Recursion, Iteration and Hume Scheduling

Greg Michaelson and Robert Pointon

School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Scotland, EH14 4AS
greg@macs.hw.ac.uk or
rpointon@macs.hw.ac.uk

Abstract

Converting programs from full or PR-Hume to FSM- or HW-Hume involves transforming expression recursion to box iteration. However, this can add considerable overheads through unnecessary scheduling of other boxes dependent on the iteration output. Here we explore how analysis of output behaviour can identify boxes which may be executed independently of normal super-step scheduling, without affecting program meaning, and minimising the impact of the recursion to iteration transformation.

1 INTRODUCTION

Hume[MK02] is a novel language that encapsulates a simple functional language within a finite state machine framework. A program consists of a collection of automata, termed *boxes*, which are *wire*d together, while the functional language is used to handle the availability and pattern matching at box input, and the production of box output.

A central motivation for Hume's development was to produce a language amenable to resource use analysis[HM03, MHJ04]. It is well known that precise static behavioural analyses cannot be developed for Turing complete languages, because of the undecidability of termination and equivalence. It is also well known that trying to program in languages whose expressiveness are restricted to enable precise analysis is unpalatable. Thus, we have conceived of Hume as a *multi-level* language where different levels have different decidability properties depending on the types and control constructs that are deployed, as shown in Table 1.

We propose the following methodology for program development:

write program in full Hume
until program costed or deemed uncostable
    apply static analyses
    if analyses break then
        transform offending construct to lower level

Level	Types/Constructs	Time/Space Analyses
HW-Hume	tuples of bits	precise costs
FSM-Hume	bounded types + conditions	accurate bounds
PR-Hume	unbounded types+ primitive recursion	weak bounds
full Hume	general recursion	undecidable

TABLE 1. Hume levels and cost properties.

2 FROM LINEAR RECURSION TO BOX ITERATION

Program transformation is central to our programming methodology. The greatest novelty lies in our ability to exploit transformations that move processing from weak costability at the expression level to stronger costability at the coordination level. Here, the key transformation is from expression recursion to box iteration, moving from full, PR- or HO- Hume to FSM-Hume.

In the well known transformation[Man74], linear recursion may be replaced by iteration:

```
linrec f g h x =
   if f x
      then return g x
      else return linrec f g h (h x)

⇔

iter f g h x =
   while not (f x)
      x := h x
   return g x
```

This transformation may be extended to primitive recursion, through the introduction of accumulation variables, and to higher order functions, through unfolding of HOF applications to form equivalent specialised functions, but these are not considered here.

In Hume, the iteration may be expressed by a box with a looped wire:

```
box iterbox
 in  (i::t1,iter::t3)
 out (o::t2,iter'::t3)
match
 (i,*) -> (*,i) |
 (*,iter) -> if f iter
             then (h iter,*)
```

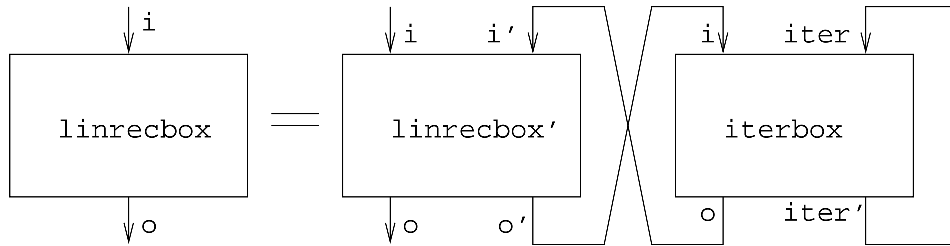

FIGURE 1. Recursion to box iteration.

```
           else (*,g iter);

wire iterbox (...,iterbox.iter')
             (...,iterbox.iter);
```

To be a valid Hume program the original `linrec` function must have been in a box. Therefore a box that directly calls a linear recursive function may be transformed to an equivalent two box form: the original box calls a new iterative box with a a looped wire to manage the state:

```
box linrecbox
 in  (i::t1)
 out (o::t2)
match
 (x) -> (...linrec f g h x...);

⇔

box linrecbox'
 in  (i::t1,i'::t2)
 out (o::t2,o'::t1)
match
 (i,*) -> (*,i) |
 (*,iter) -> (...iter...,*);

wire linrecbox' (...,iterbox.o)
                (...,iterbox.i);
wire iterbox (linrecbox'.o',iterbox.iter')
             (linrecbox'.i',iterbox.iter);
```

as shown schematically in Figure 1.

For example, considering multiplying two integers by repeated addition:

```
mult r x 0 = r;
mult r x y = mult (r+x) x (y-1);
```

embedded in a program to generate successive squares:

```
type integer = int 64;

box mult1
in (i::integer)
out (i'::integer,o::string)
match
 x -> (x+1, (mult 0 x x) as string++"\n");

stream output to "std_out";

wire mult1 (mult1.i' initially 0) (mult1.i,output);
```

Rewriting the function in curried form with an explicit condition:

```
mult (r,x,y) = if y==0 then r else mult (r+x,x,y-1)
```

we can identify:

```
f (r,x,y) = y=0
g (r,x,y) = (r+x,x,y-1)
h (r,x,y) = r
```

giving the transformed program:

```
box mult2
in (i::integer,iter'::integer)
out (i'::integer,iter::(integer,integer,integer),o::string)
match
(x,0) -> (x+1,(0,x,x),*) |
(x,r) -> (x+1,(0,x,x),r as string++"\n");

wire mult2 (mult2.i' initially 0,itermult.o initially 0)
           (mult2.i,itermult.i,output);

box itermult
in (i::(integer,integer,integer),
    iter::(integer,integer,integer))
out (o::integer,iter'::(integer,integer,integer))
match
 ((r,x,y),*) -> (*,(r,x,y)) |
 (*,(r,x,y)) -> if y==0
                then (r,*)
                else (*,(r+x,x,y-1));

wire itermult (mult2.iter,itermult.iter')
              (mult2.iter',itermult.iter);
```

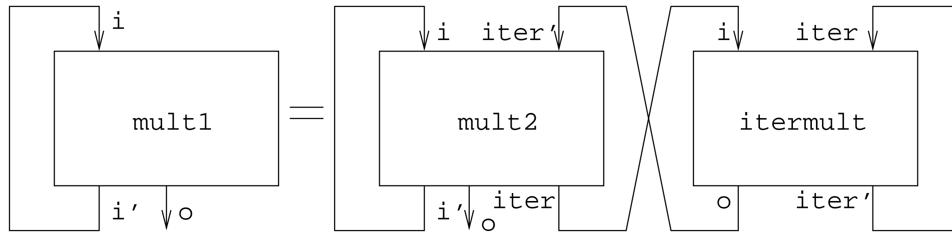

FIGURE 2. Recursive multiplication as iteration.

Version	Time (secs)
recursive (mult1)	100.4
iterative (mult2)	70.9

TABLE 2. Multiplication recursion and iteration times.

as shown schematically in Figure 2.

As well as making static analysis more tractable, this transformation also significantly reduces space requirements. For some recursive functions, it may be necessary to retain intermediate stack and heap space until the recursion terminates. In contrast, in the iterative form, space can be reclaimed on each iteration, with only the accumulating result retained on the feedback wire between iterations.

Table 2 shows the times to find the squares of the first 1411 integers using the Hume reference interpreter.

Essentially, around one million calls to the recursive function are replaced by around one million iterative box invocations. There is a net saving of around 30 seconds or 30% for the iterative boxes compared with the recursion.

3 HUME SCHEDULING

While the two Hume versions are equivalent in terms of the result they produce, they may differ in their temporal behaviour, depending on the chosen scheduling strategy in the implementation.

A Hume wire is a single value buffer that connects exactly two boxes. A Hume box can only run once per schedule super-step, it will try to match its input, and upon success consume from the input wires, evaluate some result, and then block until the output wires are free to accept the result. These semantics lead to deterministic programs, where non-deterministic programs can be constructed by specifying that a box is 'fair', and this enables fair (LRU) matching of the box input patterns.

At the end of each super-step cycle, a box may be in one of the following states:

1. **Runnable** The box has successfully consumed inputs and asserted outputs.

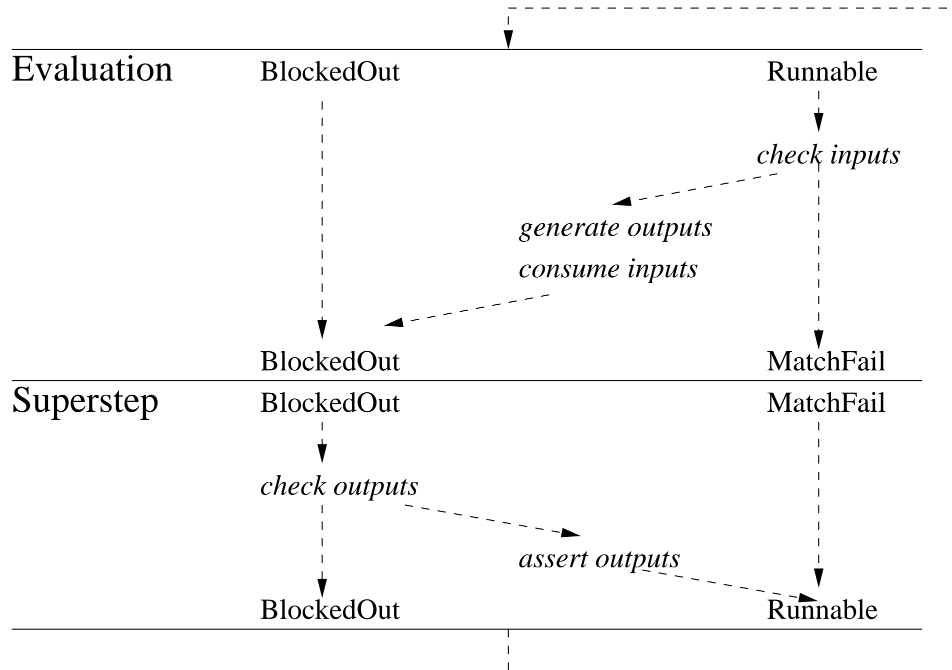

FIGURE 3. Execution cycle.

2. **Blocked-output** The box has successfully consumed inputs but failed to assert outputs. It will attempt to assert outputs on subsequent cycles.

3. **Match-fail** The box has failed to find required inputs.

The Match-fail state is equivalent to Runnable, but is distinguished to support program tracing and debugging. In what follows, we understand Runnable to include Match-fail.

The execution cycle is illustrated in Figure 3.

One scheduling implementation of this is lock-step scheduling where all non-blocked boxes are run once and then the inputs and updated with the new outputs.[1] Thus, the normal lock-step scheduling of Hume programs is as follows:

for ever
  execute each Runnable box
  super-step

To return to the recursion to iteration transformation, where `linrecbox` takes one schedule step to complete, `iterbox` will take many, depending on the original depth of recursion. In principle this is not problematic as repeated box iteration should have the same order of cost as recursion. However, because the base

[1]Due to the current lack of any data-flow, or temporal dependency analysis, lock-step scheduling is used in all valid Hume implementations.

Box	Runnable	Matchfail
mult1	1412	0
mult2	1412	998589
itermult	998589	1412

TABLE 3. Scheduling states.

scheduling mechanism always tries to run every box on each execution cycle, any boxes that depend directly or indirectly on the box iteration generating an output will be repeatedly scheduled but fail to consume inputs. Such boxes nonetheless incur a repeated unnecessary scheduling overhead.

Table 3 shows the overall scheduling behaviour of the recursive and iterative multiplication programs. Note that in this example boxes never enter the blocked output state.

In the recursive case, the single box `mult1` calls `fac` 1411 times. In the iterative case, the top level box `mult2` passes initial data to `itermult` 1411 times, with itermult failing to match inputs once prior to each communication. Thereafter, both `mult2` and `itermult` are scheduled 998,489 times, with `mult2` always failing to match inputs.

As well as being introduced by this transformation, the use of feedback wiring and iteration is a standard Hume programming construct. Here, it is common for many schedule steps to depend on the looped wire and not on any external input or output. Thus, while these conditions hold, such boxes can be scheduled repeatedly for greater efficiency. Note that changing the scheduling locally *may* change the temporal behaviour of the program and affect program meaning.

4 EFFICIENT SCHEDULING

There are at least two distinct approaches to improving the scheduling efficiency: one is to take a hierarchical view of the expression decomposition, and the other is to amend the scheduler for improved efficiency in particular cases.

4.1 Hierarchical Scheduling

By treating the decomposition of the expressions of a box into a set of boxes that are then 'nested' within the skeleton of the original box, then it is possible to use a hierarchical scheduling mechanism. The child boxes can read the input from skeleton (once input is matched), and then run using a nested instance of the Hume scheduler until they produce output for the skeleton. The scheduler within the skeleton box can only be active while the skeleton box itself is being executed.

for ever
    execute each Runnable box

```
    super-step

execute(box):
    if matched input then
        evaluate expression
        do output

execute(skeleton box):
    if matched input then
        while no output
            execute each Runnable child box
            super-step children
```

This approach preserves the super-step meaning within the original program and therefore program meaning.

We believe there is a way of transforming any nested box hierarchy into the usual flat representation. However, any resultant flat representation would have many boxes that would be idle the majority of the the time, and so efficiency would likely be lost.

Nesting of the scheduler is an overkill for what we are trying to accomplish and requires Hume programs to be decomposed with the special skeleton boxes. At present, we are trying to keep Hume stable and are resistant to feature bloat; therefore we would prefer not to add the machinery to support the skeleton boxes and nested scheduling.

4.2 Towards Staged Scheduling

By keeping the original scheduling approach but amending it to automatically schedule more efficiently in certain cases, we should gain benefits for free in many existing programs regardless of whether they contain boxes generated by expression decomposition or not.

In the usual execution of a box, a Runnable box may have either asserted outputs to other boxes and itself, or just to other boxes, or just to itself. If it has asserted outputs just to itself then it can have no impact on the ability of any other box to consume inputs. We say that such boxes have the $self-output$ property.

Thus, in principle, such boxes may execute repeatedly until they assert an output for another box, without affecting the overall outcome of program execution, provided there are no strong timing dependencies elsewhere in the program.

Let us add a fourth execution cycle state of **Self-output**. Then a first staged scheduling strategy might be:

```
for ever
    for next Self-output box
        until Self-output box is Runnable
            execute each Self-output box
```

Name	Method	Super-step	Starvation	Performance
Original	n/a	safe	possible	poor
Hierarchical	language	safe	possible	better
Staged One	implementation	unsafe	danger	best
Staged Two	implementation	unsafe	possible	better
Staged Three	implementation	unsafe	safe	ok

TABLE 4. Comparing approaches.

execute each Runnable box
super-step

This corresponds to executing an original expression recursion until it terminates. However, in a sequential Hume implementation, all boxes are blocked until such recursion has terminated and this effects persists for the equivalent iteration in the new strategy.

It might be more effective to interleave all Self-output boxes so that more boxes make local progress:

for ever
  until no box is Self-output
    execute each Self-output box
  execute each Runnable box
  super-step

The disadvantage of this approach is that all Runnable boxes are starved until all Self-output boxes become Runnable.

A better strategy might be:

for ever
  while no box is Runnable
    execute each Self-output box
  execute each Runnable box
  super-step

Comparing each approach with the original untransformed Hume program we would expect to see the results as show in Table 4.

5 STATIC AND DYNAMIC SELF-OUTPUT IDENTIFICATION

It would be highly beneficial to identify statically box matches with the self-output property at compile time, to enable optimal scheduling in their presence. For HW-Hume this is straightforward as there are no conditional expressions on the right hand side of box matches. Thus, self-output matches can be identified by direct inspection of associated actions and wiring.

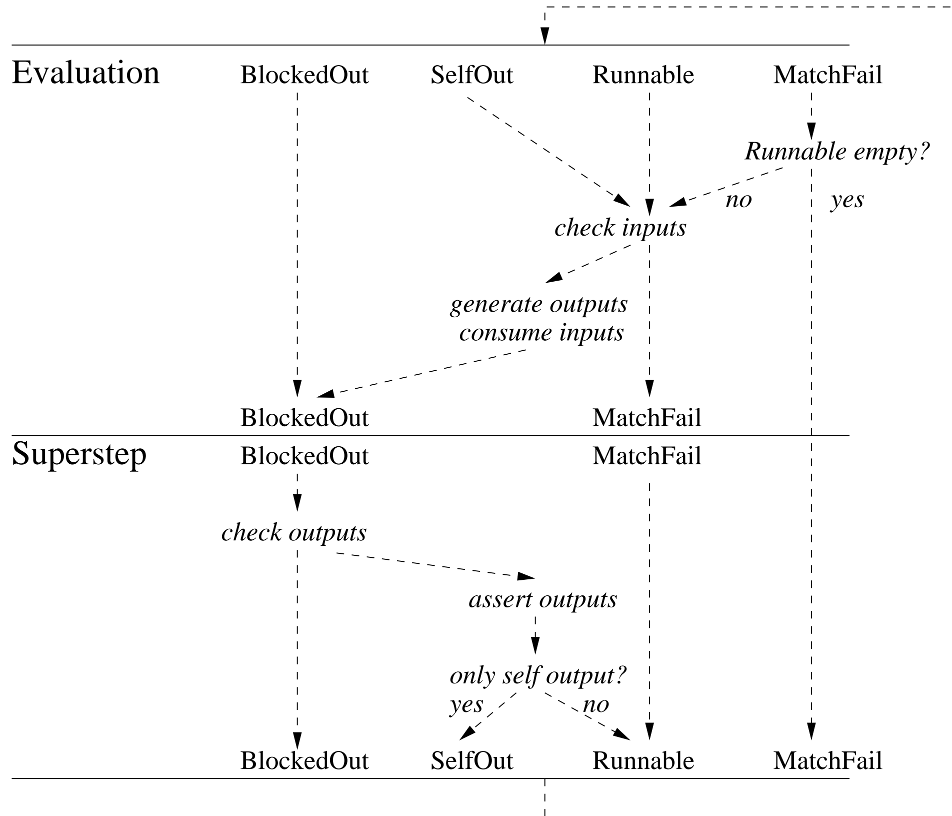

FIGURE 4. Execution cycle with self-output prioritisation.

However, FSM-Hume introduces conditional expressions in box match actions and so analysis can at best identify matches which may self-output. If such matches are treated liberally as self-output then their boxes may be prioritised inappropriately. In contrast, if such matches are treated conservatively as non-self-output then opportunities for scheduling efficiencies will be lost. This is compounded for PR- and full Hume which allow mutually recursive functions, additionally requiring full dataflow analysis.

Thus, we have modified the reference interpreter to implement the third modified scheduling approach, as shown in Figure 4, with dynamic identification of self-output boxes.

In this implementation, on the superstep we actively inspect both the output values and the wiring for each box and deem a box self-output if it has only produced values on feedback wires.

Executing the iterative multiplication example on the original and modified interpreters gives a further 14% time saving for this example, as shown in Table 5.

This saving results from a significant reduction in unnecessary scheduling, as shown in Table 6.

Version	Scheduling	Time (secs)
recursive	original	100.4
iterative	original	70.9
iterative	new	57.0

TABLE 5. Execution times with self-out scheduling.

Box	Scheduling	Runnable	Matchfail	Selfout
mult2	new	1412	1412	0
itermult	new	1411	1411	97178

TABLE 6. New scheduling states.

By comparison with the old scheduling state behaviour shown in Table 3, with the new scheduling scheme, `mult2` is never scheduled once it has entered a failed match state, so long as `itermult` is looping in the self output state, giving a saving of around 997,000 unnecessary schedules out of around 2,000,000.

6 CORRECTNESS

The Hume super-step scheduling results in program output being deterministic and independent of the order of box scheduling. The transformation of functions into boxes and modified scheduling is now shown informally as preserving the program super-step semantics.

The evaluation of Hume expressions (within a box) is performed in a strict and deterministic manner, that is, there is a single thread of execution for evaluation. By observing that the single threaded evaluation of an expression means that evaluation can occur at only one locus in an expression at any single point in time, we claim that, in an equivalent set of generated boxes, one and only one box can ever be runnable at any point in time. Therefore, in transforming functional code into a set of boxes, the boxes must under-utilise the potential for concurrency.

With only one box runnable at any time, the scheduling of these boxes becomes trivial - evaluate the one box then super-step the outputs of this same box. Thus the super-step semantics are preserved but irrelevant and can be made substantially more efficient.

The original set of boxes would typically utilise concurrency and hence the super-step semantics are vital. The two main approaches suggested in this paper - hierarchical and self-output - preserve the program super-step semantics by forcing all of the transformed generated boxes to run to completion before enabling the original boxes again.

The hierarchical approach (as shown in Figure 5) treats the generated boxes as a Hume program $embeddedwithinaHumeprogram$. While it preserves the necessary

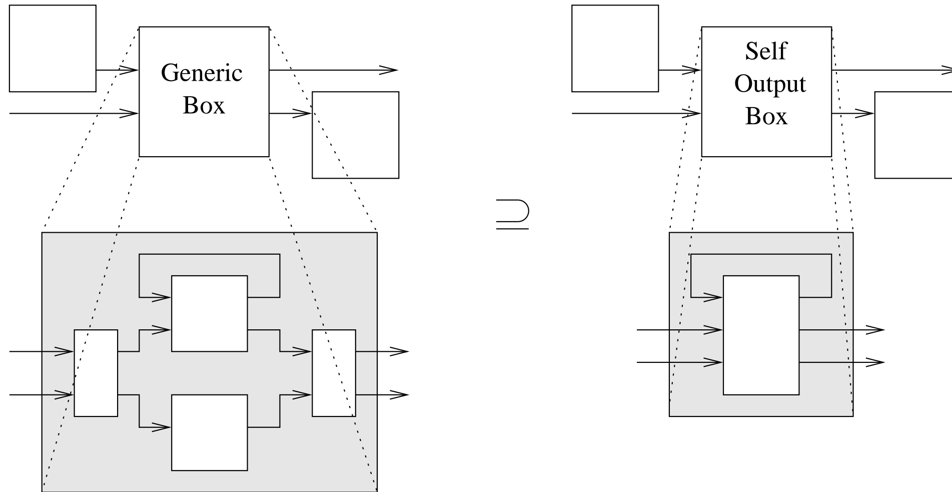

FIGURE 5. Decomposition of Hume boxes.

semantics it also enables a more general and powerful programming model, i.e. it does not restrict the nested boxes to being single threaded and it also allows hidden state. The self-output approach (as shown in Figure 5) optimises the scheduling for a particular class of generated boxes, and thus is slightly restrictive. A more general solution would mark the generated boxes at compile time and use this information along with the dynamic scheduling.

While we stated that only one box could ever be runnable at any time, it may in practise be useful to allow more than one to be runnable, but in a parallel pipelined approach. However the generated boxes are still not, and cannot be, concurrent.

7 SUMMARY

We have introduced a transformation from linear recursion to box iteration, which increases the applicability of static resource analysis and improves performance by reducing memory overheads. This transformation highlights scheduling inefficiencies in the presence of self-output boxes which repeatedly process their own outputs without affecting other boxes, resulting in unnecessary scheduling of other boxes. We have presented a modified scheduling approach that substantially improves scheduling of self-ouput boxes. Finally, we have discussed the benefits of box abstraction for generalising this improvement to enable independent Hume program components to execute in parallel.

We now wish to:

- automate the transformation;
- apply the transformation to realistic programs;

- explore the impact of the modified scheduling strategy on a wide range of programs;
- initiate debate on how best to abstract over Hume boxes;
- investigate related dataflow analysis techniques.

8 ACKNOWLEDGEMENTS

This research is supported by the EU FP6 EmBounded Project.

We would like to thank our collaborators in the EmBounded and SEAS projects, in particular Kevin Hammond.

REFERENCES

[HM03] K. Hammond and G. Michaelson. Hume: A Domain Specific Language for Real-Time Embedded Systems. In *Proceedings of GPCE'03: Generative Programming and Component Engineering, Erfurt, Germany*. Springer, LNCS, September 2003.

[Man74] Z. Manna. *Mathematical Theory of Computing*. McGraw-Hill, 1974.

[MHJ04] G. Michaelson, K. Hammond, and J.Serot. FSM-Hume: Programming Resource-Limited Systems using Bounded Automata. In *Proceedings of ACM Symposium on Applied Computing, Nicosia, Cyprus*, pages 1455–1461. ACM Press, March 2004.

[MK02] G. Michaelson and K.Hammond. The Hume Language Definition and Report, Version 0.2. Technical report, Heriot-Watt University and University of St Andrews, January 2002.