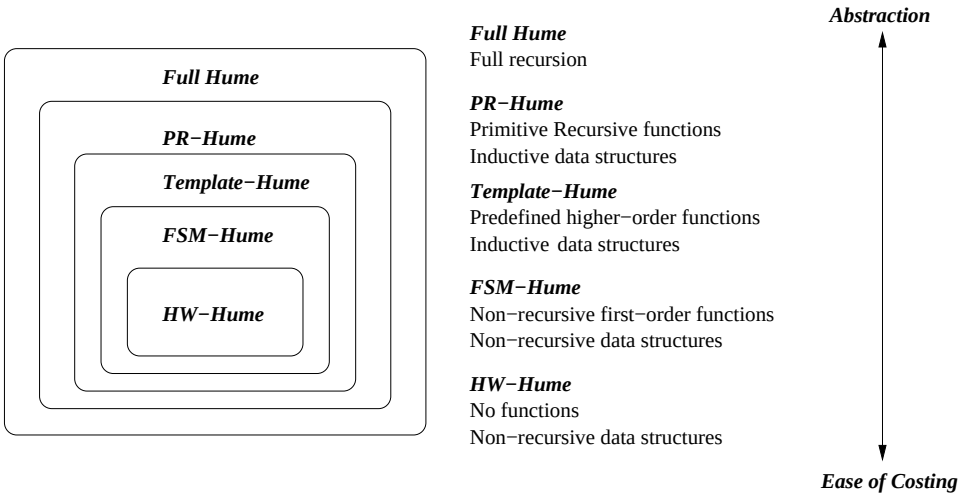

Fig. 2. Hume Levels

HW-Hume:. a hardware description language — capable of describing both synchronous and asynchronous hardware circuits, with pattern matching on tuples of bits, but with no other data types or operations;

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

Template-Hume:. a language for template-based programmimng — FSM-Hume plus predefined higher-order functions, polymorphism and inductive data structures, but no user-defined higher-order functions or recursive function definitions;

PR-Hume:. a language with decidable termination but restricted recursion — Template-Hume plus user-defined primitive recursive and higher-order functions;

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

We will expand on the formal properties that can be associated with each of these levels in Section 2. We first consider how they can be justified from a language design perspective.

1.3 Generalising finite state machines

When designing a general-purpose programming language, it is usual to aim for Turing Completeness, i.e. a language that is capable of expressing any computable function. Unfortunately, it is not possible to provide strong formal properties of program equivalence or termination for such languages. In our quest for a language with predictable time and space behaviour, we have therefore started at the other end of the programming language spectrum and revisited finite state machines (FSMs). FSMs are the simplest of computing devices with the most limited of

posable higher-order functions, midway between FSM-Hume and PR-Hume, rather than Template-Hume as here. HO-Hume can be justified in terms of introducing interesting program properties for analysis, but provides little increased functionality over Template-Hume.

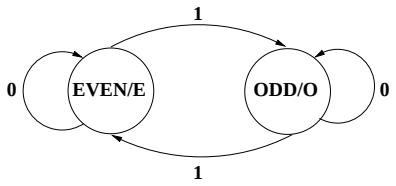

Fig. 3. Parity checking Moore machine.

expressive power in terms of the things they can compute. However, termination and equivalence *are* both decidable for FSMs.

A pure FSM consists of a number of states linked by arcs and has a single source of input symbols. Arcs are marked by symbols. If a FSM is in some state and one of the arcs from that state is marked with the next input symbol, then that input symbol is consumed and the state at the other end of the arc is entered. In this purist form, a FSM generates no output. A Moore machine [Hopcroft and Ullman 1979] is a FSM that may generate an output when a state is entered.

Simple example: a Moore machine for parity checking. For example, Figure 3 shows a Moore machine for checking the parity of a sequence of binary bits. Entering state `EVEN`, it outputs "E" With an input of 0 it stays in state `EVEN` and with an input of 1 it enters state `ODD`. Entering state `ODD`, it outputs "O". With an input of 0 it stays in state `ODD` and with an input of 1 it enters state `EVEN`. We can characterise the operation of a Moore FSM as a series of rules of the form:

$$(OldState, Input) \rightarrow (NewState, Output)$$

so the parity machine has rules:

```
(EVEN,0) -> (EVEN,E)
(EVEN,1) -> (ODD,O)
(ODD,0)  -> (ODD,O)
(ODD,1)  -> (EVEN,E)
```

For example, starting in state `EVEN` with inputs 1, 0 and 1, the machine goes through the sequence:

Old State	Input	New State	Output
EVEN	1	ODD	O
ODD	0	ODD	O
ODD	1	EVEN	E

The fundamental property of FSMs is that they have no memory other than that implicit in the current state: unlike Turing Machines, FSMs can only consume input but cannot store any intermediate information. A Turing Machine records its decisions on its tape, which, for a parity checker, could, in principle, be reverse engineered to give the original bit sequence. In contrast, the parity-checking FSM has no memory of its prior sequence of behaviours.

Extending FSMs to the Hume notation. From our interest in functional languages, we were struck by two correspondences. Firstly, the left hand sides of FSM rules are similar to *patterns* that have to be matched against the current state and input. Secondly, the right-hand-sides of FSM rules are similar to *tuples* of expressions that generate the next state and the output. Thus, we can generalise FSM rules to:

pattern → *function(pattern)*

such that the result of matching an input *pattern* is some *function* of that *pattern*. The effect of this generalisation is to view a FSM as an entity with a number of inputs — each matched by a *pattern* — and a number of outputs — each generated by the corresponding *function* of the inputs. A natural extension is to describe systems of linked FSMs with wires joining inputs to outputs, that is to enable coordinated, concurrent FSMs.

Thus, a Hume program consists of a number of autonomous but interconnected *boxes*, linked by *wires*. Each box's behaviour is described by a sequence of *match rules* associating input patterns with output tuples. The form of the input patterns depends on the types of the input wires. The output tuples may contain tuples written in the Hume *expression layer*, a purely functional notation. Simple constructs (described in Section 2) enable an input to be ignored or an output to be skipped, thereby facilitating asynchronicity.

Hume uses a concurrent execution model, with individual *microthreads* [Papadopoulos et al. 1993] used to process each iteration of a box. A microthread is scheduled if the required wire inputs to a box become available either as the result of input or through scheduling some other microthread. It then runs to completion as follows. Firstly, the available inputs are matched against each of the rule patterns, and the first matching rule selected for execution. Subsequently, the right-hand-side of the rule is evaluated and the resulting values placed on the outputs. If one or more outputs from a previous step have not yet been consumed then the box *blocks* until the output is consumed. Otherwise the microthread completes and a new microthread may be scheduled in the next cycle. This process continues repeatedly and indefinitely.

Example: a Hume implementation of the parity checker. A full description of the parity checker in Hume is:

```
data STATE = ODD | EVEN;
type Bit = word 1;

box parity
in  (s:: STATE, next::Bit)
out (s'::STATE, outp::string)
match
  (EVEN,0) -> (EVEN,"E\n")
| (EVEN,1) -> (ODD,"O\n")
| (ODD,0)  -> (ODD,"O\n")
| (ODD,1)  -> (EVEN,"E\n");
```

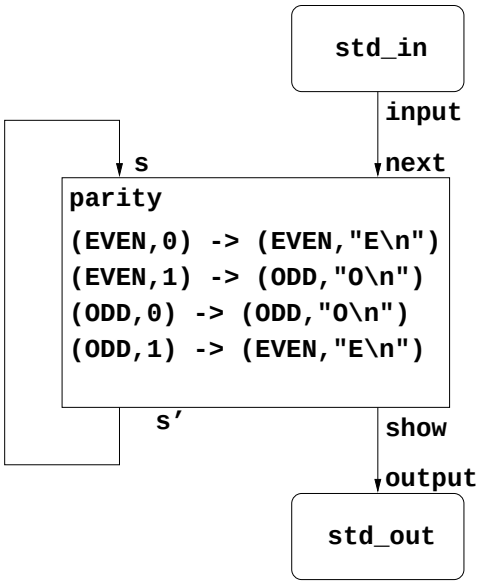

Fig. 4. Parity-checking Hume program.

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

wire input       to parity.next;
wire parity.s'   to parity.s initially EVEN;
wire parity.outp to output;
```

First we define two *streams* linked to the standard input and output, and a data type `STATE` to represent box states with values `ODD` and `EVEN`. Next, we specify that a `parity` box has two inputs, the current state `s`, and `next`, a 1 bit word. It also has outputs `s'`, the new state, and `out`, a string. The transition rules are similar to those for the FSM above. Finally, we specify the wiring for the box. The input `s` is wired to the output `s'`, the input `next` is wired from the stream `input`, and the output `out` is wired to the stream `output`. The input `s` is initially set to the state `EVEN`. Figure 4 shows a pictorial representation of this box.

2. PREDICTABLE BEHAVIOUR FOR HUME LEVELS

We wish to be able to determine the time and space behaviours of Hume programs. If we know the time and space behaviour of all the rules that define a box, then we can employ standard techniques and tools, such as coloured Petri Net models [Jensen 1989], to analyse static networks of boxes. However, full Hume is Turing-complete, and we are thus unable to statically determine time and space costs (or even termination) for arbitrary Hume programs. Rather than imposing unwieldy syntactic and type restrictions in order to ensure that costs can be determined (for example, by syntactically enforcing primitive recursion or *shapeliness* [Jay 2005]), we prefer to develop a set of tools that can check whether or

program ::=	$decl_1$; ... ; $decl_n$	$n \geq 1$
decl ::=	*box* | *wire* | *type*	
box ::=	**box** boxid *ins outs* **match** *matches*	
ins/outs ::=	(ioid_1 : τ_1 , ... , ioid_n : τ_n)	
τ ::=	$\tau_1 \times \ldots \times \tau_m$ | **vector** n **of** τ | **word** 1 | typeid	$m, n \geq 1$
matches ::=	$match_1$ | ... | $match_n$	$n \geq 1$
match ::=	(pat_1 , ... , pat_n) → *expr*	
expr/pat ::=	0 | 1 | varid | * | ($expr_1/pat_1$, ... , $expr_n/pat_n$)	$n \geq 2$
	| **vector** $expr_1/pat_1$ ... $expr_n/pat_n$	$n \geq 1$
wire ::=	**wire** $link_1$ **to** $link_2$ [**initially** *expr*]	
link ::=	boxid . ioid | deviceid	
type ::=	**type** typeid = τ	

Fig. 5. HW-Hume Syntax

not particular properties hold. These will offer various strengths of constraint on Hume through restrictions on the types of inputs and outputs, and on the forms of expressions that may be used on the right hand side of a match.

2.1 Non-Recursive Data Types: HW-Hume

At the strongest, if we insist that all wires can only carry single bits, and that only individual bit values may be generated on outputs, then we have classical binary logic circuits with determinate time and space behaviour. We can allow fixed-size product types on wires, again without loss of properties. Such types include vectors of objects with the same type, or n-ary tuples of objects with different types. These types simply provide a convenient notation for treating groups of wires as unitary objects. We may allow construction of such objects in the right hand of matches, and their deconstruction on the left hand side. We also allow variables to match any group of one or more bits on the LHS and to substitute for those bits on the RHS. Finally, we allow * to stand for any wire whose input is not to be consumed in this scheduling cycle (on the LHS of a rule), or whose value is not to be produced as an output from this rule (on the RHS of a rule), thereby introducing asynchronicity into the design. The language containing all these constructs corresponds to *HW-Hume*, whose syntax is shown in Figure 5. τ defines the valid HW-Hume types: tuple types $\tau_1 \times \ldots \times \tau_n$; bounded vector types, **vector** n **of** τ, where n is the bound; bit types, **word** 1 and named types, typeid.

A Simple Traffic Lights Example in HW-Hume. For example, consider a set of traffic lights as used in the UK, which displays a sequence of red (stop), red and amber (prepare to go), green (go) and amber (prepare to stop) lights. We might encode these state changes as:

light(s)	state	meaning	red	amber	green
red	0	stop	1	0	0
red/amber	1	prepare to go	1	1	0
green	2	go	0	0	1
amber	3	prepare to stop	0	1	0

where a `1` indicates that the corresponding light is on and a `0` that it is off. In HW-Hume, we could model a traffic light as a box which changes state when it receives a signal. We encode the state as a two-bit binary number, and the light settings as a tuple of bits as:

```
type Bit = word 1;

box lights1
in  (signal::Bit, state::(Bit,Bit))
out (state'::(Bit,Bit),lights::(Bit,Bit,Bit))
match
  (1,(0,0)) -> ((0,1),(1,1,0))
| (1,(0,1)) -> ((1,0),(0,0,1))
| (1,(1,0)) -> ((1,1),(0,1,0))
| (1,(1,1)) -> ((0,0),(1,0,0));

wire change         to lights1.signal;
wire lights1.state  to lights1.state';
wire lights1.lights to display;
```

where `change` and `display` are unspecified external connections. On each box cycle, if the `signal` on `change` is `1` then, for the current `state`, a new setting on `lights` is sent to `display` and the `state'` changes. Note that HW-Hume may either be implemented directly in hardware or in software using statically allocated memory and/or registers.

2.2 First-Order Functions, Conditionals and Local Definitions: FSM-Hume

The standard boolean operators (and, or, not etc.) can easily be constructed from boolean logic circuits so there is no loss of formal properties if we allow these operators to be used on the right hand sides of matches. We could generalise booleans to user-defined ordinal types, with operations over such types defined by explicit pattern matching against values of those types, again without loss of formal properties.

FSMs can express addition and subtraction for arbitrary integers, but not multiplication. However, they can express multiplication for fixed-precision integers. Once again, we lose nothing by allowing fixed-precision integers on wires and basic fixed-precision arithmetic in the right hand sides of matches. Similarly, fixed-precision floating point numbers can be represented at finite bit sequences and manipulated by digital circuits, so we can allow these as well, and we can also allow characters and fixed-size strings. Finally, we also allow (non-recursive) discrimated unions of such types (*datatypes*), which could be implemented as tagged tuples. For example, we can define a datatype of possible kinds of input as:

decl ::=	... | *function* | *funtype* | *datatype*	
datatype ::=	**data** typeid = *constr*$_1$ | ... | *constr*$_n$	$n \geq 1$
constr ::=	conid τ_1 ... τ_n	$n \geq 1$
function ::=	*fmatch*$_1$ | ... | *fmatch*$_n$	$n \geq 1$
fmatch ::=	funid *pat*$_1$ ... *pat*$_n$ = *expr*	$n \geq 1$
funtype ::=	funid :: τ_1 → ... → τ_n	$n \geq 1$
τ ::=	... | **bool** | **int** n | **word** n | **float** n | **char** | **string**	
expr ::=	...	
	| *bool* | *int* | *word* | *float* | *char* | *string*	
	| conid *expr*$_1$ ... *expr*$_n$	$n \geq 0$
	| funid *expr*$_1$ ... *expr*$_n$	$n \geq 1$
	| **if** *expr*$_1$ **then** *expr*$_2$ **else** *expr*$_3$	
	| **case** *expr* **of** *fmatches*	
	| **let** *valdefn*$_1$; ... ; *valdefn*$_n$ **in** *expr*	$n \geq 1$
	| *expr* **within** *heap* , *stack*	
valdefn ::=	varid = *expr* | varid :: τ	
pat ::=	...	
	| *int* | *float* | *char* | *string*	
	| conid *pat*$_1$ ... *pat*$_n$	$n \geq 0$
box ::=	**box** boxid *ins* *outs* **match/fair** *matches*	

Fig. 6. Additions to HW-Hume Syntax for FSM-Hume

```
data Input = Mouse (int 16) (int 16)
           | Joystick (int 8) (int 8) (int 8) bool
           | Keyboard char;
```

It is convenient to introduce function abstractions to provide better program structure and reduce repetition. If they are total, such function abstractions do not change termination properties, if they merely abstract over existing computation forms. Moreover, since we have only finite non-recursive data structures and cannot form partial functions, it is possible to construct an extensional representation of a function as a pairing from each possible input value to the corresponding result. In this way we can retain decidability of equivalence even in the presence of functions. We allow functions to be formed over a series of patterns, each of which matches one argument against a concrete value, a variable (which is bound to the actual value), or a constructed pattern with a specific constructor.

A common situation occurs when a function is applied to a single concrete expression. A convenient syntactic form to deal with this is a *case-expression*. Where the case-expression ranges over boolean values, this may be simplified to a conventional conditional (*if*-expression). Local value definitions (*let*-expressions) are a natural complement to functions. A local definition allows one or more values to be named, so avoiding recomputation (assignment serves a similar purpose in an imperative

language). Local definitions scope over an enclosed expression. We can allow static constraints to be imposed on time or space usage using *within*-expressions. These constraints must be statically verified against the actual costs, or else raise an exception if violated dynamically. For simplicity, we omit a treatment of exceptions and time here. Finally, top-level rule matching may now be either *unfair* as with HW-Hume, always starting with the first rule and considering each rule in the order given by the programmer, or *fair*, where a rule that matches successfully is tried last in the succeeding iteration.

HW-Hume plus these extensions (Figure 6) corresponds to *FSM-Hume*. This moves us from a pure finite-state language, similar to e.g. Esterel [Boussinot and de Simone 1991], to one which provides some stronger programming abstractions.

The Traffic Lights Example in FSM-Hume. The traffic lights example may be adapted to FSM-Hume as follows:

```
data Light = Red | Amber | Green | Red_Amber;

box lights2
in  (signal::Bit,   state::Light)
out (state'::Light, lights::(Bit,Bit,Bit))
match
  (1,Red) ->        (Red_Amber, (1,1,0))
| (1,Red_Amber) ->  (Green,     (0,0,1))
| (1,Green) ->      (Amber,     (0,1,0))
| (1,Amber) ->      (Red,       (1,0,0));
```

Here, we have simply substituted a discriminated union type `Light` for the bit representation of the state used in HW-Hume. We could equally well have used integers. However, we may next modify this example to drive the lights with internal delays instead of using an external signal:

```
type Delay = int 32;

red_delay = ...;   redAmber_delay = ...;
amber_delay = ...; green_delay = ...;

box lights3
in  (delay:: Delay, state:: Light)
out (delay'::Delay, state'::Light, lights::(Bit,Bit,Bit))
match
  (d,Red) ->       if d < red_delay
                   then (d+1,Red,*)
                   else (0,Red_Amber,(1,1,0))

| (d,Red_Amber) -> if d < redAmber_delay
                   then (d+1,Red_Amber,*)
                   else (0,Green,(0,0,1))

| (d,Green) ->     if d < green_delay
```

datatype ::=	**data** typeid $\alpha_1 \ \ldots \ \alpha_m$ = $constr_1$ | ... | $constr_n$	$m \geq 0,\ n \geq 1$
type ::=	**type** typeid $\alpha_1 \ \ldots \ \alpha_m$ = σ	$m \geq 0$
funtype ::=	funid :: $\sigma_1 \rightarrow \ \ldots \ \rightarrow \sigma_n$	$n \geq 1$
constr ::=	conid $\sigma_1 \ \ldots \ \sigma_n$	$n \geq 1$
σ ::=	$\tau \mid \alpha \mid \sigma_1 \times \ \ldots \ \times \sigma_m \mid$ **vector** $(n \mid \text{id})$ **of** σ	
	| typeid $\sigma_1 \ \ldots \ \sigma_n \mid \sigma_1 \rightarrow \sigma_2$	$n \geq 1$

Fig. 7. Modifications to FSM-Hume Syntax for Template-Hume

```
                    then (d+1,Green,*)
                    else (0,Amber,(0,1,0))

| (d,Amber) ->      if d < amber_delay
                    then (d+1,Amber,*)
                    else (0,Red,(1,0,0)) ;

wire lights3.delay' to lights3.delay;
wire lights3.state' to lights3.state;
wire lights3.lights to lights;
```

Now, at each stage, the box will cycle repeatedly in the same state without changing the light settings until the delay has been reached. Finally, we can abstract over the common expressions in the RHS using a function definition:

```
check d delay state state' =
  if d < delay
  then (d+1, state,  *)
  else (0,   state', lights_to_bits state');

lights_to_bits Red =       (1,0,0);
lights_to_bits Red_Amber = (1,1,0);
lights_to_bits Amber =     (0,1,0);
lights_to_bits Green =     (0,0,1);

box lights4
in  (delay:: Delay, state:: Light)
out (delay'::Delay, state'::Light, lights::(Bit,Bit,Bit))
match
  (d,Red) ->       check d red_delay      Red       Red_Amber
| (d,Red_Amber) -> check d redAmber_delay Red_Amber Green
| (d,Green) ->     check d green_delay    Green     Amber
| (d,Amber) ->     check d amber_delay    Amber     Red
```

Note that although function definitions are allowed in FSM-Hume, the type definitions restrict these to be first-order definitions (no function parameters). Moreover, functions are not allowed to be passed on wires, although datatypes may be.

In this way, wires are restricted to finite *data* structures: infinite *co-data* structures (streams) or functions may not be carried on wires.

2.3 Predefined Higher-Order Functions: Template-Hume

As the lengths of vectors grow, expressing vector operations through pattern matching and operations on explicit fields becomes increasingly cumbersome. A simplification is to introduce pre-defined *higher-order functions* (HOFs) over vectors. Higher order functions take other functions as arguments and apply them across structures, so acting as generic patterns supporting program construction from user-defined components. For example, the commonly used *map* family of HOFs applies some other function to every element of a structure, building a new structure with the same shape but potentially different type (the inverse of the object-oriented approach, where the structure cannot be abstracted across in this way).

Such HOFs are shorthand for explicit pattern-matching and field manipulation across a vector. In the general case, the structure that is being built may be a control structure as well as a data structure, and the result may itself be a function. Moreover, since the HOF does not affect the structure, it may be defined to be *generic* (or *parametrically polymorphic*). In this way, we achieve a powerful tool for software engineering: constructing software from small reusable components using powerful, user-definable function combining forms (patterns or HOFs). We use arrows to indicate function types, so that **int 8** $\rightarrow$ *char* denotes a function from an 8-bit integer to a character etc.

For example, `mapvec :: (a->b) -> vector m of a -> vector m of b` is a function of two arguments, `a->b` and `vector m of a`. It applies its first argument (a *polymorphic* function from some type `a` to some type `b`) to each element of its second argument (a vector of length `m` whose elements are each of type `a`) to yield a new result vector, also of length `m`, whose elements are of type `b`. We can then use this predefined function in the following definition of `doublevec`, a function which doubles each of the elements contained in an arbitrary vector of integers.

```
double :: Int -> Int;
double x = x + 2;

doublevec :: vector m of Int -> vector m of Int;
doublevec v = mapvec double v;
```

Other commonly found HOF patterns include *filters* for selecting elements from a structure according to some predicate, *folds* for iterating over some structure, and *compositions* for building complex functions from simpler ones.

FSM-Hume plus a standard set of predefined polymorphic higher-order functions and data types corresponds to *Template-Hume*. Note that the syntax of Template-Hume (Figure 7) is identical to that of FSM-Hume apart from the types, which now support polymorphism and function types. The differences between the two levels arise entirely in the types of legal programs that are supported by each level. Note also that boxes remain monotyped and do not accept function arguments. In this way we retain the FSM-Hume restrictions on the kinds of values that may be carried over wires.

The power of expressivity available with HOF templates imposes a price in terms of determining both time and space behaviour. For example, the cost of evaluating the `mapvec` HOF is a product of the length of the vector and the cost of applying the function to each element of that vector. It follows that, in order to determine the cost of using a HOF, it is necessary to track the cost of evaluating any function argument (the *latent cost* of the function), and that this cost may need to be passed through several HOFs. We also require cost equations to be provided for each HOF. These equations may be *checked* against the definition of the HOF, and may also be used to *infer* the cost of using that HOF in any program context.

2.4 Primitive Recursive Functions: PR-Hume

If we allow *general recursion* in either functions or data, then we lose the ability to decide equivalence over values. Restricting ourselves to *primitive recursion* does, however, enforce termination [Arkoudas and McAllester 1996] and hence permit determinate time- and space-costing, subject to the above caveats.

Classical primitive recursive functions [Peta 1967] are defined by induction over a natural number, n, with a base case when n is 0, and a general case defined in terms of its predecessor, $n-1$. There are a variety of standard primitive recursive forms including linear recursion and course of values recursion. So, for example:

```
fac 0 = 1;
fac n = n * fac (n-1);
```

defines the primitive recursive factorial function, which clearly runs in bounded time and space provided n is finite. We can extend this notion to functions on more complex data structures, provided there is always both a base case and an inductive general case that always makes progress towards the base case. In this way, the function will allow the use of structural induction. For example:

```
data List a = Empty | Cons a (List a);

map f Empty = Empty;
map f (Cons x l) = Cons (f x) (map f l);
```

defines the inductive `List` data structure, with the base case `Empty` and the inductive `Cons` case, which constructs a new list by adding an element to the predecessor list (which is therefore one element smaller). It also defines the primitive recursive `map` function, which inducts over the structure of its second argument to create a result list. Note that `L` is always one element shorter than `Cons x l` and so the recursion case makes progress towards the base case for `Empty` which always ends a `List`.

Note that non-primitive recursive functions typically embody nested or *double* recursion, for example Ackermann's Function [Ackermann 1928] which grows faster than primitive recursive functions. Note also that it is undecidable whether or not an arbitrary function is primitive recursive: were it decidable then, since primitive recursive functions must terminate, such an analysis would also solve the halting problem.

PR-Hume is defined to be Template-Hume extended to allow primitive recursion in function definitions, as discussed above, and also to support inductive datatype

definitions. Since we have seen that precise determination of primitive recursion is undecidable, it follows that we can only precisely characterise the PR-Hume level by reference to a specific partial analysis, although we might determine by inspection that a given function is definitely primitive recursive through direct instantiation of a classic form. [Michaelson 2000] discusses such a partial analysis for primitive recursiveness based on syntactic exposure of nested recursion.

As before, the syntax of PR-Hume is identical to that of Template-Hume and FSM-Hume: the difference lies in allowing (primitive) recursion in user-defined functions and in supporting inductive data structures. For example, consider the generic `paths` function that determines all possible paths through a polymorphic binary tree.

```
data Tree a = Leaf a | Node a (Tree a) (Tree a);

paths :: Tree a -> [[a]];
paths (Leaf v) = [[v]];
paths (Node v lt rt) = [map (v :) (paths lt), map (v :) (paths rt)];
```

The cost of the `paths` function is composed from the costs of the recursive calls to construct the paths for the left and right subtrees.

2.5 Unrestricted Recursion: Full Hume

Finally, when we allow general recursion in either functions or data, we return to our Turing Complete language, *Full Hume*, about which we can determine no properties. It is an open question whether there are levels of recursion above PR-Hume for which we may statically determine time or space costs, though *Walther recursion* [Arkoudas and McAllester 1996], a generalisation of primitive recursion, may offer such a possibility if it can be extended to cover higher-order polymorphic functions.

3. SPACE COST MODELS FOR THE HUME LEVELS

This section describes a series of cost models for determining the space usage of boxes for each Hume level in turn, from HW-Hume up to PR-Hume. The models are defined with reference to our prototype Hume implementations, and provide a statically derivable upper bound on the space usage of Hume programs in each level. Further details of our approach including cost inference algorithms and proof sketches for soundness and completeness of the analysis can be found in earlier papers [Portillo et al. 2003b; 2003a] and in Vasconcelos' forthcoming thesis [Vasconcelos 2006].

3.1 Costs as Types

Our cost models are defined as *type-and-effect systems*, deriving cost information from the structure of program terms as an adjunct to the usual type information. This approach has recently found favour for program analysis since it allows behavioural information to be derived structurally and compositionally from a program, supports separate compilation, provides a framework for supporting proofs, and allows well-understood type-inference engines to be exploited for the imple-

$$\boxed{\mathrm{E} \overset{\text{box}}{\vdash} \text{box \$ Bits}}$$

$$(1)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{type}}{\vdash} \tau_i\ \$\ c_i \qquad \mathrm{E} \overset{\text{body}}{\vdash} \text{body} : (\tau'_1 \times \ldots \times \tau'_m) \rightarrow (\tau_1 \times \ldots \times \tau_n)\ \ \$\ \ c}{\mathrm{E} \overset{\text{box}}{\vdash} \textbf{box}\ \text{boxid}\ \textbf{in}\ (\ \text{inid}'_1 : \tau'_1, \ldots,\ \text{inid}'_m : \tau'_m\ )\ \textbf{out}\ (\ \text{outid}_1 : \tau_1, \ldots,\ \text{outid}_n : \tau_n\ )\ \text{body}\ \$\ \textstyle\sum_{i=1}^{n}\ c_i + c}$$

$$\boxed{\mathrm{E} \overset{\text{body}}{\vdash} \text{body} : \tau\ \ \$\ \ \text{Bits}}$$

$$(2)\quad \frac{\forall i.\ 1 \le i \le n,\ \ \mathrm{E} \overset{\text{hpat}}{\vdash} \mathit{pat}_i : \tau\ \ \Rightarrow\ \ \mathrm{E}_i \qquad \forall i.\ 1 \le i \le n,\ \ \mathrm{E} \cup \mathrm{E}_i \overset{\text{bit}}{\vdash} \mathit{expr}_i : \tau'\ \ \$\ \ c_i}{\mathrm{E} \overset{\text{body}}{\vdash} \textbf{match}\ \ \mathit{pat}_1\ \ \rightarrow\ \ \mathit{expr}_1 \mid \ldots \mid \mathit{pat}_n\ \ \rightarrow\ \ \mathit{expr}_n : \tau\ \rightarrow\ \tau'\ \ \$\ \ \max_{i=1}^{n}\ c_i}$$

Fig. 8. HW-Hume space cost axioms for boxes and box bodies

mentation of the analysis. In our systems, type-and-effect judgements determine both types and costs from program terms, so, for example:

$$\boxed{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr} : \tau\ \ \$\ \ \text{Cost}}$$

is the form of a judgement that derives a type τ and a cost Cost from an expression *expr* in the context of an environment E that maps identifiers to types. The \$ symbol is used to separate the effect(s) from the type.

3.2 A Space Cost Model for HW-Hume

We first define a cost model for the coordination layer (Figure 8). The rules use costs for type declarations defined in Figure 9. Costs are given in the context of an environment E, which contains type, heap and cost bindings for variables and constructors. The initial environment is extended by patterns in box rules, and where appropriate by function definitions, local definitions and case-expressions. Cost values of type Bits are given in terms of integral numbers of bits, representing the cost of storing those values in our prototype HW-Hume implementation.

Figure 10 gives cost rules for HW-Hume expressions. The cost of constructing a bit constant (rule 7) is just one bit. The cost of a variable (rule 8) is looked up from the environment, and the cost of an ignored output position (rule 9) is, of course, 0. New constructed data values such as tuples or vectors (rules 10/11) are costed as the sum of their components. Finally, in HW-Hume, patterns in box matches are used solely to construct type environments for bound variables (Figure 11).

3.3 Dynamic Memory Usage in the Prototype Hume Abstract Machine

Before extending the analysis to more powerful Hume levels than HW-Hume, we need to consider how we should cost programs written in such levels. Since we have

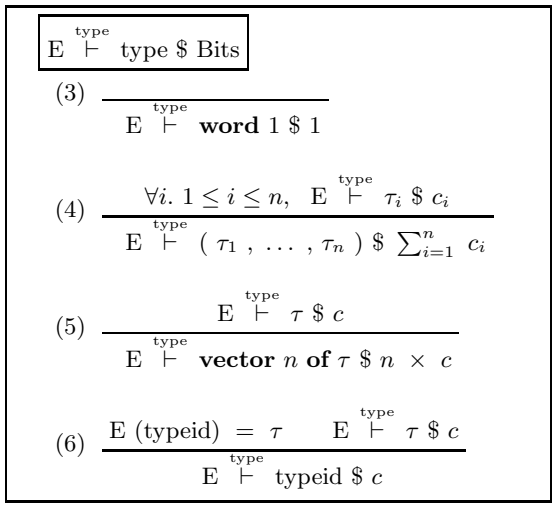

Fig. 9. Cost axioms for HW-Hume type declarations

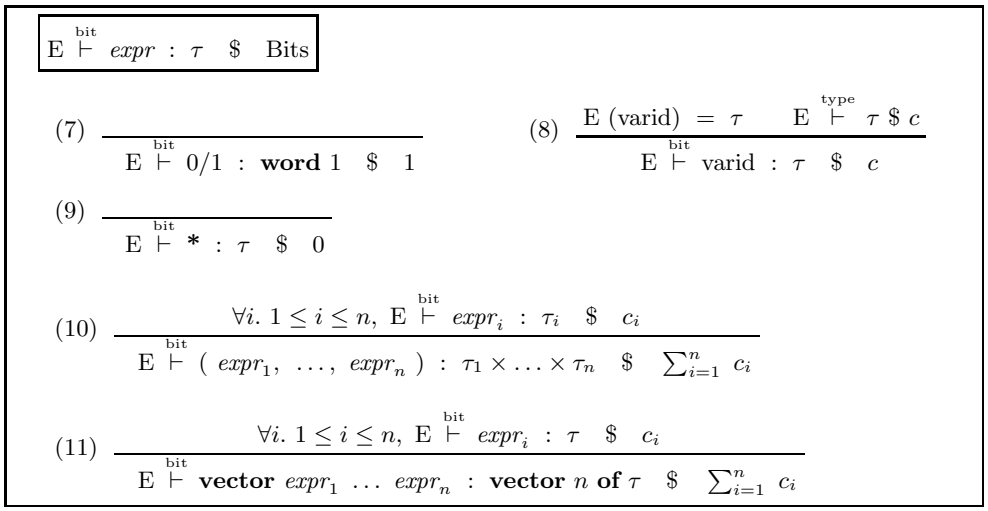

Fig. 10. Cost axioms for HW-Hume expressions

already constructed a prototype abstract machine implementation for demonstration purposes that is characteristic of a real machine implementation, but which is deliberately easy to describe, we have chosen to use this as the target for the analyses described here. The prototype Hume Abstract Machine (`pham`) [Hammond 2006] is *loosely* based on the design of the classical G-Machine [Augustsson 1987; Johnsson 1984], restricted to strict evaluation and with extensions to manage concurrency and asynchronicity. Each box has its own dynamic stack and heap. For simplicity, all arguments to function calls, return values and box inputs are held on the stack as (1-word) heap pointers. All available box inputs are copied from

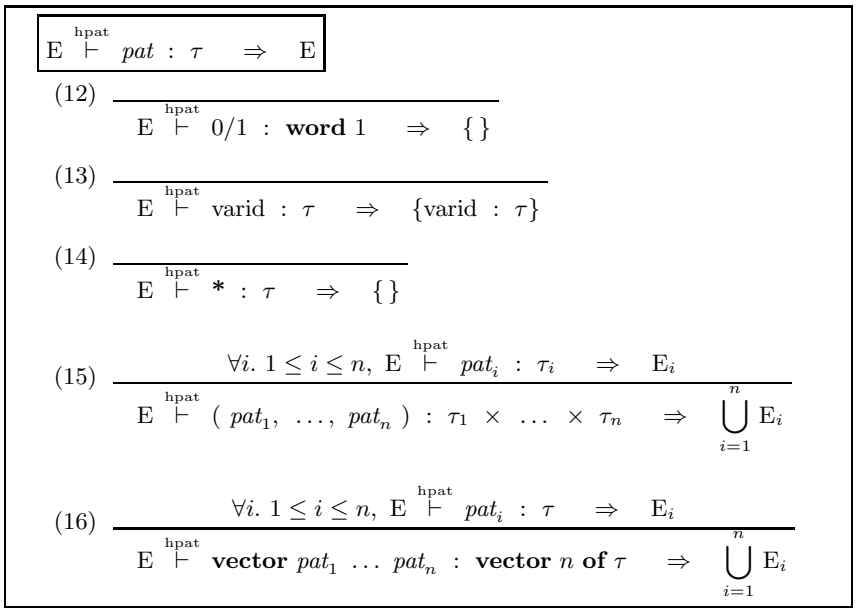

Fig. 11. Type axioms for HW-Hume patterns

the corresponding wire into the box heap at the start of each cycle. All other heap allocation happens as a consequence of executing some right-hand-side expression. For now, we consider only heap allocation in the cost model – deallocation of all allocated heap will occur when a box completes. In future, we expect to also consider real-time deallocations.

In the prototype implementation, all heap cells are *boxed* [Peyton Jones 1992] with tags distinguishing different kinds of objects. Furthermore, tuple structures require *size* fields, and data constructors also require a *constructor tag* field to distinguish different constructors for the same datatype. All data objects in a structure are referenced by pointers (again, a boxed representation). For simplicity, each field in a structure is constrained to occupy exactly one word of memory. There is one special representation: strings are represented as a tagged sequence of bytes. Figure 12 gives a table of constants representing the fixed heap costs for each kind of structure in terms of words of memory.

Clearly, it would be easy to reduce heap usage using a more compact heap representation such as that used by the STG-Machine [Peyton Jones 1992] or the OCAML implementation [Cousineau and Mauny 1998][4]. For now, we are, however, primarily concerned with the in-principle problem of bounding and predicting memory usage with reference to some quantifiable model and measurable implementation. Small changes to data representations and other optimisations 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.

[4]Indeed, we estimate that heap usage can be reduced by a factor of 2-4 this way.

constant	value (words)
$\mathcal{H}_{bool}$	2
$\mathcal{H}_{int32}$	2
$\mathcal{H}_{float32}$	2
$\mathcal{H}_{con}$	3
$\mathcal{H}_{tuple}$	2
...	...
$\mathcal{H}_{string}$	1

Fig. 12. Sizes of tags etc. in the prototype Hume Abstract Machine, required by the analysis.

3.4 A Space Cost Model for FSM-Hume

Figure 13 gives a cost model for a representative selection of FSM-Hume expressions. Here, $\boxed{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr} : \tau \quad \$ \quad \mathrm{Heap}, \mathrm{Stack}}$ is a judgement deriving a sized type τ and two costs which are natural numbers representing total heap allocations (Heap) and maximum stack depth (Stack) respectively, when evaluating *expr*. As before, the environment E maps identifiers to (sized) types. We will consider each case in turn.

Firstly, an integer literal (rule 17) yields the Int type plus heap and stack information. In our prototype implementation, a new boxed integer is constructed (with heap size $\mathcal{H}_{int32}$, and a pointer to this is pushed on the stack (increasing the stack depth by 1).

Secondly, since Hume uses strict evaluation, and values must therefore be evaluated before being bound to a variable, variables have no heap cost (rule 18). There is a stack cost, however, since a pointer to the value that has been looked up will be pushed on the stack.

Thirdly, the heap allocated by a function application (rule 19) is simply that used in evaluating the body of the function plus that used for each argument. Since each evaluated argument is pushed on the stack before the function is applied, this must be taken into account when calculating the maximum stack depth. The cost of building a new data constructor value such as a tuple (rule 20) or a user-defined constructed type (omitted) are similar to those for function applications but with constant latent costs (for building the structure in heap).

Finally, the heap usage of a conditional (rule 21) is the sum of 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 or either branch. Case expressions (omitted) are costed analogously.

Figure 14 gives the costs of function declarations. Here, the stack and heap costs of evaluating the body of the function are recorded as latent costs in the type of the function, and this is returned in the form of an environment. Finally, patterns in function and box matches contribute to stack usage in two ways (Figure 15): firstly, the value attached to each variable is recorded in the stack frame (rule 25); and secondly, each nested data structure that is matched must be unpacked onto the stack (requiring n words of stack) before its components can be matched by the abstract machine (as shown in rule 26 for tuples).

It is also necessary to modify the rules for boxes to record both stack and heap costs, and the rules for types to record maximum heap costs for specific constructors. We omit definitions of these rules here.

$$\boxed{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr} : \tau \quad \$ \quad \text{Heap}, \text{Stack}}$$

$$(17)\ \frac{}{\mathrm{E} \overset{\text{space}}{\vdash} n : \mathit{Int} \quad \$ \quad \mathcal{H}_{int32},\ 1}$$

$$(18)\ \frac{\mathrm{E}\ (\text{varid}) = \tau}{\mathrm{E} \overset{\text{space}}{\vdash} \text{varid} : \tau \quad \$ \quad 0,\ 1}$$

$$(19)\ \frac{\begin{array}{c}\mathrm{E}\ (\text{funid}) = \tau_1 \xrightarrow{0;\ 0} \ldots \xrightarrow{0;\ 0} \tau_n \xrightarrow{h;\ s} \tau \\ \forall i.\ 1 \leq i \leq n,\ \ \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_i : \tau_i \quad \$ \quad h_i, s_i\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} \text{funid}\ \mathit{expr}_1\ \ldots\ \mathit{expr}_n : \tau \\ \$ \quad \sum_{i=1}^{n} h_i + h,\quad \max\left(\max_{i=0}^{n-1}\ (s_{n-i} + i), s + n\right)\end{array}}$$

$$(20)\ \frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_i : \tau_i \quad \$ \quad h_i, s_i}{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} (\ \mathit{expr}_1,\ \ldots,\ \mathit{expr}_n\ ) : \tau_1 \times \ldots \times \tau_n \\ \$ \quad \sum_{i=1}^{n} h_i + n + \mathcal{H}_{tuple},\quad \max_{i=0}^{n-1}(s_{n-i} + i)\end{array}}$$

$$(21)\ \frac{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr} : \tau \quad \$ \quad h_e, s_e}{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}\ \texttt{within}\ h, s : \tau \quad \$ \quad \mathit{min}\ (h, h_e),\quad \mathit{min}\ (s, s_e)}$$

$$(22)\ \frac{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_1 : \text{bool} \quad \$ \quad h_1, s_1 \\ \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_2 : \tau \quad \$ \quad h_2, s_2 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_3 : \tau \quad \$ \quad h_3, s_3\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} \texttt{if}\ \mathit{expr}_1\ \texttt{then}\ \mathit{expr}_2\ \texttt{else}\ \mathit{expr}_3 : \tau \\ \$ \quad h_1 + \mathit{max}(h_2, h_3),\quad \mathit{max}(s_1, s_2, s_3)\end{array}}$$

Fig. 13. Stack and Heap cost axioms for FSM-Hume expressions

$$\boxed{\mathrm{E} \overset{\text{decl}}{\vdash} \text{decl}\ \$\ \mathrm{E}}$$

$$(23)\ \frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n, \forall j.\ 1 \leq j \leq m,\ \ \mathrm{E} \overset{\text{pat}}{\vdash} \mathit{pat}_{ij} : \tau_j \quad \$ \quad sp_{ij} \quad \Rightarrow \quad \mathrm{E}_{ij} \\ \mathrm{E}_i = \bigcup_{j=1}^{m} \mathrm{E}_{ij} \qquad \forall i.\ 1 \leq i \leq n,\ \ \mathrm{E} \cup \mathrm{E}_i \overset{\text{space}}{\vdash} \mathit{expr}_i : \tau \quad \$ \quad h_i, s_i \\ h = \max_{i=1}^{n} h_i \qquad \mathcal{S} = \max_{i=1}^{n} \left(s_i + \max_{j=1}^{m}(sp_{ij} + j - 1)\right) \\ \tau' = \tau_1 \xrightarrow{0;\ 0} \ldots \xrightarrow{0;\ 0} \tau_m \xrightarrow{h;\ s} \tau\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{decl}}{\vdash} \text{funid}\ \mathit{pat}_{11}\ \ldots\ \mathit{pat}_{1m} = \mathit{expr}_1 \mid \ldots \mid \text{id}\ \mathit{pat}_{n1}\ \ldots\ \mathit{pat}_{nm} = \mathit{expr}_n \\ \$\ \{\ \text{funid} : \tau'\}\end{array}}$$

Fig. 14. Cost axioms for FSM-Hume function declarations

$$\boxed{\mathrm{E} \overset{\text{pat}}{\vdash} pat : \tau \quad \$ \quad \mathrm{Stack} \quad \Rightarrow \quad \mathrm{E}}$$

$$(24)\ \frac{}{\mathrm{E} \overset{\text{pat}}{\vdash} n : Int \quad \$ \quad 0 \quad \Rightarrow \quad \{\}}$$

$$(25)\ \frac{}{\mathrm{E} \overset{\text{pat}}{\vdash} \mathrm{varid} : \tau \quad \$ \quad 1 \quad \Rightarrow \quad \{\mathrm{varid} : \tau\}}$$

$$(26)\ \frac{\forall i.\ 1 \leq i \leq n,\ \mathrm{E} \overset{\text{pat}}{\vdash} pat_i : \tau_i \quad \$ \quad s_i \quad \Rightarrow \quad \mathrm{E}_i}{\mathrm{E} \overset{\text{pat}}{\vdash} (\ pat_1,\ \ldots,\ pat_n\ ) : \tau_1 \times \ldots \times \tau_n \quad \$ \quad \sum_{i=1}^{n} s_i + n \quad \Rightarrow \quad \bigcup_{i=1}^{n} \mathrm{E}_i}$$

Fig. 15. Stack cost axioms for FSM-Hume patterns

3.5 Space Cost Model for Template-Hume

Template-Hume introduces two concepts that affect costing: polymorphism and higher-order functions. Parametric polymorphism is introduced in the usual functional-language way by allowing types to contain type variables, which may be universally quantified. For example, for the predefined *maplist* function on lists, we can derive the type $map \ : \ \forall a, b.(a \rightarrow b) \rightarrow (\mathbf{List}\ a) \rightarrow (\mathbf{List}\ b)$. In the type-and-effect system, this is manifested in the rules for defining and using functions (rules 28 and 29), variables (rule 27) and constructors (omitted).

A number of changes must be made in order to deal with higher-order functions. The costs of applying higher-order functional arguments clearly depend on the costs associated with the concrete functions that are ultimately supplied as those arguments. The latent cost mechanism is consequently used to record those costs, to pass them to applications of higher-order functions, and supply the actual cost where functions are finally applied. Predefined functions must be supplied with appropriate latent costs.

Since the costs of functions may now depend on the sizes of their arguments (for example, where recursive templates such as *map* are used), it is necessary to extend the types we use internally in the analysis to also include size information. We consequently define *sized types* in the style of [R.J.M. Hughes and Sabry 1996] to be:

$$\tau ::= \tau_1 \times \ldots \times \tau_m \ \mid \ \tau_1 \xrightarrow{h_1;\ s_1} \tau_2 \ \mid \ \kappa^z\ \tau_1\ \ldots\ \tau_n \ \ m, n \geq 1$$

where $\kappa^z\ \tau_1\ \ldots\ \tau_n$ represents a *sized* data type with type constructor κ and size z (defined below). $\tau_1 \xrightarrow{h;\ s} \tau_2$ is the type for a function, where h and s the *latent* stack and heap costs for that function (i.e. the stack and heap costs that are incurred when the function is applied to some argument). Size and cost expressions are defined as follows:

$$\mathrm{Heap}, \mathrm{Stack} \ \in \ z \ ::= \ \ell \mid n \mid \omega \mid z_1 + z_2 \mid z - n \mid z_1 \times z_2 \mid \max(z_1, z_2) \mid \min(z_1, z_2)\,.$$

where ℓ is a size variable, n is a concrete size, ω represents the unknown size and

we also allow some basic arithmetic operations on size/cost. Since sizes appear in types, types may now be *polymorphic* in size or cost variables as well as in the usual type variables.

For example, we could give a sized type for the standard *map* function on lists as $map\ :\ \forall a, b, z, h, s.\ (a \xrightarrow{h;\ s} b) \xrightarrow{5;\ 1} [a]^z \xrightarrow{((h+4)\times z+2);\quad (s+3)} [b]^z$ where *map* takes a function with latent costs h and s and a list of length z ($[a]^z$) and returns a list of the same length but with a possibly different element type. The heap allocated when the result is returned ($(h+4)\times z+2$) is proportional to the length of the list (z) and the heap allocated by each call to the function argument (h). The constants 4 and 2 reflect the sizes of list nodes and empty lists respectively. The maximum stack depth ($s+3$) reflects the cost of evaluating the function argument (s) in some context where additional values have been pushed on the stack (shown here as the constant 3). The first argument position is assigned constant heap and stack costs ($\xrightarrow{5;\ 1}$), reflecting the heap cost of building a *closure* to hold intermediate values where the *map* function is *partially applied* to only its first argument, and the cost of subsequently placing this on the stack. Therefore, in the definition of *pathlengths* below, where we assume that *Tree*, *paths* and *length* are predefined.

```
maplen :: List a -> Int;
maplen l = map length l;

pathlengths :: Tree a -> List Int;
pathlengths tree = map maplen (paths tree);
```

`pathlengths` will take `tree` as its argument, apply the `paths` function to that tree to give a list of paths, where each path is a list of elements, of type a, and then use `maplen` to map the *length* function over each path. The result will be a list of integers representing the lengths of each path in the tree.

Since we are interested in obtaining upper bounds on sizes and costs (the former to give limits on inductions over recursive templates, the latter to constrain stack and heap bounds), we choose our sizes to indicate the greatest magnitude that specific values may take. For example, we can state that $10 : \mathrm{Nat}^{10}$ or $[4, 2, 1] : \mathrm{List}^3\mathrm{Nat}^4$, and this information can then give us bounds on the sizes of recursive template applications.

Where sizes do not match exactly (as with conditionals), it is necessary to introduce a notion of *size subtyping* over the ordinal ordering on sizes, with ω, the limit size, encompassing all finite sizes. It is also necessary to allow *weakening* of sizes as required. This concept is extended to heap and stack costs. For example, given `f m n = m + n` we can determine $f\ :\ \forall m, n. Nat^m\ \rightarrow\ Nat^n\ \rightarrow\ Nat^{m+n}$, $f\ 2\ 3\ :\ Nat^5$, $f\ 100\ :\ \forall n. Nat^{100+n}$. Note that functions are *contravariant*: a function will accept any smaller concrete parameter than that required by the formal parameter, and the result may be *weakened* to any required size that is greater than the result size.

Figure 16 shows how the cost rules change to deal with Template-Hume. In rule 27, we derive a size from an integer constant that is the magnitude of that constant. Rule 28 modifies the FSM-Hume rule for variables to allow polymorphism in both types and sizes/costs. Rule 29 allows for polymorphism in function applications,

allows latent heap/stack costs to be carried in positions other than the final application (so allowing for the costs of partial applications) and supports size subtyping for both function arguments and results. Rule 30 shows how the sizes and costs are weakened for conditionals. Finally, rule 31 shows how Here $\mathcal{H}_{clos}$ is the fixed cost of constructing a function closure, TV (τ) gives a vector of the free type variables in τ (represented as $\vec{\alpha}$), and ZV (τ) gives a vector of the free size variables in τ.

3.6 Space Cost Model for PR-Hume

We are now in a position to state cost rules for primitive recursive function definitions[5]. Figure 17 extends the cost rules for Template-Hume function definitions to cover primitive recursive functions. The only changes are to firstly insert the type of the recursive function into the type environment before typing the body of the definition, to secondly identify an induction variable k that indicates how sizes varying from one use of the recursive function to the next (so giving a general induction term), and finally to ground the recurrence so that when the least size ε is subsituted for the induction variable, the *universal type* (**U**, the type containing all possible types) results. The domain of sizes is consequently extendeded to be $z ::= \ldots \mid \varepsilon$. [Portillo et al. 2003a] contains further details of this approach including relevant soundness and completeness proofs.

3.7 Cost Analyses for Hume Levels

The cost models we have given here may now be used as the basis for inference algorithms capable of deriving stack and heap cost information for all Hume levels up to and include PR-Hume. We refer to previous papers [Portillo et al. 2003b; 2003a] and to Vasconcelos' forthcoming thesis [Vasconcelos 2006] for full details of how this can be done. The primary technical difficulties involved in constructing such an analysis occur with obtaining solutions to size recurrences for primitive recursive definitions in the presence of higher-order polymorphic function definitions. We have investigated solutions involving hybrid type-based and abstract interpretation approaches that yield accurate sizes and costs in many situations. Some results obtained from these analyses will be described in the following section.

4. REFACTORING PR-HUME COMPONENTS TO HW-HUME COMPONENTS

We have been exploring a software development methodology for Hume where program components written in full Hume that have non-existent or highly approximate costs are successively refactored to lower levels of Hume where costs are increasingly precise. Our long term goal is that such refinement will be interactive and guided by static analyses to identify components which either cannot be costed at all (i.e. with ω cost), or which yield excessively inaccurate costs.

4.1 Refactoring Hume Levels

In Section 3, we have shown how to construct bounded cost models for Hume levels up to and including PR-Hume using a layered approach. We will now show how HW-Hume programs may be constructed from equivalent PR-Hume programs

[5]We note in passing that there are alternative possible definitions of primitive recursion. The notion used here is by reference to the types that we can form using the rules given in this paper.

$\boxed{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr} : \tau \quad \$ \quad \text{Heap, Stack}}$

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

(28) $$\frac{\mathrm{E}\ (\text{varid}) \ = \ \forall \vec{\alpha}\vec{\ell}.\tau}{\mathrm{E} \overset{\text{space}}{\vdash} \text{varid} : \tau[\tau_i/\alpha_i,\ z_j/\ell_j] \quad \$ \quad 0,\ 1}$$

(29) $$\frac{\begin{array}{c}\mathrm{E}\ (\text{funid}) \ = \ \forall \vec{\alpha}\vec{\ell}.\tau \\ \tau[\tau_i/\alpha_i,\ z_j/\ell_j] \ = \ \tau'_1 \xrightarrow{h_1;\ s_1} \ldots \xrightarrow{h_{n-1};\ s_{n-1}} \tau'_n \xrightarrow{h;\ s} \tau'_{n+1} \qquad \tau'_{n+1} \trianglelefteq \tau' \\ \forall i.\ 1 \leq i \leq n, \ \ \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_i : \tau''_i \quad \$ \quad h'_i, s'_i \quad \tau''_i \trianglelefteq \tau'_i \\ h' \ = \ \sum_{i=1}^{n-1} h_i + h \qquad s' \ = \ \max\left(\max_{i=1}^{n-1} s_i, s\right)\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} \text{funid } \mathit{expr}_1 \ \ldots \ \mathit{expr}_n : \tau' \\ \$ \quad \sum_{i=1}^{n} h'_i + h', \max\left(\max_{i=0}^{n-1} s'_{n-i}+i, \ \ n+s'\right)\end{array}}$$

(30) $$\frac{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_1 : \text{bool} \quad \$ \quad h_1, s_1 \\ \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_2 : \tau_2 \quad \$ \quad h_2, s_1 \qquad \mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr}_3 : \tau_3 \quad \$ \quad h_3, s_3 \\ \tau_2 \trianglelefteq \tau \qquad \tau_3 \trianglelefteq \tau\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{space}}{\vdash} \texttt{if } \mathit{expr}_1 \texttt{ then } \mathit{expr}_2 \texttt{ else } \mathit{expr}_3 : \tau \\ \$ \quad h_1 + max(h_2, h_3), \ \ max(s_1, s_2, s_3)\end{array}}$$

$\boxed{\mathrm{E} \overset{\text{decl}}{\vdash} \text{decl } \$ \ \mathrm{E}}$

(31) $$\frac{\begin{array}{c}\forall i.\ 1 \leq i \leq n, \forall j.\ 1 \leq j \leq m, \ \ \mathrm{E} \overset{\text{pat}}{\vdash} \mathit{pat}_{ij} : \tau_j \quad \$ \quad sp_{ij} \quad \Rightarrow \quad \mathrm{E}_i \\ \forall i.\ 1 \leq i \leq n, \ \ \mathrm{E} \cup \mathrm{E}_i \overset{\text{space}}{\vdash} \mathit{expr}_i : \tau \quad \$ \quad h_i, s_i \\ h \ = \ \max_{i=1}^{n} \ h_i \qquad s \ = \ \max_{i=1}^{n} \ \left(s_i + \max_{j=1}^{m}(sp_{ij} + j - 1)\right) \\ \tau' \ = \ (\tau_1 \xrightarrow{\mathcal{H}_{clos};\ 1} \ldots \xrightarrow{\mathcal{H}_{clos}+(m-1);\ 1} \tau_m \xrightarrow{h;\ s} \tau) \\ \vec{\alpha} \ = \ \mathrm{TV}\ (\ \tau'\ ) \qquad \vec{\ell} \ = \ \mathrm{ZV}\ (\ \tau'\ )\end{array}}{\begin{array}{c}\mathrm{E} \overset{\text{decl}}{\vdash} \text{funid } \mathit{pat}_{11} \ \ldots \ \mathit{pat}_{1m} = \mathit{expr}_1 \mid \ldots \mid \text{funid } \mathit{pat}_{n1} \ \ldots \ \mathit{pat}_{nm} = \mathit{expr}_n \\ \$ \ \{\ \text{funid} : \forall \vec{\alpha}\vec{\ell}.\tau'\ \}\end{array}}$$

Fig. 16. Stack and Heap cost axioms for Template-Hume expressions and function definitions

$$\boxed{\mathrm{E} \overset{\text{space}}{\vdash} \mathit{expr} \ : \ \tau \quad \$ \quad \mathrm{Heap}, \mathrm{Stack}}$$

$$\forall i.\ 1 \leq i \leq n, \forall j.\ 1 \leq j \leq m, \quad \mathrm{E} \overset{\text{pat}}{\vdash} \mathit{pat}_{ij} \ : \ \tau_j \quad \$ \quad sp_{ij} \quad \Rightarrow \quad \mathrm{E}_i$$
$$\forall i.\ 1 \leq i \leq n, \quad \mathrm{E} \ \cup \ \mathrm{E}_i \ \cup \ \{ \ \mathrm{funid} : \tau \ \} \overset{\text{space}}{\vdash} \mathit{expr}_i \ : \ \tau[k+1/k] \quad \$ \quad h_i, s_i$$
$$\tau[\varepsilon/k] = \mathbf{U}$$

$$h \ = \ \max_{i=1}^{n} \ h_i \qquad s \ = \ \max_{i=1}^{n} \ \left(s_i + \max_{j=1}^{m}(sp_{ij} + j - 1)\right)$$
$$\tau' \ = \ (\tau_1 \xrightarrow{\mathcal{H}_{clos};\ 1} \ \ldots \ \xrightarrow{\mathcal{H}_{clos}+(m-1);\ 1} \tau_m \xrightarrow{h;\ s} \tau)$$
$$\vec{\alpha} \ = \ \mathrm{TV} \ (\ \tau' \ ) \qquad \vec{\ell} \ = \ \mathrm{ZV} \ (\ \tau' \ )$$

(32)
$$\frac{}{\mathrm{E} \overset{\text{decl}}{\vdash} \mathrm{funid} \ \mathit{pat}_{11} \ \ldots \ \mathit{pat}_{1m} = \mathit{expr}_1 \mid \ldots \mid \mathrm{funid} \ \mathit{pat}_{n1} \ \ldots \ \mathit{pat}_{nm} = \mathit{expr}_n \quad \$ \ \{ \ \mathrm{funid} : \forall \vec{\alpha}\vec{\ell}.\tau' \ \}}$$

Fig. 17. Cost axiom for PR-Hume Function Definitions

in a series of four refactoring *stages*: stage 1 – PR-Hume; stage 2 – Template-Hume; stage 3 – FSM-Hume; and stage 4 – HW-Hume. In this way, we will have demonstrated how PR-Hume programs may be implemented in HW-Hume. We will assume that all recursion is statically bounded (as required to construct a static cost model). Although we do not consider Hume metaprograms in this paper, metaprogramming constructs as described in [Hammond and Michaelson 2003] could easily be added as a stage 0, if required.

Firstly, PR-Hume programs may be refactored to Template-Hume programs by:

—expanding calls to recursive functions, either by inlining the function call, or by introducing a new non-recursive function definition for each expansion;

—translating operations on inductively defined data structures, such as trees, either into operations on fixed size structures such as vectors or into streams at the box level (thereby separating finite *data* from potentially infinite *codata*).

While it is not necessary to refactor predefined higher-order functions or data structures involving predefined types in order to obtain a Template-Hume program, it is convenient to do this at this stage to avoid repeating similar refactorings for user-defined and predefined structures.

This refactoring will generally result in a much larger, and perhaps less efficient, program than the original PR-Hume program. It is, however, guaranteed to be finite, since we know that, by definition, all primitive recursive programs must terminate. We can determine *bounds* on the size of the resulting program from the size analysis for PR-Hume that was presented in Section 3.6. A similar approach is taken in Lustre/SCADE [Caspi et al. 1987], where so-called *static recursion* is eliminated through expansion.

Subsequently, Template-Hume programs may be refactored to FSM-Hume programs by:

—replicating polymorphic definitions;

—inline expansion of higher-order function definitions.

Again, these translations will generally result in less compact FSM-Hume code. Time and space efficiency may, however, both be improved since it is no longer necessary to implement function closures, and all data values may be *unboxed* [Peyton Jones 1992]. Similar transformations are often applied to allow simpler program analyses to be applied, or following type checking to produce code that is specialized to the types that are actually used in the program.

Finally, FSM-Hume programs may be translated to HW-Hume programs by:

(1) replacing all higher-level types with equivalent bit/tuple/vector representations
(2) expanding all function calls to become case-expressions;
(3) applying copy-propagation to variables defined in let-expressions;
(4) lifting case-expressions/conditionals either to the top-level of the enclosing box or to form new boxes.

The resulting HW-Hume code will be less compact, and perhaps less efficient: it is no longer possible to use the stack to reuse space for nested function calls, for instance. However, it may now be possible to implement the code using FPGAs or similar circuitry, and space is no longer wasted in stack frame overhead.

4.2 Example: bit sequence conversion

We will now show how such a refactoring can be applied to a simple example by successively refining our initial source from PR-Hume through FSM-Hume to HW-Hume. Consider converting a bit sequence to a byte and checking its parity. Suppose we are given:

```
type Bit =  word 1;
type Byte = word 8;

data PARITY = ODD | EVEN;

flip ODD =  EVEN;
flip EVEN = ODD;
```

where `flip` changes `ODD` to `EVEN` and vice versa.

The classic functional programming solution is based on using recursion along a list of bits to accumulate the parity and final byte value:

```
check1 :: ([Bit],PARITY,Byte) -> (bool,Byte);
check1 ([],    parity, byte) = (parity==EVEN,byte);
check1 (bit:t, parity, byte) = check1 (t,flip parity,2*byte+bit);

expression check1 ([1,0,1,0,1,1,0,0],EVEN,0);
```

When the list is empty a tuple is returned comprising a boolean, to indicate whether or not the parity is even, paired with the final byte value `byte`. For a non-empty list, the function recurses on the tail of the list with a flipped parity and the bit value added to the accumulated byte value, which is first shifted left.

4.3 From PR-Hume to FSM-Hume

`check1` is a primitive recursive tail recursion of the form:

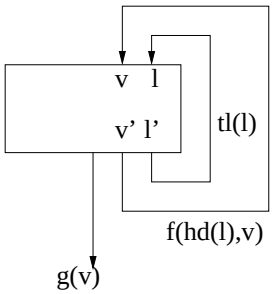

Fig. 18. Tail recursion as iterative box.

tailrec (f,g,v,l) =
if l = [] then return g(v)
else return tailrec (f,g,f(hd(l),v),tl(l))

As recursion progresses over the list l, a partial result is accumulated in v. At each stage of the recursion, if the list is not empty then f is applied to a pair containing the head of the list and the current partial result, v, to form the partial result for the next stage. If the list is empty, then g is applied to v to give the final result. This has the well known iterative equivalent:

iterate (f,g,v,l) =
while l != [] do { v := f(hd(l),v); l := tl(l) }
return g(v)

Here, v and l are now imperative variables which are updated at each iteration of the *while*-loop. While Hume does not have imperative variables, single values may be held on wires between box cycles. Hence, we may use this equivalence to convert a recursive function over the list `l` with the accumulating variable `v` to a box with feedback wires for `l` and `v`, as shown in Figure 18. For our example, the resulting Hume program is:

```
box check2
in  (l ::[Bit], p ::PARITY, byte ::Byte)
out (l'::[Bit], p'::PARITY, byte'::Byte,result::(bool,Byte))
match
  ([],   parity,byte) -> (*,*,*,(parity==EVEN,byte))
| (bit:t,parity,byte) -> (t,flip parity,2*byte+bit,*) ;

stream output2 to "some_sink";

wire check2.l'    to check3.l    initially [1,0,1,0,1,1,0,0];
wire check3.p'    to check3.p    initially EVEN;
wire check3.byte' to check3.byte initially 0;

wire check2.result to output2;
```

where the feedback wires `p` for the parity and `byte` for the accumulated byte value are equivalent to v in the schema.

Note that while this form is no longer recursive, there is now an arbitrary sized list on the feedback wire for `l`. To complete the transition to FSM-Hume, we must replace the feedback wire for `l` with an external input from a stream of bits. We now need to keep count of the number of bits that have been input from the stream so that bytes may be output at appropriate intervals, so we introduce a new feedback wire from `c` to `c'` whose value is initially 1:

```
type integer = int 32;

box check3
in  (c ::integer, b::Bit, p ::PARITY, byte ::Byte)
out (c'::integer,         p'::PARITY, byte'::Byte, result::(bool,Byte))
match
  (9,*,  parity,byte) -> (1,   EVEN, 0, (parity==EVEN,byte))
| (c,bit,parity,byte) -> (c+1, flip parity, 2*byte+bit, * );

...

stream input3  from "some_source";
stream output3 to   "some_sink";

wire input3        to check3.b;
wire check3.result to output3;
```

4.4 From FSM-Hume to HW-Hume

Our FSM-Hume program is recursion-free but still uses types such as integers. For HW-Hume, we need to systematically replace these types with equivalent low-level representations based on bits and bit tuples. First, we replace `byte` with an eight-bit tuple, with the associated zero value (`zeroT`) and an update function (`update`):

```
type ByteT = (Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit);

zeroT = (0,0,0,0,0,0,0,0);

update :: ByteT -> integer -> Bit -> ByteT;
update (b1,b2,b3,b4,b5,b6,b7,b8) 1 bit = (bit,b2,b3,b4,b5,b6,b7,b8);
...
update (b1,b2,b3,b4,b5,b6,b7,b8) 8 bit = (b1,b2,b3,b4,b5,b6,b7,bit);
```

Next we replace `integer`, which is only used for counting from one to nine, with a four-bit tuple, plus a definition for the associated value one (`oneC`), and an increment function (`inc`):

```
type Count = (Bit,Bit,Bit,Bit);

oneC = (0,0,0,1);

inc :: Count -> Count;
inc (0,0,0,1) = (0,0,1,0);
```

```
...
inc (1,0,0,0) = (1,0,0,1);
```

`update` is then changed to use the new `integer` representation:

```
update :: ByteT -> Count -> Bit -> ByteT
update (b1,b2,b3,b4,b5,b6,b7,b8) (0,0,0,1) bit = (bit,b2,b3,b4,b5,b6,b7,b8);
...
update (b1,b2,b3,b4,b5,b6,b7,b8) (1,0,0,1) bit = (b1,b2,b3,b4,b5,b6,b7,bit);
```

We also replace `PARITY` with an equivalent bit representation, using `1` for `EVEN` and `0` for `ODD`:

```
flip 0 = 1;
flip 1 = 0;
```

This gives us:

```
box check4
in  (c ::Count, b::Bit,p ::Bit, byte::ByteT)
out (c'::Count,        p'::Bit, byte'::ByteT, result::(bool,ByteT))
match
  ((1,0,0,1),*,parity,byte) -> (oneC,1,zeroT,(parity==1,byte))
| (c,bit,parity,byte) ->       (inc c,flip parity,update byte c bit,*);

...

wire input4        to check4.b;
wire check4.result to output4;
```

Note that formally this is still not HW-Hume as we employ a boolean result, auxiliary functions `update`, `inc` and `flip`, and some auxiliary constant definitions, `oneC` etc. To complete the refinement, we need to replace these with their in-line bit-based equivalents and simplify the result:

```
box check5
in  (c::Count,  b::Bit,  p:: Bit, byte:: ByteT)
out (c'::Count,          p'::Bit, byte'::ByteT, result::(Bit,ByteT))
match
  ((1,0,0,1), *,   parity, byte) ->
       ((0,0,0,1), 1, (0,0,0,0,0,0,0,0),         (parity,byte))
| ((0,0,0,1), bit, 1, (b1,b2,b3,b4,b5,b6,b7,b8)) ->
       ((0,0,1,0), 0, (bit,b2,b3,b4,b5,b6,b7,b8), *)
| ((0,0,0,1), bit, 0, (b1,b2,b3,b4,b5,b6,b7,b8)) ->
       ((0,0,1,0), 1, (bit,b2,b3,b4,b5,b6,b7,b8), *)
| ((0,0,1,0), bit, 1, (b1,b2,b3,b4,b5,b6,b7,b8)) ->
       ((0,0,1,1), 0, (b1,bit,b3,b4,b5,b6,b7,b8), *)
| ((0,0,1,0), bit, 0, (b1,b2,b3,b4,b5,b6,b7,b8)) ->
       ((0,0,1,1), 1, (b1,bit,b3,b4,b5,b6,b7,b8), *)
 ...
```

Variant	Hume Level	Box Costs		Wire Costs	Function Costs		Totals	% diff
		Heap	Stack		Heap	Stack		
check1	PR	–	–	–	190 (191)	84 (87)	274 (278)	1.5%
check2	PR	80 (89)	19 (19)	64(64)	–	–	163 (172)	5.6%
check3	FSM	38 (39)	20 (23)	9 (9)	–	–	67 (71)	6.0%
check4	(FSM)	107 (108)	37 (53)	44 (44)	–	–	188 (205)	9.0%
check5	HW	*2 (2)*	– –	*1 (1)*	–	–	3 (3)	0

Fig. 19. Actual costs and guaranteed upper bounds (in parentheses) in terms of 4-byte words of memory for each refactoring variant. For the HW-Hume example, `check5` required 36 bits of registers/latches and the wires required 14 bits.

4.5 Costs for each Refactoring Variant

Figure 19 gives the guaranteed upper bound heap and stack costs for each of the variants we have described above, comparing these with actual costs for a sample execution under our `pHAM` implementation. Here, the upper bound costs for the first two variants are obtained from our prototype analysis for PR-Hume programs; the remaining costs are obtained from the analysis for FSM-Hume that is built in to the `pHAM` compiler. All figures are in terms of four-byte words of memory – the base unit of memory usage in the current implementations.

The FSM-Hume and PR-Hume variants (`check1`–`check4`) have total dynamic memory requirements varying between 67 and 274 words, with the corresponding bounds varying between 71 and 278 words. The HW-Hume variant (`check5`) shows particularly good memory usage (50 bits) due to the ability to compile the program to a minimal bit-oriented representation.

Note that the actual costs shown in Figure 19 simply represent the measured costs for a single execution of the program. This will normally be lower than any bound that could be inferred statically, even by hand, since some execution paths will be not taken in a given run; and some wire buffers may not be completely filled. It would not, of course, be safe to reduce memory usage to the measured values, since some execution path that has not been tested here might require additional memory.

The results for the PR-Hume variants are particularly interesting because of the relative complexity of the analysis. For `check1`, the recurrences we obtain are:

$$\begin{aligned} s(z) &= 8 + max(1 + s(z-1), 7) \\ h(z) &= max(14 + h(z-1), 9) \end{aligned}$$

and the solutions that are found are:

$$\begin{aligned} s &\geq 15 + z \times 9 \\ h &\geq 9 + z \times 14 \end{aligned}$$

where z is the size of the list argument and the other figures are constants derived from the abstract machine costs. For `check2`, the analysis accurately predicts the

bound on the size of the input list to be 8 and the bound on the size of the result list to be 7.

Actual heap costs are within one word of the guaranteed bound for three variants (`check1`, `check3` and `check4`), but there is greater variation on stack usage (between 3 and 16 words). Conversely, for `check2`, the stack bound is identical to the actual usage, but the heap bound is 9 words greater.

It is worth reiterating that the bounds we obtain are not engineering estimates (however good), but are limits on memory usage which are guaranteed *never* to be exceeded. It is also worth emphasing that even though the bounds are not always identical to the recorded costs; the differences between guaranteed bounds and actual usage are very small in both absolute and relative terms. In the worst case above (`check4`) our guaranteed bounds are precise to 9% of actual usage (17 words of heap), with the program needing 284 bytes of dynamic memory; and in the best case (`check1`) the bounds are precise to within 1.5% of actual usage (4 words of heap), with the program requiring 1112 bytes of dynamic memory. The guaranteed bounds for HW-Hume programs are, of course, 100% precise. These results have been confirmed by other examples we have tried: we have run the analyses on a large number of other test programs with very similar results [Vasconcelos 2006].

5. RELATED WORK

5.1 Adapting General Purpose Languages for Embedded Systems Programming

While general purpose languages such as C/C++, Ada or Java have found widespread use in real-time systems engineering, the absence of strong guarantees about dynamic behaviour has motivated the development of specialised versions which can provide such guarantees.

Two extreme approaches to real-time language design are exemplified by SPARK Ada [Barnes 1997] and the real-time specification for Java (RTSJ) [Bollela and et al. 2000]. SPARK Ada epitomises the idea of language design by elimination of unwanted behaviour from a general-purpose language. This includes the notion of concurrency, a highly desirable feature in a real-time reactive system. The remaining program behaviour is, however, guaranteed by strong formal models. In contrast, RTSJ provides specialised runtime and library support for real-time systems work, but makes no absolute performance guarantees. Thus, SPARK Ada provides a minimal, highly controlled environment for real-time programming emphasising *correctness by construction* [Amey 2002], whilst Real-Time Java provides a much more expressible, but less controlled environment, and one without formal behavioural guarantees. Our objective with the Hume design is to maintain formal correctness whilst also providing high levels of expressibility, that is to provide advantages from both extremes, as far as possible.

5.2 Languages based on Finite-State Notations

As we have seen, finite-state approaches are attractive when dealing with certain kinds of real-time system, since they allow a system to be defined by composing small, easily costed components. Such approaches are seen as being especially appropriate for defining systems with strong control requirements, such as device drivers or protocol implementations. Pure approaches often, however, prove prob-

lematic when constructing complex programs: typically the finite-state-machines derived for such systems will have a large number of states, which can be difficult for the programmer to manage; moreover relatively small extensions can cause exponential growth in the number of states. A number of extended finite-state languages have therefore been proposed incorporating composition, communication, and data structures to give Turing-Complete notations. Many also incorporate quantitative notions of time. Three common examples are Estelle [Budkowski and Dembrinski 1987], an imperative language developed for OSI communications protocols; SDL [International Telecommunication Union 1999], a language similar to Estelle, which has a graphical dialect used as a design tool; and TTM [Ostroff 1990], a graphical notation similar to Petri nets, used to describe real-time discrete event processes. Unlike Hume, none of these notations has been built around strong layered cost models, as defined here.

5.3 Synchronous Dataflow Languages

In a synchronous dataflow language, every *action* (whether computation or communication) has a zero time duration. In practice this means that actions must complete before the arrival of the next event to be processed. Communication with the outside world occurs by reacting to external stimuli and by instantaneously emitting responses. The advantage of synchronous dataflow languages lies in providing a powerful and simple model of time costs, by eliminating the timing complexities associated with asynchronicity. Several languages have consequently applied this model to real-time systems control. For example, Signal [Gautier et al. 1987] and Lustre [Caspi et al. 1987] are similar declarative notations, built around the notion of timed sequences of values. Esterel [Boussinot and de Simone 1991] is an imperative notation that can be translated into finite state machines or hardware circuits, and Statecharts [Harel 1987] uses a visual notation, primarily for design.

One obvious deficiency of these notations is the lack of expressiveness, notably the absence of recursion and higher-order combinators, the emphasis on data flow rather than control flow, and the restriction to synchronous problems. While synchronous Kahn networks [Caspi and Pouzet 1996] do incorporate both higher-order functions and recursion, they consequently lose the strong guarantees of resource boundedness that motivate our work. Recent work has also explored the design of languages that incorporate finite-state ideas into an essentially synchronous language model. In this way, it is possible to program mixed systems, where control flows are captured using an finite-state approach and data flows are captured using synchronous dataflow. One recent example is Lucid Synchrone [Colaço et al. 2005], which allows Mealy and Moore state machines to be combined with normal dataflow equations, with timing information from the automata exposed in the underlying synchronous dataflow notation in the form of explicit delays. Lucid Synchrone also supports the use of higher-order functions and recursion. The primary differences from the Hume approach are in our separation of box level and expression level; in the inclusion of explicit delay operators, the treatment of automata as a class of time-dependent function in Lucid Synchrone, and the treatment of recursive functions as sequential streams; and in the explicit identification of varying language levels and their associated properties in Hume.

5.4 Functional Languages for Soft Real-Time Programming

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [Portillo et al. 2003a]. Hume is thus, as far as we are aware, unique both in being a practical language based on strong automatic cost models, and in being specifically designed to allow straightforward space- and time-bounded implementation for hard real-time systems, those systems where tight real-time and hard space guarantees must be met. A number of functional languages have, however, looked at *soft* real-time issues [Armstrong et al. 1993; Wallace and Runciman 1995; Wan et al. 2001], there has been work on using functional notations for hardware design (essentially at the HW-Hume level) [Mycroft and Sharp 2000; Claessen and Sheeran 2000; Matthews et al. 1998; Donadio 1999], including specialised notations for specifying FPGA circuits [Hawkins and Abdallah 2002] and DSP designs [Donadio 1999]. There has also been much recent theoretical interest both in the problems associated with costing functional languages [Portillo et al. 2003a; R.J.M. Hughes and Sabry 1996] and in bounding space/time usage [Hughes and Pareto 1999; Tofte and Talpin 1997; Hofmann 2000; Wan et al. 2001], including work on statically predicting heap and stack memory usage [Unnikrishnan et al. 2000].

The most widely used soft real-time functional language is the impure, strict language Erlang [Armstrong et al. 1993], a concurrent language with a similar design to Concurrent ML [Reppy 1991]. Erlang has been used by Ericsson to construct a number of successful telecommunications applications in the telephony sector [Blau and Rooth 1998], including a real-time database, Mnesia [Wikström and Nilsson 1996]. Erlang is concurrent, with a lightweight notion of a process. Unlike Hume boxes, Erlang processes are constructed using explicit spawn operations, with communication occurring through explicit *send* and *receive* operations to nominated processes.

5.5 Functional Languages with Guaranteed Termination

Several other approaches have focused on the use of restricted patterns of recursion to ensure program termination, though without our focus on determining actual executions costs. For example, [Burstall 1987] proposed the use of an extended *ind case* notation in a functional context, to define inductive case discrimination on data types. This notation is used to identify the component of an inductive data structure that is the target of the recursion. While *ind case* enables program termination to be determined statically, Burstall's examples suggest that considerable ingenuity can be required to recast terminating functions into this syntax. Turner's *elementary strong functional programming* [Turner 1995; 2004] has similarly explored issues of guaranteed termination in a purely functional programming language. Turner's approach separates finite data structures such as tuples (data) from potentially infinite structures such as streams (co-data), in a way that is analogous to the box/expression separation in Hume. In this way, it is possibly to define programs that can be syntactically identified as primitive recursive. However, as with Burstall's approach, definitions must be refactored into this new syntax. A good introduction to co-inductive types showing how they can used to verify the alternating-bit protocol can be found in [Giménez 1995]. Finally, McBride [McBride 2003] has studied the use of *structural induction* in a *dependently typed* language

similar to Epigram [McKinna 2003]. By exposing structural induction over inductive data types directly in the type system, which itself posseses strong termination properties, McBride is able to guarantee program termination provided only that the program is correctly typed. A primary advantage is that the expression language needs no special syntactic forms, other than some form of case analysis, to identify structural recursion. As with Embedded ML, it is necessary, however, to provide more precise type information in order to encode termination properties for verification.

5.6 Approaches to Bounding Space Usage

Compile-time garbage collection techniques attempt to statically eliminate some or all heap-based memory allocation. One approach that has recently found favour is the use of region types [Tofte and Talpin 1997], in which memory cells are tagged with a statically scoped allocation region. When the region is no longer required, all memory associated with that region may be freed without invoking a garbage collector. In non-recursive contexts, the memory may be allocated statically and freed following the last use of any variable that is allocated in the region. In a recursive context, this heap-based allocation can be replaced by (possibly unbounded) stack-based allocation.

In their proposal for Embedded ML, Hughes and Pareto [Hughes and Pareto 1999] combine region types with their earlier *sized type system* [R.J.M. Hughes and Sabry 1996] to give bounded space and guaranteed termination properties for a first-order strict, but recursive, functional language. Compared with Hume, Embedded ML is restricted in a number of ways: most notably in not supporting higher-order functions, and in requiring the programmer to specify detailed information on memory usage through type specifications.

Hofmann's linearly-typed functional programming language LFPL [Hofmann 2000] uses linear types to determine resource usage patterns. So-called *diamond* resource types are used to count constructors. First-order LFPL definitions can be computed in bounded space, even in the presence of general recursion. Hofmann has recently considered the extension of LFPL to higher-order functions with reference to *non size-increasing* recursive definitions on lists [Hofmann 2002], where the size of all intermediate computations is bounded by the size of the inputs. Where definitions are restricted to primitive recursion only, this then guarantees polynomial size complexity. Unfortunately, for arbitrary higher-order functions, the cost of introducing closures means that an unbounded stack is required. More recently, Hofmann and Jost have introduced automatic inference of their resource types [Hofmann and Jost 2003; 2006], and thus of heap-space consumption, using an amortized cost model built on a difference metric similar to that of Crary and Weirich [Crary and Weirich 2000] and avoiding linearity restrictions. We are presently working on an analysis that will combine amortized costs with sized types in order to produce an efficient time and space cost analysis for Hume. We are also exploring how such patterns of resource usage can be captured through dependent types that will allow complex properties to be expressed and automatically verified [Brady and Hammond 2005].

Finally, Camelot and Grail [Mackenzie and Wolverson 2004] use a *proof carrying code* approach that allows formal properties of resource usage to be expressed in the form of easily checked certificates. Camelot is a resource-aware functional pro-

gramming language that can be compiled to a subset of JVM bytecodes; Grail is a functional abstraction over these bytecodes. This abstraction possesses a formal operational semantics that allows the construction of a program logic capable of capturing program behaviours such as time and space usage [Aspinall et al. 2004]. The objective of the work is to synthesise proofs of resource bounds in the Isabelle theorem prover, and to attach these proofs to mobile code in the form of more easily verifiable proof derivations. In this way the recipient of a piece of mobile code can cheaply and easily verify its resource requirements.

6. CONCLUSIONS AND FURTHER WORK

There is great interest in the real-time embedded systems community in developing high level programming notations that can still be effectively costed. In this way, software engineering advantages can be obtained in the form of lower development costs, reduced time to market, software reuse and increased portability without sacrificing essential space and time properties. Hume provides a framework to explore these issues, and to permit the development of effective cost models in a relatively clean, but still practical, experimental setting.

By structuring Hume as a series of increasingly expressive levels, we have been able to provide determinate space cost models for each level as an extension of the model for the previous level. We have also shown how Hume programs in one level may be staged into the next lower level, albeit with a possible increase in space usage. In this way, programs may be written at an appropriate level of abstraction for the problem, or generic libraries exploited in low levels of Hume code, without losing required implementation properties

Moreover, since the translations are formally specified, and the cost models may be queried by external tools, the semantic gap between specification and implementation has been significantly reduced compared with more conventional approaches. We conjecture, but have not demonstrated, that there are very few Full-Hume programs that are interesting in practice, but which cannot be transformed easily into PR-Hume or simpler programs.

6.1 Limitations and Further Work

Extended cost models and analyses. This paper has studied heap and stack memory allocation as an exemplar of how formal cost models can be constructed, and with a view to constructing the bounded space implementations of Hume described above. We have not yet studied the equally important issues of garbage collection, time usage or power consumption. By focusing on stack usage and heap allocations, we have, however, been able to develop models and analyses in an essentially system-independent framework, and so to concentrate on key structural issues affecting the models and analyses.

With funding from the European Union and the UK's Ministry of Defence we are now investigating how our models and analyses may be adapted to include the vitally important time information. Our approach is based on a hybrid analysis, where high-level information on loop boundaries, function calls and conditions as obtained here is interfaced with a commercial analysis tool that exploits a complex and precise formal model of processor and cache behaviour [Kästner 2003]. We are also investigating whether the model of linear space usage developed by Hofmann

and Jost [Hofmann 2000; Hofmann and Jost 2003] can be combined with our models to give a fast analysis for linear time/space programs but with the ability to also cost non-linear programs.

Enhanced box structuring. Hume incorporates high-level metaprogramming features as an essential part of the design. Box templates and wiring macros have proved to be good abstraction mechanisms for dealing with larger or more complex applications. Our metaprogramming constructs allow the encapsulation of repetitive wiring and box definitions, provide parametric polymorphism at the box level, and allow abstraction over manifest constants defining the structure of the process network etc. We have shown how metaprogramming constructs can be staged into normal Hume programs. We are now investigating structured forms of template (a "box calculus") to capture higher levels of process and wiring abstraction. This will allow us to capture common patterns of box composition, permitting the construction of *libraries* of box constructors etc.

Verification of temporal properties. There has recently been renewed interest in automated software verification in both academia and industry. Verification is especially useful for concurrent systems and other situations where timing properties are involved, and there are obvious benefits in the Hume setting. Mechanized abstraction [Baudin et al. 2002] coupled with theorem proving [Ellis and Ireland 2004; Flanagan et al. 2002; Flanagan and Leino 2001], model checking [Regan and Hamilton 2004; Corbett et al. 2000; Holzmann and Smith 2000] or hybrids [Ball and Rajamani 2002] can be applied to Hume programs in order to verify important temporal properties such as liveness or absence of deadlocks. We have integrated a temporal specification language into HW-Hume, translated the HW-Hume semantics into the Promela notation used by the Spin model checker, and applied this to several examples, including one large one, where we have successfully shown the presence and identified the cause of a livelock situation[Grov et al. 2004]. We are now investigating more sophisticated properties including box timings, scheduling and relative box rates, and considering how this model can be extended to higher levels of Hume program. Where space/time properties are concerned, this extension will build on the cost models described in this paper.

Measuring Productivity. Finally, while we believe that the expressivity offered by Hume leads to productivity gains, we have not attempted to quantify these in this paper. Although considerable effort would obviously be required to obtain sound experimental results, this is an issue that would repay careful investigation.

Clearly, the Hume approach of boxes with implicit execution cycles and communication reduces code size for concurrent programs over more conventional approaches such as Java `Threads` or Ada tasks. Equally clearly, both complexity and error rates are reduced by our encapsulation of asynchronous communication and abstraction into FSM-based boxes. However, it is possible that the Hume model might make certain kinds of communication pattern more difficult to express, and this must be investigated in the light of real applications such as the real-time computer vision applications we are developing in the Framework VI EmBounded project.

Our initial measurements of code size (a crude, but accepted metric of both productivity and complexity) show that Hume programs are about a quarter the size of

the equivalent threaded Java programs or about a third the size of the equivalent concurrent Ada programs. About 60% of the Hume code represents type declarations and wiring specifications (so about 40% represents the actual algorithm), where about 80% of both the Ada and Java programs is devoted to algorithmic code. While these figures are consistent with claims made for other functional languages, such as Erlang [Wiger 2001], more work, based on systematically comparing program sizes for a number of realistic examples, is necessary to verify and refine any such productivity claims. We anticipate undertaking such studies as part of our ongoing research projects.

REFERENCES

Ackermann, W. 1928. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen 99*, 118–133.

Amey, P. 2002. Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, 24–28.

Arkoudas, K. and McAllester, D. 1996. Walther Recursion. In *Proc. CADE 13*. LNCS 1104. Springer Verlag, 643–657. provides a decision procedure. Walther recursion can be transformed to primitive recursion.

Armstrong, J. 2005. Personal Communication.

Armstrong, J., Virding, S., and Williams, M. 1993. *Concurrent Programming in Erlang*. Prentice-Hall.

Aspinall, D., Beringer, L., Hofmann, M., and Loidl, H.-W. 2004. A Resource-Aware Program Logic for a JVM-like Language. In *Trends in Functional Programming, Volume 4*. Intellect.

Augustsson, L. 1987. Compiling Lazy Functional Languages, Part II. Ph.D. thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden.

Ball, T. and Rajamani, S. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. POPL'02: 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. Portland, Oregon, 1–3.

Barnes, J. 1997. *High Integrity Ada: the Spark Approach*. Addison-Wesley.

Baudin, P., Pacalet, A., Raguideau, J., Schoen, D., and Williams, N. 2002. CAVEAT: A Tool for Software Validation. In *Proc. IEEE Conf. on Dependable Systems and Networks (DSN02)*. IEEE Computer Society.

Blau, S. and Rooth, J. 1998. AXD-301: a New Generation ATM Switching System. *Ericsson Review 1*.

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

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

Brady, E. and Hammond, K. 2005. A Dependently Typed Framework for Static Analysis of Program Execution Costs. In *Proc. Implementation and Applications of Functional Language, 2005*. Springer-Verlag LNCS. Springer-Verlag, Dublin. to appear.

Budkowski, S. and Dembrinski, P. 1987. An Introduction to Estelle: a Specification Language for Distributed Systems. *Computer Networks and ISDN Systems 4*, 3–23.

Burstall, R. 1987. Inductively Defined Functions in Functional Programming Languages. Tech. Rep. ECS-LFCS-87-25, Univ. of Edinburgh. April 1987.

Caspi, P., Pilaud, D., Halbwachs, N., and Place, J. 1987. Lustre: a Declarative Language for Programming Synchronous Systems. In *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL '87), München, Germany*.

Caspi, P. and Pouzet, M. 1996. Synchronous Kahn Networks. *ACM SIGPLAN Notices 31,* 6, 226–238.

Claessen, K. and Sheeran, M. 2000. A Tutorial on Lava: a Hardware Description and Verification System. Tech. rep., Chalmers University of Technology, Sweden. August 2000.

Colaço, J.-L., Pagano, B., and Pouzet, M. 2005. 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.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. 2000. Bandera: Extracting Finite-State Models from Java Source Code. In *International Conference on Software Engineering*. 439–448.

Cousineau, G. and Mauny, M. 1998. *The Functional Approach to Programming*. Cambridge University Press.

Crary, K. and Weirich, S. 2000. Resource Bound Certification. In *Proc. POPL '00: 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 184–198.

Donadio, M. 1999. Functional DSP, `http://users.snip.net/~donadio/mpd-hs-dsp.html`.

Ellis, B. and Ireland, A. 2004. An Integration of Program Analysis and Automated Theorem Proving. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM-04)*, E. Boiten, J. Derrick, and G. Smith, Eds. LNCS 2999. Springer Verlag, 67–86.

Flanagan, C. and Leino, K. R. M. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proc. Formal Methods Europe (FME 2001)*. LNCS 2021, Springer-Verlag.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J., and Stata, R. 2002. Extended Static Checking for Java. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI 2002)*.

Gautier, T., Guernic, P. L., and Besnard, L. 1987. SIGNAL: A Declarative Language For Synchronous Programming of Real-Time Systems. In *Proc. FPCA '87: Conf. on Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Lect Notes in Computer Science, vol. 274. Springer-Verlag, 257–277.

Giménez, E. 1995. An Application of Co-Inductive Types in Coq: Verification of the Alternating Bit Protocol. In *Proc. 1995 Workshop on Types for Proofs and Programs*. LNCS, vol. 1158. Springer-Verlag, 135–152.

Grov, G., Ireland, A., and Michaelson, G. 2004. Model Checking HW-Hume. In *Draft Proceedings of 5th Symposium on Trends in Functional Programming, Ludwig-Maximillian's Universitat, Munich, Germany*, H.-W. Loidl, Ed. 33–48.

Hammond, K. 2006. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *Central European Summer School on Functional Programming, July 2005*. Spinger-Verlag LNCS, to appear.

Hammond, K. and Michaelson, G. 2003. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*. Lecture Notes in Computer Science. Springer-Verlag.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8,* 3 (June), 231–274.

Hawkins, J. and Abdallah, A. 2002. Behavioural Synthesis of a Parallel Hardware JPEG Decoder from a Functional Specification. In *Proc. EuroPar 2002*.

Hofmann, M. 2000. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing 7,* 4, 258–289.

Hofmann, M. 2002. The strength of non size-increasing computation. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*. 258–289.

Hofmann, M. and Jost, S. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*. ACM Press, New Orleans, LA, USA.

Hofmann, M. and Jost, S. 2006. Type-Based Amortised Heap-Space Analysis. In *Proc. ESOP 2006: 2006 European Symposium on Programming*. 22–37.

Holzmann, G. J. and Smith, M. H. 2000. Automating Software Feature Verification. *Bell Labs Technical Journal 5,* 2 (Apr.-June), 72–87. Issue on Software Complexity.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Hughes, R. and Pareto, L. 1999. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*. 70–81.

International Telecommunication Union. 1999. [Z.100] Recommendation Z.100 (11/99) – Specification and description language (SDL).

Jay, C. 2005. Shaping distributions. In *Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson, Eds. Springer-Verlag.

Jensen, K. 1989. Coloured Petri nets: A High Level Language for System Design and Analysis. In *Proc. 10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989*. 342–416.

Johnsson, T. 1984. Efficient Compilation of Lazy Evaluation. In *Proc. 1984 ACM SIGPLAN Symposium on Compiler Construction*. 58–69.

Kästner, D. 2003. TDL: a Hardware Description Language for Retargetable Postpass Optimisations and Analyses. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*. Springer-Verlag LNCS 2830, 18–36.

Mackenzie, K. and Wolverson, N. 2004. Camelot and Grail: Compiling a Resource-Aware Functional Language for the Java Virtual Machine. In *this book*.

Matthews, J., Launchbury, J., and Cook, B. 1998. Microprocessor Specification in Hawk. In *Proc. International Conference on Computer Science*.

McBride, C. 2003. First-Order Unification by Structural Recursion. *Journal of Functional Programming 13,* 6, 1061–1075.

McDermid, J. 1996. *Engineering Safety-Critical Systems*. Cambridge University Press, 217–245.

McKinna, J. 2003. First-Order Unification by Structural Recursion. *Journal of Functional Programming 13,* 6, 1061–1075.

Michaelson, G. 2000. Constraints on recursion in the Hume expression language. In *Proceedings of 12th International Workshop on Implementation of Functional Languages, RWTH, Aachen*, M. Mohnen and P. Koopman, Eds. Aachner Informtik-Berichte, 231–246.

Mycroft, A. and Sharp, R. 2000. A Statically Allocated Parallel Functional Language. *Automata, Languages and Programming*, 37–48.

Ostroff, J. 1990. A Logic for Real-Time Discrete Event Processes. *IEEE Control Magazine 10,* 2, 95–102.

Papadopoulos, G. M., Boughton, G. A., Greiner, R., and Beckerle, M. J. 1993. *T: Integrated Building Blocks for Parallel Computing. In *Proc. 1993 ACM/IEEE conference on Supercomputing*. ACM Press, New York, NY, USA, 624–635.

Peta, R. 1967. *Recursive Functions*. Academic Press.

Peyton Jones, S. 1992. Implementing Lazy Functional Languages on Stock Hardware: Spineless Tagless G-Machin. *Journal of Functional Programming*.

Portillo, A. R., Hammond, K., Loidl, H.-W., and Vasconcelos, P. 2003a. A Sized Time System for a Parallel Functional Language (Revised). In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*. Number 2670 in Lecture Notes in Computer Science. Springer-Verlag.

Portillo, A. R., Hammond, K., Loidl, H.-W., and Vasconcelos, P. 2003b. Cost Analysis using Automatic Size and Time Inference. In *Proc. IFL 2002 – Implementation of Functional Languages, Madrid, Spain*. LNCS 2670. Springer-Verlag.

Regan, P. and Hamilton, S. 2004. NASA's Mission Reliable. *IEEE Computer 37,* 1 (Jan.), 59–68.

Reppy, J. 1991. CML: a Higher-Order Concurrent Language. In *Proc. 1991 ACM Conf. on Prog. Lang. Design and Impl. (PLDI '91)*. 293–305.

R.J.M. Hughes, L. P. and Sabry, A. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL'96 — 1996 ACM Symp. on Principles of Programming Languages*. St. Petersburg Beach, FL.

Schrage, M., van Ijzendoorn, A., and van der Gaag, L. 2005. Haskell Ready to Dazzle the Real World. In *Proc. 2005 Haskell Workshop, Sept. 30, Tallinn, Estonia*. 17–26.

Tofte, M. and Talpin, J.-P. 1997. Region-Based Memory Management. *Information and Computation 132,* 2 (1 Feb.), 109–176.

Turner, D. 1995. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Funct. Prog. Langs. in Education — FPLE '95*. LNCS. Springer-Verlag.

Turner, D. 2004. Total Functional Programming. *Journal of Universal Computing 10,* 7, 751–768.

Unnikrishnan, L., Stoller, S., and Liu, Y. 2000. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Technical Report 538, Computer Science Dept, Indiana University. Apr.

Vasconcelos, P. 2006. Cost Inference and Analysis for Recursive Functional Programs. Ph.D. thesis, University of St Andrews. in preparation.

Wallace, M. and Runciman, C. 1995. Extending a Functional Programming System for Embedded Applications. *Software: Practice & Experience 25,* 1 (Jan.).

Wan, Z., Taha, W., and Hudak, P. 2001. Real-time FRP. In *Intl. Conf. on Functional Programming (ICFP '01)*. ACM, Florence, Italy.

Wiger, U. 2001. Four-Fold Increase in Productivity and Quality – Industrial-Strength Functional Programming in Telecom-Class Products. In *Proc. 2001 Workshop on Formal Design of Safety Critical Embedded Systems, Munich*.

Wikström, C. and Nilsson, H. 1996. Mnesia — an Industrial Database with Transactions, Distribution and a Logical Query Language. In *Proc. Intl. Symp. on Cooperative Database Systems for Advanced Applications*.

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach

KEVIN HAMMOND
University of St Andrews
GREG J. MICHAELSON
Heriot-Watt University
and
PEDRO B. VASCONCELOS
Universidade do Porto

ACM Transactions on Software Engineering and Methodology, Vol. X, No. X, XX XXXX, Pages 1–??.