Towards a Box Calculus for Hierarchical Hume

Gudmund Grov and Greg Michaelson

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

Abstract

We present a first approach towards a calculus of transformations of Hume boxes, using an extended version of Hume called Hierarchical Hume. We present and motivate Hierachical Hume, transformations and the calculus and derive some rules and strategies. The approach is then illustrated through two examples.

1 INTRODUCTION

We have been exploring a new cost-driven, transformational approach to software construction from certified components which is highly suited to dynamic, reconfigurable embedded systems. This approach builds on the modern *layered* programming language *Hume*[HM03], whose strengths lie in the explicit separation *coordination* and *control* concerns. Hume is based on autonomous *boxes* linked by *wires* and controlled by generalised transitions. Boxes and wires are defined in the finite state *coordination language* with transitions defined in the *expression language* through pattern matching and associated recursive actions. Both coordination and expression languages share a rich polymorphic type system, comparable to contemporary functional languages like Haskell and Standard ML.

Hume offers programmers different programming *levels* where expressivity is balanced against accuracy of behavioural modelling. *Full Hume* is a general purpose, Turing complete language with undecidable correctness, termination and resource bounds. *PR-Hume* restricts Full-Hume expressions to primitive recursive constructs, enabling decidable termination and bounded resource prediction. *Template-Hume* further restricts expressions to higher-order functions with precise cost models, enabling stronger resource prediction. In *FSM-Hume*, types are restricted to those of fixed size and expressions to conditions over base operations, enabling highly accurate resource bounds. Finally, *HW-Hume* is a basic finite state language over tuples of bits, offering decidable correctness and termination, and exact resource analysis.

However, rather than requiring programmers to choose a level from the outset, we have elaborated an iterative methodology based on cost-driven transformation. An initial Hume program, designed to meet its specification, is analysed to establish resource bounds. Where established bounds are unacceptable, the offending program constructs are transformed, usually to lower levels, and the program is again analysed, with the cycle continuing until the required analytic precision is reached.

At present, transformations are chosen on an ad-hoc basis and applied by hand. In this paper we propose a calculus to direct the transformation of on an extension of Hume, called *Hierarchical Hume* The calculus is intended to guarantee that each step of a transformation preserves the program's behaviour, and therefore introduces the *correctness by construction* principle.

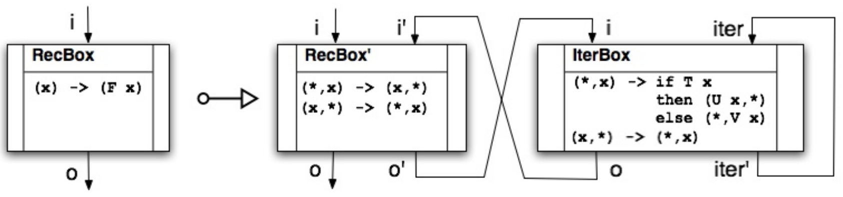
2 HIERARCHICAL HUME

In Hume, a program consists of unitary boxes connected by wires, where a box consists solely of transitions from inputs to outputs. The transitions are guided by a list of *matches* – each consisting of a *pattern* and a corresponding *expression*. A pattern match triggers the corresponding expression which produces the output. '$*$' in any pattern/expression means ignore input/output. A box is *Blocked* if an output cannot be asserted because there is already a value on one of its output wires. The box will not execute again until that output can be asserted i.e. another box consumes the old wire value.

The Hume execution model is based on cyclical execution, where on each cycle all boxes attempt to consume inputs to generate outputs once, and all input/output changes are then resolved in a unitary super-step. In this model, execution order is irrelevant: boxes are stateless and have no side effects on the external environment. However, as every box executes once on each cycle, in a naive implementation, as the number of boxes grows so does the potential for unnecessary but nonetheless resource consuming activity, where boxes repeatedly fail to consume inputs until other boxes make them available as outputs.

Now, the main loci of transformation from an upper to a lower level is to move activity from control to coordination, reducing activity within a box but increasing the number of boxes in compensation. For example, in moving from primitive recursive forms in PR-Hume to iterative forms in FSM-Hume, using a variant of the well known tail recursion optimisation [Man74], a call to recursion within a PR-Hume box is replaced by wires from that box to a new FSM-Hume box using feedback wires to enable iteration - see Figure 1. Wires, represented by labelled directed arcs, are indicative and the labels refer to the name of the output/input in the box. The original `RecBox` box would execute once for a recursion of depth *N*, now both `RecBox'` and `IterBox` will execute *N* times with the original box `RecBox'` executing needlessly. Furthermore, while box execution is order independent it is time dependent: changing the number of boxes and hence the time for each overall execution cycle may have unpredictable effects on other boxes with explicit time constraints.

In the proposed *Hierarchical Hume* extension[GPMI07], a box may contain an entire Hume program, so one box may be composed from a hierarchy of nested boxes. At the top level, the program is still scheduled by a single superstep. However, nested boxes may now be scheduled repeatedly for one cycle of the nesting box.



```
(x) -> (F x)
```

```
(*,x) -> (x,*)
(x,*) -> (*,x)
```

```
(*,x) -> if T x
         then (U x,*)
         else (*,V x)
(x,*) -> (*,x)
```

FIGURE 1. Recursion to Box Iteration.

The introduction of nested boxes greatly mitigates the impact of transformation. If one box is replaced by a hierarchy, then timing effects are localised and may be considered independently of the rest of the program, provided the transformed box retains the same or compatible top-level timing behaviour.

Figure 2 illustrates the effect of the transformation of a single box half adder to a hierarchical box containing the equivalent multi-box AND/XOR configuration. The original single box (a) on the left is a straight transcription of the equivalent truth table[1]. In the new hierarchy (c) on the right, the nesting box inputs and outputs are wired explicitly to the appropriate nested box inputs and outputs. In the graphical representation of the hierachical box (b), the transition details within the box are elided.

For nested boxes, parent box inputs are immediately wired to child box inputs, and child box outputs to parent box outputs. Thus the matches in a parent box are soley indicative of the combinations of presence or absence of inputs and outputs that characterise termination. For instance, in box `half2`, '`(_,_)->(_,_)`' means that it will start executing when all inputs are available and terminate when all outputs are available.

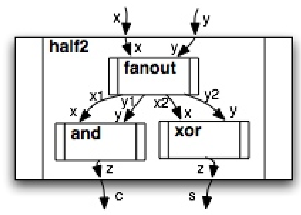
Informally, a transformation is correct if the top level wires in the new configuration always have the same values at the same stages in execution as the equivalent wires in the original configuration. Consequently, in a correct transformation the top level boxes observationally implement the boxes before the transformation. In the example given in Figure 2 this means that the transformed box `half2` (b/c) must behave as `half1` (a). Due to the hierarchy, top level timing can be ignored, and the focus is within the new box. Firstly, `half1` is defined for all type correct inputs, and produces values on all outputs. This is achieved by the '`(_,_)->(_,_)`' match of `half2`. It is easy to see that the `c` (first) output of `half1` with the given inputs is basically and XOR gate. Further, `s` is an AND gate. By "fanning out" the inputs to an XOR/AND pair the same output will be produced, which is exactly the case in `half2`. It therefore implements `half1`. In the example given by Figure 1, if we assume functional correctness, then the transformation is correct if `RecBox'` and `IterBox` are nested inside a first level box, with `i` and `o` wired to the parent box.

[1]as are the AND and XOR boxes within the new hierarchy (c).

```
-- Only 0 and 1
type Bit = int 1;

box half1
  in (x,y::Bit)
  out (s,c::Bit)
match
 (0,0) -> (0,0) |
 (0,1) -> (1,0) |
 (1,0) -> (1,0) |
 (1,1) -> (0,1);
```

a. Half Adder 1: Truth Table



b. Half Adder 2: (Graphic) XOR and AND gates

```
box half2
   in (x,y::Bit)  out (s,c::Bit)
match
  (_,_) -> (_,_)
boxes
   box fanout
      in (x,y::Bit)
      out (x1,y1,x2,y2::Bit)
   match
       (x,y) -> (x,y,x,y);
   wire fanout (half1.x,half2.y)
               (xor.x,xor.y,and.x,and.y);
   box xor
       in (x,y::Bit)  out (z::Bit)
   match
      (0,0) -> 0 |
      (0,1) -> 1 |
      (1,0) -> 1 |
      (1,1) -> 0;
   wire xor(fanout.x1,fanout.y1)
             (half1.s);
   box and
      in (x,y::Bit)  out (z::Bit)
   match
     (0,0) -> 0 |
     (0,1) -> 0 |
     (1,0) -> 0 |
     (1,1) -> 1;
   wire and (fanout.x2,fanout.y2)
            (half1.c);
end;
```

c. Half Adder 2: Source Code

FIGURE 2. Half Adders in Hierarchical Hume

3 THE RULE SYNTAX AND SEMANTICS

While, the box calculus should reflect the formal semantics of Hume, Hierarchical Hume does not yet have a formal semantics and many features in the Hume semantics are not relevant for this discussion. Consequently, we have simplified the Hume semantics slightly to ease presentation of the new hierarchical features.

A Hume program configuration consists of a triple $\langle\theta,\eta,bcs\rangle$: θ is the wire heap with allocated space for each wire. It also holds potential initial wire values; η is the internal heap, including internal wires for hierarchical boxes – which in this instance for simplicity, we assume remains allocated throughout execution; *bcs* is a list of box configuration. Each box configuration consists of the elements $\langle id,iws,ows,rs,ii,io,ibcs\rangle$: *id* is the box's name; *iws* is a list of locations holding the input wires; *ows* is a list of locations holding the output wires; *rs* is a list of matches; The three last elements are empty (lists) if the box is not nested: *ii* is a list of location of internal inputs wires; *io* is a list of location of internal output wires; and *ibcs* is a list of box configurations of internal (nested) boxes. We let run_{bcs} represent one execution cycle of the program including the super step. It is a predicate on two pairs of wire and internal heaps $\langle\langle\theta,\eta\rangle,\langle\theta',\eta'\rangle\rangle$ where $\langle\langle\theta,\eta\rangle$ is a 'before heaps' and $\langle\theta',\eta'\rangle$ an 'after heaps'. run_{bcs} then holds if given $\langle\theta,\eta\rangle$ the result of executing *bcs* is $\langle\theta',\eta'\rangle$.

The box calculus consists of a set of conditional rewrite rules. A rule changes the triple $\langle\theta,\eta,bcs\rangle$ and has the syntax

$$\theta,\eta;bcs \vdash \mathbf{Rule}(X_1,\cdots,X_n) \Downarrow \theta',\eta';bcs'.$$

This is read as "**Rule** with parameters $X_1,\cdots,X_n$ will, under the configuration $\langle\theta,\eta,bcs\rangle$ create the configuration $\langle\theta',\eta',bcs'\rangle$". To achieve a set of rules that is expressive enough, steps that change timing behaviour must be allowed. It is therefore imperative that the preconditions are strong enough to ensure that the actual behaviour remains unchanged. This is mostly a coordination issue and the nature of this layer often require temporal properties. In fact, this is the reason for using the HW-Hume level as a starting point for the calculus.

TLA [Lam94] allows us to separate the control and coordination for reasoning, and fits very well into the Hume framework [HGMI06, GPMI07]. However the idiosyncrasies of TLA, and particularly the syntax, make it hard to understand for novices. Since these details are beyond the scope of this paper they have been omitted. Consequently the proof rules below have a more complex underlying TLA machinery. In particular, hierarchies must be flattened for us to prove that a transformation is in fact correct, i.e. global and local steps are not separated at this level. At the end of the day, it is only the values on the wires that are interesting, while the internal details purpose is to help achieving this. These are therefore *hidden*. The details are again very intricate, and will not be discussed in great detail. Although, it should be noted that logically by hiding a component the specification is strictly weaker. Further, hiding internal details is essential in transformation proofs involving hiearchies. We write $\overline{\eta}$ to show that η is hidden, and $run_{\overline{bcs}}$ for running *bcs* with η hidden. We then use an induction principle to show the correctness: initially, the new heaps must be strictly stronger than before the transformation – and all actions updating the heaps must be strictly stronger than the actions before the transformation:

$$\frac{\langle\theta',\overline{\eta'}\rangle \Rightarrow \langle\theta,\overline{\eta}\rangle \qquad run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}}{\langle\theta',\eta',bcs'\rangle \Rightarrow_T \langle\theta,\eta,bcs\rangle}$$

Note that the primed components are the translated ones. Further, $\Rightarrow_T$ is actually a specialisation of TLA rules for Hume, and its soundness therefore follows the soundness of TLA. An important feature, which underpins the calculus, is the transitivity of $\Rightarrow_T$:

Theorem 1. *$\langle\theta,\eta,bcs\rangle \Rightarrow_T \langle\theta',\eta',bcs'\rangle$ and $\langle\theta',\eta',bcs'\rangle \Rightarrow_T \langle\theta'',\eta'',bcs''\rangle$ implies $\langle\theta,\eta,bcs\rangle \Rightarrow_T \langle\theta'',\eta'',bcs''\rangle$.*

Proof. The proof reduces to transitivity of $\Rightarrow$ which is trivial. $\square$

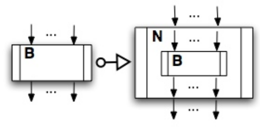
4 RULES

The general categories of transformation rules in the box calculus will be familiar from many comparable calculi. Thus, there are rules to: introduce/eliminate identity boxes; introduce/eliminate nesting boxes; introduce/eliminate wires; combine/separate boxes horizontally and vertically; expand/contract match patterns and results; reorder patterns and results. Special to Hume are rules for moving activity between result expressions within boxes and coordination between boxes. Indeed, in Hume, coordination and expression level transformation are tightly coupled, and there are necessarily strong links between the apparently distinct categories above.

A full formal definition of all the rules will require much more space than available here. We will therefore limit this to two rule derivation and sketch their correctness proofs. The remaining rules are listed in Appendix A. The listing also includes some auxiliary functions, which do not have any side effects on the program configuration. Details, like pre conditions, has been omitted in the listings. We use standard logical terminology in the rules: A rule postfixed by 'I' is a rule that "introduces something", and its dual, the elimination rule, is postfixes by 'E'. In the rule derivation we give an informal graphical representation of impact of the rule. In the graphical representation we do not show any potential siblings or parents of relevant boxes. Henceforth these, together with the box itselft, will be known as the *context* of the box. It is important to note timing contraints only relates to the context of a box. Everything outside the context is independent of this.

The first rule nests one box B inside another box A with name N. This rule introduces a *bounded context* for B, only consisting of A and B. By applying this rule we can ignore the top level timing dependies when transforming B, and many (temporal) preconditions of rules require a bounded context. The rule copies input and output wires to the internal heap, by using **HeapLocs_Copy**. These are the new wires of the newly created nested box B', and the internal wires of the nesting box A. Further, A consists of one nested box B' and *generalises* B's rule set into the more restricted hierarchical form, by **Gen_Rules**:

$$\frac{\begin{array}{c}\langle B, iws, ows, rs, iw, ow, ibcs\rangle = \mathbf{get\_box}(B, bcs) \\ \langle niw, \eta''\rangle = \mathbf{HeapLocs\_Copy}(iws, \theta, \eta) \\ \langle now, \eta'\rangle = \mathbf{HeapLocs\_Copy}(ows, \theta, \eta'') \\ B' = \langle B, niw, now, rs, iw, ow, ibcs\rangle \qquad irs = \mathbf{Gen\_Rules}(rs) \\ A = \big\langle N, iws, ows, irs, niw, now, [B']\big\rangle \\ \theta, \eta; bcs \vdash \mathbf{Replace}\big([A],[B]\big) \Downarrow \theta, \eta'; bcs'\end{array}}{\theta, \eta; bcs \vdash \mathbf{HieI}(B, N) \Downarrow \theta, \eta'; bcs'}$$



Next we sketch the proof that shows that the transformatoin is indeed correct.

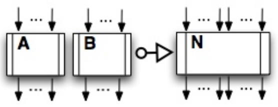
Theorem 2.

$$\begin{array}{ll} \textit{If} & \theta, \eta; bcs \vdash \mathbf{HieI}(A, N) \Downarrow \theta', \eta'; bcs' \\ \textit{then} & \quad \langle \theta', \eta', bcs'\rangle \Rightarrow_T \langle \theta, \eta, bcs\rangle \end{array}$$

Proof. Since we only extend $\overline{\eta}$ and do not change θ, $\langle\theta',\overline{\eta}'\rangle \Rightarrow \langle\theta,\overline{\eta}\rangle$ holds. In *bcs*, B is replaced by A. Since As rule set generalises Bs the matching will be the same. Further, with this and since A only contains B, the computation and termination will be the same, and therefore also the result. Therefore $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ holds. $\square$

In the second derivation two non-nested boxes, A and B, are horizontally composed into a new box called N. However, A and B must always have the same *Blocked* status, since N will be *Blocked* if either of them are: If one, but not the other, is *Blocked* the behaviour of the composed box N will not capture the sum of A and B. The inputs and outputs of A prefixes Bs inputs and outputs. For all matches, the patterns and expression of the A and B are pairwise composed by **project**. This projection might introduce non-determinacy, so the patterns must be mutually exclusive. Finally, A might execute while B fail to pattern match the inputs, and vice verse. This is captured by postfixing the composed rule set above with a rule set where A's rule set is composed with only '$*$'s, and the same for B. The box N', capturing all the above, replaces A and B:

$$\frac{\begin{array}{c}
\langle A, iws_A, ows_A, rs_A, [], [], []\rangle = \textbf{get\_box}(A, bcs) \\
\langle B, iws_B, ows_B, rs_B, [], [], []\rangle = \textbf{get\_box}(B, bcs) \\
\square\big(\textbf{is\_Blocked}(A) \equiv \textbf{is\_Blocked}(B)\big) \\
\textbf{mutually\_exclusive}(rs_A) \qquad \textbf{mutually\_exclusive}(rs_B) \\
iws = iws_A @ iws_B \qquad ows = ows_A @ ows_B \\
n_A = \textbf{len}(iws_A) \quad m_A = \textbf{len}(ows_A) \\
n_B = \textbf{len}(iws_B) \quad m_B = \textbf{len}(ows_B) \\
*_A = [\underbrace{\langle *, \cdots, * \rangle}_{n_A} \rightarrow \underbrace{\langle *, \cdots, * \rangle}_{m_A}] \quad *_B = [\underbrace{\langle *, \cdots, * \rangle}_{n_B} \rightarrow \underbrace{\langle *, \cdots, * \rangle}_{m_B}] \\
rs = \textbf{project}(rs_A, rs_B) \ @ \ \textbf{project}(rs_A, *_B) \ @ \ \textbf{project}(*_A, rs_B) \\
N' = \langle N, iws, ows, rs, [], [], []\rangle \\
\theta, \eta; bcs \vdash \textbf{Replace}([N'], [A, B]) \Downarrow \theta, \eta; bcs'
\end{array}}{\theta, \eta; bcs \vdash \textbf{HCompI}(A, B, N) \Downarrow \theta, \eta; bcs'}$$



The unification with the empty lists ensures that the boxes are not nested, when calling **get_box**. The mutual exclusiveness test is straightforward, and the proof of the *Blocked* status temporal invariance proof. This has therefore be prefixed by the the temporal 'always' operator $\square$ – denoting that this must hold throughout execution.

Theorem 3.

$$\begin{array}{ll} \textit{If} & \theta, \eta; bcs \vdash \textbf{HCompI}(A, B, N) \Downarrow \theta, \eta; bcs' \\ \textit{then} & \qquad \langle\theta', \eta', bcs'\rangle \Rightarrow_T \langle\theta, \eta, bcs\rangle \end{array}$$

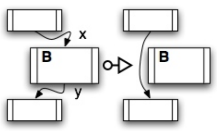
Proof. There is no nesting, hence $\overline{\eta} = \eta$. Further, it is obvoius that $\theta' = \theta$ and $\eta' = \eta$, thus $\langle\theta',\overline{\eta'}\rangle \Rightarrow \langle\theta,\overline{\eta}\rangle$. The proof of $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ is by case-analysis on the "execution state" of A and B: Since $\square\big(\textbf{is\_Blocked}(A) \equiv \textbf{is\_Blocked}(B)\big)$ we know that A and B are always *Blocked* at the same time. Hence, if one is *Blocked* then so is the other, and since N will be *Blocked* if either of them are, then so is N. If both A and B succeeds then, since all possible matches are composed, so will

N. Since the patterns are mutually exclusive only one pattern that can succeed, and the result is obviously the same. If both boxes fail to execute, then so will N since it only composes A and B. Finally, the case where only one box succeeds is captured by the case where each match is composed with only '$*$'s. Thus the goal holds. □

5 STRATEGIES

The rules will often be too low-level to work with. Instead a user will work with higher-level *strategies*, which are derived from rules and other strategies. An example of a strategy, although still rather low-level, is the elimination of *threading*. A wire is threaded through a box if there is a one-to-one correspondence between a pattern x and an expression y in all matches. x cannot be used in other expressions ($\neq y$). Further, x and y must form an identity box. When elimated, the threaded value will arrive earlier at the destination. This must not have any effect on the context. Finally, a *Blocked* state on B will prevent the threaded value leaving B, which is not the case when eliminated. This must again not have any impact on the context. Since the rule is derived from other rules these precondition can be ignored since they are implicitly captured by the precondition of the rules in the derivation. Threading elimination, **ThreadE**, is derived as follows: x and y are horizontally de-composed into a new box *Id* by **HCompE**. *Id* is then an identity box eliminated by **IdE**:

$$\frac{\begin{array}{c}\Sigma,\theta,\mathcal{L} \vdash \mathbf{HCompE}(B,[x],[y],\mathit{Id},B) \Downarrow \Sigma_1,\theta_1,\mathcal{L}_1 \\ \Sigma_1,\theta_1,\mathcal{L}_1 \vdash \mathbf{IdE}(\mathit{Id}) \Downarrow \Sigma',\theta',\mathcal{L}'\end{array}}{\Sigma,\theta,\mathcal{L} \vdash \mathbf{ThreadE}(B,x,y) \Downarrow \Sigma',\theta',\mathcal{L}'}$$



The correctness proof for strategies are trivial since they only rely on Theorem 1:

Theorem 4.

If $\Sigma,\theta,\mathcal{L} \vdash \mathbf{ThreadE}(B,x,y) \Downarrow \Sigma',\theta',\mathcal{L}'$
then $\langle \Sigma',\theta',\mathcal{L}' \rangle \Rightarrow_T \langle \Sigma,\theta,\mathcal{L} \rangle$

Proof. Since the two given are applied sequentially the proof reduces to the transitivity of $\Rightarrow_T$. This is proved by Theorem 1. □

6 EXAMPLES

6.1 Example 1: Half-Adder

First we apply the box calculus to the half adder example above. We will do it stepwise, and a graphical representation of each of these steps is shown in Figure 3. We use a dot '.' notation to refer to nested boxes, starting from the first level. If a rule has more than one parameter, it is sufficient to give the full path to one of the boxes, since we can only work in one context at a time. We omit the configuration triple to make the text easier to read. The rules are sequentially applied.

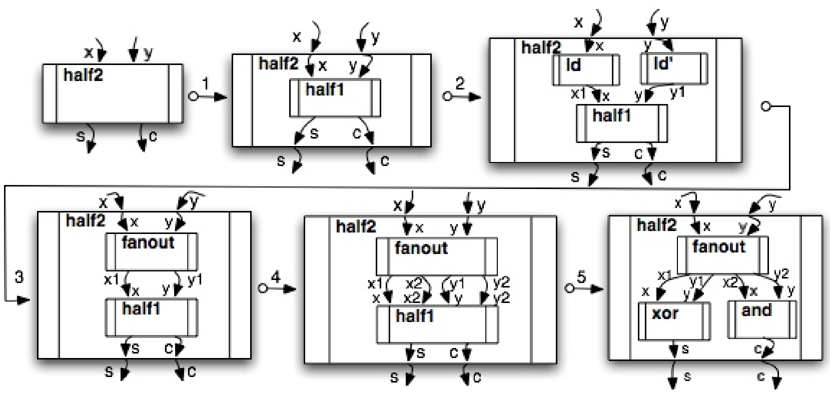


FIGURE 3. Transformation of Half Adder

1. Since the transformation has a forward direction we start with the box shown in Figure 2a. First rule **HieI**(`half1`,`half2`) which replaces box `half1` with a box `half2` that simply nests it.

2. Since there are no '$*$' in the context nested by `half2` there are no dependencies. We can therefore introduce identity boxes for both input wires of `half1`: **IdI**(`half2.half1`,`x`,`Id`) followed by **IdI**(`half2.half1`,`y`,`Id′`). The input/output variables of the identity boxes are `v/v′` by default. These are renamed to `x/x1` and `y/y1` respectively: **VRename**(`half2.Id`,`v`,`x`), **VRename**(`half2.Id`,`v′`,`x1`), **VRename**(`half2.Id′`,`v`,`y`) and **VRename** (`half2.Id′`,`v′`,`y1`).

3. The two identity boxes are then horizontally composed into one box called `fanout`: **HCompI**(`half2.Id`,`Id′`,`fanout`):

   ```
   box fanout
     in (x,y::Bit)  out (x1,y1::Bit)
   match
     (x,y)->(x,y)  |  (x,*)->(x,*)  |  (*,y)->(*,y) ;
   ```

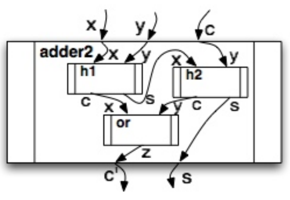
   A simple invariant of the internal behaviour of `half2` shows that it will never be the case that only one of `fanout`'s inputs is empty. The last two matches of `fanout` will therefore never succeed. This is the only precondition in the match elimination rule, and can therefore be applied: **MatchE**(`half2.fanout`,`3`) and **MatchE**(`half2.fanout`,`2`).

4. We then duplicate the two wires connecting `fanout` and `half1`. We name them `x2` and `y2`: **DupI**(`half2.fanout`,`x1`,`x2`,`half1`,`x`,`x2`) followed by **DupI**(`half2.fanout`,`y1`,`y2`,`half1`,`y`,`y2`).

5. In `half1` we now have two set of identical inputs: $\{x,y\}$ and {`x2`,`y2`}. We can then state that output s depends on the first set and c on the second, and decompose the box. The first of this boxes is exactly the same

```
box adder1
in (x,y,c::Bit)
out (s,c'::Bit)
match
 (0,0,0) -> (0,0) |
 (0,1,0) -> (1,0) |
 (1,0,0) -> (1,0) |
 (1,1,0) -> (0,1) |
 (0,0,1) -> (1,0) |
 (0,1,1) -> (0,1) |
 (1,0,1) -> (0,1) |
 (1,1,1) -> (1,1) ;
```

a. Adder 2: Truth Table



b. Adder 2: (Graphic) Half Adder and OR gate

```
 box adder2
    in (x,y,c::Bit)  out (s,c'::Bit)
 match
   (_,_,_) -> (_,_)
 boxes
    box h1
       in (x,y::Bit) out (s,c::Bit)
    match
      (0,0) -> (0,0) |
      (0,1) -> (1,0) |
      (1,0) -> (1,0) |
      (1,1) -> (0,1);
   wire h1(adder2.x,adder2.y)(h2.x,or.x);

   box h2
       in (x,y::Bit) out (s,c::Bit)
    match ... -- same as h1
   wire h2(h1.c,adder2.c)(adder2.s,or.y);

   box or
      in (x,y::Bit)  out (z::Bit)
   match
     (0,0) -> 0 |
     (0,1) -> 1 |
     (1,0) -> 1 |
     (1,1) -> 1;
   wire or(h1.c,h2.c)(adder2.c);
end;
```

c. Adder 2: Source Code

FIGURE 4. Full Adders in Hierarchical Hume

as the `xor` while the second is the same as the `and` box of Figure 2(b/c): **HCompE**(`half2.half1`, [`x`, `y`], [`s`], `xor`, `and`). Finally, we rename the inputs of the `and` box: **VRename**(`half2.and`, `x2`, `x`) and **VRename** (`half2.and`, `y2`, `y`). This concludes the transformation.

6.2 Example 2: A Full Adder

The second example is more complex: A full adder represented as a truth table (Figure 4a) is transformed into a representation using two half adders and an OR gate (Figure 4b/c). Again, the transformation is step-by-step and each step is graphically illustrated in Figure 5:

1. The transformation starts with `adder1` from Figure 4a. First we move all the matches inside a case expression. Since the patterns are total with respect to the `Bit` type this is allowed: **CaseI**(`adder1`, `1`, `8`):

```
box adder1
  in (x,y,c::Bit) out (s,c'::Bit)
match
  (a,b,c) -> case (a,b,c) of  ...;
```

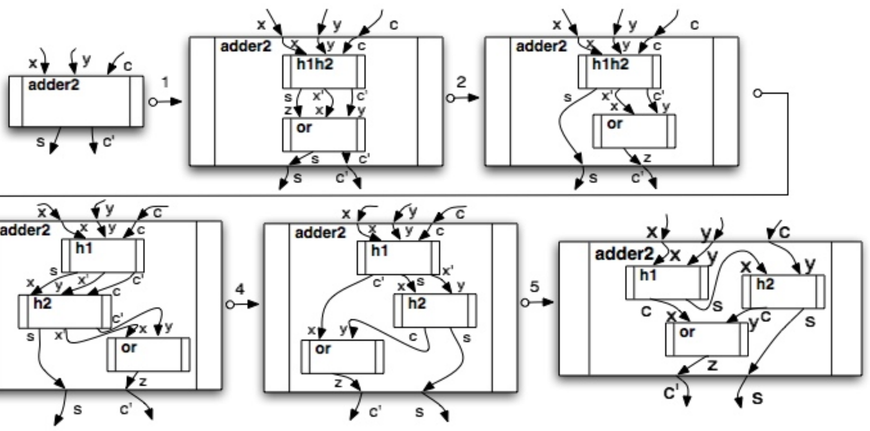


FIGURE 5. Transformation of Full Adder

```
f(a,b,c) = case (a,b,c) of                  g(a,b,c) = case (a,b,c) of
           (0,0,0) -> (0,0,0) |                        (0,0,0) -> (0,0) |
           (0,0,1) -> (1,0,0) |                        (1,0,0) -> (1,0) |
           (0,1,0) -> (1,0,0) |                        (0,0,1) -> (0,1) |
           (0,1,1) -> (0,0,1) ...;                     (1,0,1) -> (1,1) ...;


ff(a,b,c) = case (a,b,c) of                 gg(a,b,c) = case (a,b,c) of
            (0,0,0) -> (0,0,0) |                        (0,0,0) -> (0,0,0) |
            (0,0,1) -> (1,0,0) |                        (0,0,1) -> (0,1,0) |
            (0,1,0) -> (0,1,0) |                        (0,1,0) -> (1,0,0) |
            (0,1,1) -> (1,1,0) ...;                     (0,1,1) -> (1,1,0) ...;
```

FIGURE 6. Auxiliary Functions Used in Full Adder Transformation

The case expression is then replaced by the function composition `g·f(a,b,c)`: **ReplaceExpr**(`adder1`,`1`,`g·f(a,b,c)`) where `f` and `g` are shown in Figure 6. The next step is to vertically de-compose this box – where *f* is the expression of the first and *g* the expression of the second box. However, this will introduce an extra step, and we do not know anything about the context, so we need to nest the boxes first: **HieI**(`adder1`,`adder2`). The boxes can then safely be de-composed: **VCompE**(`adder2.adder1`,`h1h2`,[`s,x',c'`],`or`,[`z,x,y`]).

2. The newly created `or` box has one match with the expression `g`, where `g` consists of a (total) case expression. We unfold `g` and move the case-expression into the match: **Unfold**(`adder2.or`,`g`) followed by **CaseE** (`adder2.or`,`1`). The result is illustrated on the left side below. The first pattern and expression are identical (and total). We therefore replace them by a variable: **MatchVarI**(`adder2.or`,`x`,`s`). We now have a threading of a variable which we can eliminate (since there are no '`*`' in the context): **ThreadE**(`adder2.or`,`x`,`s`). The result is illustrated on the right side:

```
box or                           | box or
 in (z,x,y::Bit)                 |  in (x,y::Bit)
 out (s,c'::Bit)                 |  out (c'::Bit)
match                            | match
   (0,0,0) -> (0,0) |            |    (0,0) -> 0 |
   (1,0,0) -> (1,0) |            |    (0,0) -> 0 |
   (0,0,1) -> (0,1) |            |    (0,1) -> 1 |
   (1,0,1) -> (1,1) ...;         |    (0,1) -> 1 ...;
```

Matches 2,4,6 and 8 are now duplicates of their previous matches, and can therefore be removed: **MatchE**(`adder2.or`, 8),**MatchE**(`adder2.or`, 6), **MatchE**(`adder2.or`, 4) and **MatchE**(`adder2.or`, 2). Finally, the output wire is renamed to `z`: **VRename**(`adder2.or`,`c'`,`z`). The `or` box is now the same as in Figure 4c.

3. Box `h1h2` consists of one match with expression `f`. This function can be replaced by function composition `gg`·`ff(a,b,c)` where `ff` and `gg` are shown in Figure 6: **ReplaceExpr**(`adder2.h1h2`, 1, `gg`·`ff(a,b,c)`). Since the context do not contain any '$*$'s we can apply vertical function decomposition **VCompE**(`adder2.h1h2`, `h1`, [`s`, `x'`, `c'`], `h2`, [`x`, `y`, `c`]):

4. In box `h2` the match has the expression `gg` which is unfolded and the (total) case-expression is moved into the body: **Unfold**(`adder2.h2`, `gg`) and **CaseE**(`adder2.h2`, 1) as illustrated on the left side below. The last pattern and second expression in all matches can be replaced by a variable, which creates a threading that can be eliminated: **MatchVarI**(`adder2.h2`, `c`, `s`) and **ThreadE**(`adder2.h2`, `c`, `s`) – as illustrated on the right side:

```
box h2                             | box h2
 in (x,y,c::Bit)                   |  in (x,y::Bit)
 out (s,x',c'::Bit)                |  out (s,c'::Bit)
match                              | match
   (0,0,0) -> (0,0,0) |            |    (0,0) -> (0,0) |
   (0,0,1) -> (0,1,0) |            |    (0,0) -> (0,0) |
   (0,1,0) -> (1,0,0) |            |    (0,1) -> (1,0) |
   (0,1,1) -> (1,1,0) ...;         |    (0,1) -> (1,0) ...;
```

Matches 2,4,6 and 8 are now duplicates of previous matches and therefore removed: **MatchE**(`adder2.h2`, 8), **MatchE**(`adder2.h2`, 6), **MatchE**(`adder2.h2`, 4) and **MatchE**(`adder2.h2`, 2). After renaming the last output to `c'` we have created a correct implementation of a half adder: **VRename**(`adder2.h2`,`c'`,`c`).

5. The transformation of `h1` is follows the same pattern as `h2` (and `or`): First the case expression is removed, followed by a variable introduction and threading elimination: **Unfold**(`adder2.h1`, `ff`), **CaseE**(`adder2.h1`, 1), **MatchVarI**(`adder2.h1`, `c`, `x'`) and **ThreadE**(`adder2.h2`, `c`, `x'`). Then

the duplicate matches are removed, which creates a correct implementation of a *half*-adder: **MatchE**(adder2.h1,8), **MatchE**(adder2.h1,6), **MatchE**(adder2.h1,4) and **MatchE**(adder2.h1,2). By remaming c′ to c we have concluded the transformation: **VRename**(adder2.h1,c′,c). To achieve an even lower level representation we can now apply the half-adder transformation to h1 and h2, as explained above.

7 RELATED WORK

A Hume *transformation* is a strategy. Following Visser [Vis05] this can either be categorised as a *program rephrasing*, where the source and target language are the same, or as a *program translation*, where the target language deviates from the source language.

We have already stated that a transformation from an upper to a lower level is a move of activity from control to coordination, i.e. a translation from the expression layer of a box into the coordination layer within a nested box. This has also been illustrated by our examples. A (full) transformation can therefore be seen as form of program translation called *program synthesis* from a control to a coordination representation. In particular, our correctness proof rule is based on a form of synthesis called *program refinement*: the lower level transformed program *implements* the upper level program. In TLA such implementation is represented as logical implication.

However, what is distinctive here compared with synthesis techniques, like Birds-Meertens Formalism [BdM97] and calculational programming [HW06], is the necessarily strong interplay between coordination and expression transformation: changes to box/wire configurations affect matches which in turn affect patterns and results. A single rule application is not therefore just a *program migration* from one representation to another, but hold more resemblence to a form of program rephrasing called *program refactoring* [Fow99]: just as Hume integrates a finite state coordination language with a functional transition control language, the work presented here draws on the twin traditions of process network and functional program transformation. The coordination aspects of the rules have many similarities with those found in box calculus for Petri nets[DKR03] as well as process calculi [Bae05]. The control aspects resembles classic functional programming techniques including curry/uncurry, fold/unfold [BD77] and functional refactoring [LT06]. Hence, a full transformation can be seen as a program translation, consisting of several program rephrasing steps.

In principle the transformation proofs could have been achieved using (observational) bisimulations in a process algebra like CSP[Hoa85]. However, it is not possible to to achieve an adequate representation of Hume's rich expression layer in a process algebra requiring the introduction of further formalism, for example Schneider's B/CSP cpmbination[ST05]. Here, we think a "lifted logic", like TLA, that may be founded on any underpinning predicate formalism is more appropriate.

Previously, we have explored horizontal box integration in establishing informally that FSM-Hume actually is finite state [MHS04]. Different strategies for general formal verification of Hume programs are first discussed in [Gro05]. TLA is first used to verify programs in [HGMI06], while Hierarchical Hume and linear recursion to box iteration with respect to scheduling is discussed in [GPMI07].

8 CONCLUSION AND FUTURE WORK

We have presented a first approach towards a box calculus for Hume programs, which introduces correct transformation by construction, formalised through structural operational semantics and TLA. We have then discussed rule derivation and the combination of rules into strategies, and presented the use of the calculus through two HW-Hume transformation examples.

Our work at the lowest, least expressive HW-Hume level has given us confidence in the calculus and allowed us to focus on the intricate properties of the coordination layer, which are the same for all Hume levels. Extending the calculus to the higher levels of Hume will mainly require an extenstion of the purely functional transformation rules, together with data refinement. This will allow us to tackle problems that have substantive behavioural and hence resource cost implications, like the recursion to iteration example previously discussed. We sepeculate that it may also be necessary to incorporate rules which are not behaviour preserving on their own, but which can be combined into “correct” rules/strategies.

Our next step is to identfy a sufficient set of rules which is adequate for the classes of transformations between and within levels that may be used to optimise resource use. The rules will be realised in the Isabelle theorem prover, which will require a deep embedding of at least the expression layer. A TLA layer will then be built on top of this embedding. In the longer term, we intend the calculus to be used from a graphical-based IDE for Hume. We will also explore how to minimise user-interaction in cost-oriented program development using *proof planning*[Mad91, CIMS05] heuristics to guide transformations.

ACKNOWLEDGMENTS

This work is supported by the EU FP6 EmBounded project, and Gudmund Grov is supported by a James Watt Scholarship. We would like to thank our EmBounded colleagues for discussions of transformation and Hierarchical Hume, in particular Robert Pointon and Andrew Ireland. We are also grateful for the excellent student feedback from the two anonymous TFP reviewers.

REFERENCES

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Scisence*, 335(2-3):131–146, 2005.

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[CIMS05] A. Cook, A. Ireland, G.J. Michaelson, and N. Scaife. Discovering applications of higher order functions through proof planning. *Journal of Formal Aspects of Computing*, 17(1):38–57, 2005.

[DKR03] R. Devillers, H. Klaudel, and R-C. Riemann. General parameterised refinement and recursion for the M-net calculus. *Theoretical Computer Science*, 300(1-3):259–300, May 2003.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, August 1999.

[GPMI07] G. Grov, R. Pointon, G. Michaelson, and A. Ireland. Coordination, Computation and Hume Scheduling, 2007. in preparation.

[Gro05] Gudmund Grov. Verifying the correctness of hume program - an approach combining algorithmic and deductive reasoning. In *Proceedings of the* 20^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE-05)*, pages 444–447. ACM Press, 2005.

[HGMI06] K. Hammond, G. Grov, G. Michaelson, and A. Ireland. Low-Level Programming in Hume: an Exploration of the HW-Hume Level. In *International Conference on Implementation and Application of Functional Languages, Budapest, Hungary*, September 2006. accepted for publication.

[HM03] K. Hammond and G.J. Michaelson. 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, 2003.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall International, 1985.

[HW06] G. Hutton and J. Wright. Calculating an exceptional machine. In H-W. Loidl, editor, *Trends in Functional Programming Volume 5*, pages 49–64, 2006.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Toplas*, 16(3):872–923, May 1994.

[LT06] H. Li and S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In *Proceedings of 6th IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, USA*, September 2006.

[Mad91] P. Madden. *Automated Program Transformation Through Proof Transformation*. PhD thesis, University of Edinburgh, 1991.

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

[MHS04] Greg Michaelson, Kevin Hammond, and Jocelyn Sérot. The Finite State-Ness of FSM-Hume. In *Trends in Functional Programming*, volume 4, pages 19–28. Intellect, 2004.

[ST05] Steve Schneider and Helen Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput*, 17(4), 2005.

[Vis05] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

A SUMMARY OF PROOF RULES

A.1 Functions

get_box(B,bcs)**:** Returns box configuaration with box id B from list bcs.
Gen_Rules(rs)**:** Returns a *generalisation* of rs. In patterns variables are replaced by '_' while the rest is unchanged. In expression everything but '$*'$ is replaced by '_', and all function calls are removed.
HeapLocs_Copy$\big([l_1,\cdots l_n],H_1,H_2\big)$**:** Returns a tuple $\langle[l'_1,\cdots l'_n],H'_2\rangle$ holding a copy of $[l_1,\cdots l_n]$ of H_1 into H_2 and the updated H_2.
is_Blocked(B)**:** Holds if box B cannot be executed.
mutually_exclusive(rs)**:** Holds if the patterns of rule set rs are mutually exclusive.
len(L)**:** Returns the length of list L.
$L_1 @ L_2$**:** Concat list L_1 in front of list L_2.
project$\big([(p_1\rightarrow e_1),\cdots(p_n\rightarrow e_n)],[(p'_1\rightarrow e'_1),\cdots(p'_m\rightarrow e'_m)]\big)$**:** Pairwise combines each pattern p_i and p'_j with e_i and e'_j where $i\in 1..n$ and $j\in 1..m$.

A.2 Rules and Strategies

Replace$\big([A_1,\cdots,A_n],[B_1,\cdots,B_m]\big)$ **:** Replaces boxes $A_1,\cdots,A_n$ by $B_1,\cdots,B_m$.
ReplaceExpr(A,n,e) **:** The expression of match n of box A is replaced by e.
Rename(A,N)**:** Renames box A to N.
VRename(A,x,N)**:** Renames wire x of box A to N.
Unfold(B,n,f) **:** Unfolds function f in match n of box B.
HieI(B,N)**:** Replaces box B by N which only holds B.
HCompI(A,B,N)**:** Horizontally composes box A and box B into N.
MatchVarI(B,i,o)**:** Replaces constants in inputs i and output o by a variable.
CaseI(B,i,j) **:** Replaces match i to j in box B by a case-expression.
IdI(B,v,N)**:** Introduces an identy box N to wire connected to v of box B.
HieE(B)**:** Replaces B with it's (only) child box.
HCompE$(B,[i_1,\cdots,i_n],[o_1,\cdots,o_m],X,Y)$ **:** Horizontally de-composes box B into boxes X and Y, where X has inputs $[i_1,\cdots,i_n]$ and outputs $[o_1,\cdots,o_m]$. Y will have the inputs/output of B not in $[i_1,\cdots,i_n]/[o_1,\cdots,o_m]$.
VCompE$(B,N,[o_1,\cdots o_n],M,[i_1,\cdots i_n])$ **:** Vertically de-composes box B into two sequentially composed boxes N and M, where N has B's inputs and $[o_1,\cdots o_n]$ as outputs, and M has $[i_1,\cdots i_n]$ as inputs and B's outputs.
MatchE(B,n)**:** Eliminates match n of box B.
CaseE(B,i) **:** Moves case expression in match i of box B into B's rule set
IdE(B)**:** Eliminates identity box B.
DupI(A,x,x',B,y,y') **:** Duplicates wire connecting x of box A and y of B, with wire names x' (of A) and y' (of B) respectively.
ThreadE(B,x,y) **:** Removes threading of through input x and output y of box B.