Higher-Order Symb Comput
DOI 10.1007/s10990-011-9067-y

Hume box calculus: robust system development through software transformation

Gudmund Grov · Greg Michaelson

Abstract Hume is a contemporary programming language oriented to systems with strong resource bounds, based on autonomous concurrent "boxes" interacting across "wires". Hume's design reflects the explicit separation of coordination and computation aspects of multi-process systems, which greatly eases establishing resource bounds for programs. However, coordination and computation are necessarily tightly coupled in reasoning about Hume programs. Furthermore, in Hume, local changes to coordination or computation, while preserving input/output correctness, can have profound and unforeseen effects on other aspects of programs such as timing of events and scheduling of processes. Thus, traditional program calculi prove inappropriate as they tend to focus exclusively either on the coordination of interacting processes or on computation within individual processes.

The Hume box calculus offers a novel approach to manipulating multi-process systems by accounting seamlessly for both coordination and computation in individual rules. Furthermore, the "Hierarchical Hume" extension enables strong locality of the effects of program manipulation, as well as providing a principled encapsulation mechanism.

In this paper, we present an overview of the Hume box calculus and its applications in program development. First of all, a base set of rules for introducing, changing, composing, separating and eliminating Hume boxes and wires, possibly within hierarchies, is presented. Next additional strategies are derived and a constructive approach to program development is illustrated through two examples of system elaboration from truth tables. Finally, at a considerably higher level, the use of the Hume box calculus to verify a generic transformation from a single box to an equivalent multi-box program, offering a balanced parallel implementation, is discussed.

Keywords Box calculus · Hume · Program transformation

G. Grov
University of Edinburgh, Edinburgh, UK
e-mail: ggrov@inf.ed.ac.uk

G. Michaelson (✉)
Heriot-Watt University, Edinburgh, UK
e-mail: G.Michaelson@hw.ac.uk

Published online: 09 June 2011

1 Introduction

1.1 Overview

Functional languages have long been heralded as a bridge between foundational theories of computing and practical system development. Lacking state, and drawing directly on classic theories of λ calculus and recursive functions, they are reputedly more amenable to formal manipulation than their imperative relations. Furthermore, the absence of stateful time dependencies gives rise to pervasive implicit parallelism, with the promise of minimising the complexities of constructing multi-process(or) systems.

Implicit parallelism in functional programs has generally proved too fine grain for effective exploitation. Instead, explicit coarser grain language constructs may be introduced, for example in Haskell through GpH's `par` combinator [32]. Similarly, Eden [6] extends Haskell with explicit process and channel constructs. A different approach is to take advantage of the close correspondence between higher order constructs and parallel algorithmic skeletons, either implicitly as in the PMLS compiler for Standard ML [28], or explicitly by crafting skeletons from lower level parallel constructs [23]. In both approaches, however, all aspects of parallelism are captured within a unitary notation.

Hume [17] is a contemporary programming language in the functional tradition, oriented to systems with strong resource bounds and well suited to multi-process systems. Unlike other multi-process functional languages, Hume's design, based on autonomous concurrent "boxes" interacting across "wires", reflects the explicit separation of coordination and computation aspects of multi-process systems, which greatly eases establishing resource bounds for programs. Nonetheless, coordination and computation are necessarily tightly coupled in reasoning about Hume programs. In particular, in Hume, local changes to coordination or computation, while preserving input/output correctness, can have profound and unforeseen effects on other aspects of programs such as timing of events and scheduling of processes. Thus, traditional program calculi prove inappropriate as they tend to focus exclusively either on the coordination of interacting processes or on computation within individual processes.

The Hume box calculus offers a novel approach to manipulating multi-process programs by accounting seamlessly for both coordination and computation in individual rules. It may be viewed operationally as characterising a rewrite system for program transformation. Furthermore, the "Hierarchical Hume" extension enables strong locality of the effects of program manipulation, as well as providing a principled encapsulation mechanism.

In this paper, we present an overview of the Hume box calculus and its applications in program transformation. First of all, we survey Hume and its execution model, develop motivating transformations and explain how they impacts on program behaviour in non-obvious ways. Next we provide a brief overview of Hierarchical Hume before we introduce the box calculus in Sect. 2. This is followed by a presentation of the syntax, semantics, and formal reasoning and mechanisation frameworks for the box calculus in Sect. 3. We then develop an exemplary set of box calculus rules and strategies of generic applicability in Sect. 4, and illustrate their use in the derivation of two very low level multi-box systems from truth tables in Sect. 5. We also deploy the box calculus, at a considerably higher level, to verify a generic transformation from a single box to an equivalent multi-box program, offering a balanced parallel implementation in Sect. 6. Finally, we discuss the wider context for this research in Sect. 7, and its limitations and future directions in Sect. 8.

This paper represents a substantive elaboration of [14], where the box calculus was first presented. In the current paper we provide considerable additional contextualisation, stronger characterisation of rules via type signatures, greater detail of the derivation of rules and strategies, and new applications to parallel program derivation.

1.2 Hume

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* [17], based on autonomous concurrent *boxes* linked by *wires* and controlled by generalised transitions. A major strength of Hume for establishing resource costs of programs lies in the explicit separation of inter-box *coordination* and intra-box *computation* concerns, but this is not considered further here.

Boxes and wires are defined in the simple finite state *coordination language* with transitions defined in the Turing-complete *expression language* through pattern matching and associated recursive actions. Both coordination and expression languages share a rich polymorphic type system, comparable to those of contemporary functional languages like Haskell and Standard ML.

In Hume, a program consists of unitary boxes connected by wires, where a box consists solely of stateless 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.

For example, consider a single box program that inputs sequences of 16 integers followed by a checksum, and displays the sequence sum and checksum:

```
 1. type integer = int 64;

 2. stream input from "std_in";
 3. stream output to "std_out";

 4. box checksum
 5. in (n,c,s::integer)
 6. out (c',s'::integer,m::(string,integer,string,integer))
 7. match
 8.  (0,s,n) -> (16,0,("sum: ",s,"checksum: ",n)) |
 9.  (c,s,n) -> (c-1,s+n,*);

10. wire checksum
11. (checksum.c' initially 16,checksum.s' initially 0,input)
12. (checksum.c,checksum.s,output);
```

Line 1 introduces the alias `integer` for the sized integer `int 64`.

Lines 2 and 3 link the streams `input` and `output` to standard input and standard output respectively.

Lines 4 to 9 define the box `checksum`. Line 5 introduces the input links `c` for the decrementing count of inputs, `s` for the accumulating sum of inputs and `n` for inputs. Similarly, line 6 introduces the outputs `c'` for the decrementing count, `s'` for the accumulating sum, and `m` for a message showing the final accumulating sum and checksum.

Line 8 matches the inputs in the case where the count is `0` and so the current input must be the checksum. Then the outputs reset the count to `16`, the sum to `0`, and display the sum and checksum with appropriate strings.

Line 9 matches the inputs where the count is non-zero. The count is decremented, the input added to the sum and no message is displayed.

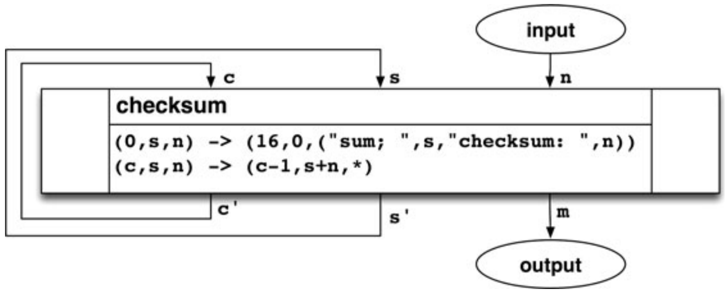


Fig. 1 Checksum program

Finally, lines 10 to 12 wire the program. Line 11 associates the box inputs `c` and `s` with the outputs `c'` and `s'`, initialising them respectively to `16` and `0`. It also associates the input `n` with standard input from `input`.

Then, line 12 associates the box outputs `c'` and `s'` with `c` and `s` respectively, and the output `m` with standard output to `output`.

The program is illustrated in Fig. 1.

Note that the box has feedback wires from `c'` to `c`, and from `s'` to `s`. Such wiring is also termed *self-wiring*.

1.3 Hume execution

The Hume execution model is based on two-stage, cyclical execution. On each cycle, all boxes first attempt to consume inputs to generate outputs once, and then all input/output changes are resolved in a unitary super-step.

Initially, all boxes are in a *Runnable* state. In the first execution stage, each box attempts to match the values on its input wires against successive patterns. When a pattern matches, variables are bound to corresponding wire values and the associated computation expression is evaluated. Note that at this stage, no inputs are consumed and no outputs are asserted.

Next, at the second stage (super step), all successfully matched values are consumed from input wires. Then, provided all relevant output wires from a box are free, its new output values are asserted on the corresponding wires and the box becomes *Runnable* once more.

Boxes which cannot assert all outputs, do not assert any, and become *Blocked*, buffering their pending outputs. On the next execution cycle, *Blocked* boxes do not attempt to match inputs at the first stage, but may assert their pending outputs on the superstep if other boxes have consumed appropriate inputs.

Note that the execution model is closely related to call-by-value-result parameter passing.

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;
– assert outputs until other boxes consume the corresponding inputs.

1.4 Transforming Hume programs

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.

Now, the main loci of transformation from an upper to a lower level is to move activity from computation to coordination, i.e. from expressions inside a box to wiring between boxes, typically reducing activity within a box but increasing the number of boxes in compensation. This increases the accuracy of behavioural modelling.

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 [27], a call to recursion within a PR-Hume box:

```
F x = if T x then U x else F (V x);
```

is replaced by wires from that box to a new FSM-Hume box using feedback wires to enable iteration equivalent to:

```
while not T x do
  x := V x;
return U x;
```

Figure 2 shows the transformation. 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` has transition `(x) -> (F x)` from input `x` to output `F x`.

After the transformation, a modified `RecBox'` communicates with a new iterating `IterBpx`. `RecBox'` has new matches:

– `(x,*) -> (*,x)`, to accept external input `x` and pass it to `IterBox` on wire `RecBox.o'/IterBox.i`;
– `(*,x) -> (x,*)`, to accept the final `x` back from `IterBox` on wire `IterBox.o/RecBox'.i'` and output it.

`IterBox` repeatedly applies `V` to `x` until `T` holds and then returns `U x` to `RecBox'`.

Note that 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 so long as `IterBox` has not produced any outputs. 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.

Even at the same level, simple local transformations may have substantial unintened effects on the rest of the program. For example, consider splitting (or joining boxes) vertically

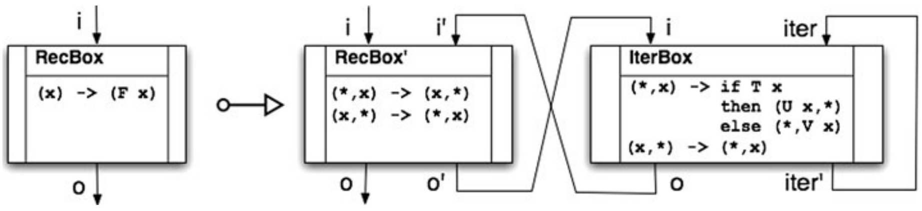


Fig. 2 Recursion to box iteration

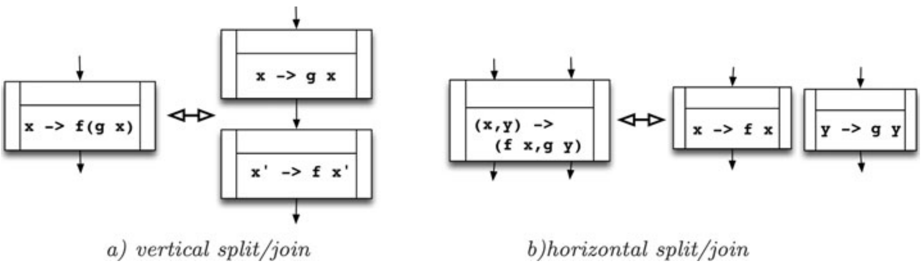


Fig. 3 Horizontal and vertical split/join transformations

or horizontally as shown in Fig. 3. In the horizontal case (b), the single box containing the tuple `(f x,g y)` may be split to enable concurrent execution of two auxiliary boxes to execute `f x` and `g y` independently. Here, although one box is introduced there will be no impact on execution as both will be in the same super step.

However, in the vertical case (a), there is considerable impact on the rest of the program. Here, one box containing a function composition `f(g x)` is split into a pipeline of boxes realising first `g x` and then `f(g x)`. Thus, an additional box and wire are introduced, and expression evaluation is shared between the first and second box. Note that, while input/output correctness may be preserved, without the introduction of hierarchy there will be a delay, maximally equivalent to executing *the entire program for one cycle*, between the first box asserting an output and the second box consuming it as an input.

1.5 Hierarchical Hume

In the *Hierarchical Hume* extension [15], 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.

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.

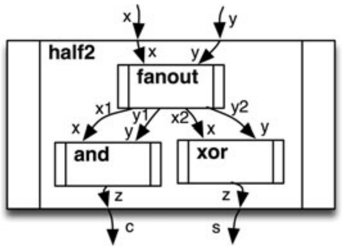
Hierarchical Hume is a minimal extension, allowing nested constructs of the form:

```
box name
 in links out links match matches
 boxes
  box id1 ...
```

```
-- 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

Fig. 4 Half Adders in Hierarchical Hume

```
wire id1 ...
box id2 ...
wire id2 ...
...
```

As before, the box has `in` and `out` links to other boxes and the external environment. However, the *matches* are now used to express constraints on links (e.g. constant, ignore (`*`), accept (_)) rather than to consume and generate values. Then, the new keyword `boxes` introduces an encapsulated sequence of box and wire definitions. Here, encapsulated boxes may be wired to each other, and to the encapsulating box, but not to the outer environment.

This simple Hume extension, as well as providing a valuable encapsulation mechanism for software design, enables the localisation of the effects of transformations. In particular, Hume hierarchies avoid timing disruption and additional global box scheduling attendant on the introduction of new boxes with local behaviours.

For example, consider transforming a single box half adder to a multi-box AND/XOR configuration, as shown in Fig. 4. 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 hierarchical box (b), the transition details within the box are elided.

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

Note that, in box `half2`, the "match" '`(_,_)->(_,_)`' means that the rule will fire when all inputs are available and complete when all outputs are available.

Informally, a transformation is correct if the program behaves the same way with respect to the environment as in the original configuration—i.e. the same values are produced and consumed for wires connected to external devices. Only first-level boxes can communicate with external devices, hence an example of a correct transformation is when the top level boxes observationally implement the first-level boxes before the transformation. In the example given in Fig. 4 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. As noted above, this is achieved by the '`(_,_)->(_,_)`' match of `half2`. It is easy to see that the `s` (first) output of `half1` with the given inputs is basically an XOR gate. Further, `c` 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` which therefore implements `half1`. In the example shown in Fig. 2, 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.

2 Overview of the box calculus

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.

In Hume, while boxes may be concurrently executed, concurrency is controlled by the static one-to-one wiring, which ensures freedom of race conditions. However, global analysis is still required when transforming boxes at the coordination layer since changes of the topology, although functionally behaviour preserving, may change *when* values arrive, which may have impact on the overall system behaviour—mainly since Hume allows us to ignore values with the `*` pattern. We have previously discussed this in [15], and this was in fact our key motivation for introducing Hierarchical Hume.

As a consequence, rules that may change the scheduling/timing properties, require additional preconditions asserting that the changes will have no effect. To ease the description of such cases we introduce the notion of *context* and *(un)bounded context*.

The *context* of a box captures the dependencies of a box—that is, where (timing) changes to a box or cluster of boxes may have an effect. If these can be localised, then the context is *bounded*. Unbounded context requires full global analysis of a program. An example of a box that has a bounded context, is a box nested by another box. Here, the dependent boxes can only be the siblings and (internal details) of the parent box—however, it may not be

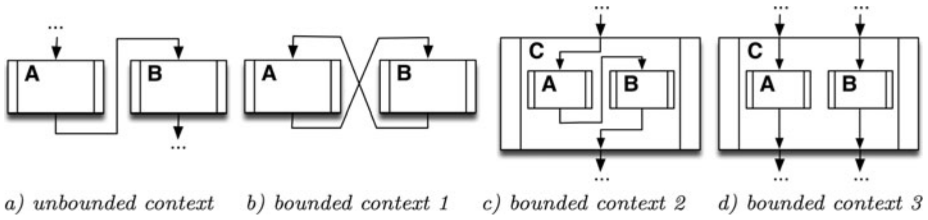


Fig. 5 Context and (un)bounded context illustration

all the siblings as illustrated in Fig. 5(d). Note that the context of a box in principle always contains itself in the general case (e.g. due to self-wiring). However, this may be ruled out by a analysis of the wiring. Figure 5 illustrates four context cases for a box A:

(a) in this case the context of A is at least A and B. The stippled lines illustrates that the remaining program is unknown—thus, full global analysis are required;
(b) boxes A and B forms a closed network—thus, the context of box A is bounded, and contains A and B. Note, that A is indirectly wired to itself via box B and is therefore part of the context;
(c) boxes A and B are nested by box C. Thus, the context of A is bounded and contains A, B and C (internally);
(d) boxes A and B are nested by box C, and the context of A is thus bounded. However, A and B are completely independent, thus the context is A and C (internally) only.

3 The rule syntax and semantics

Hume has formal semantics based upon the heap representation of values [22]. This heap representation underpins the Hume cost model (see e.g. [18]). Indeed, as satisfying resource bounds is a key motivation behind applying transformations, we plan to further integrate the calculus with the cost model through the future implementation of the *costing-by-construction* paradigm discussed in Sect. 8.

However, the Hume semantics focuses on the expression layer concerned with computations within boxes, and does not fully reflect the distinctive properties of the coordination layer. In contrast, the Hierarchical Hume semantics in TLA [24] presented in [12] focuses on the coordination layer and does not follow the standard heap representation. The TLA formalisation is also mechanised in Isabelle, and some experiments with integrating the mechanisation of [22] with the TLA embedding are conducted in [12]. However, this is based on a shallow embedding of the coordination layer, and is not an appropriate basis for the box calculus. Instead, here we have returned to [22], extended with hierarchies as formalised in TLA [12].

For our purposes, a Hume program configuration is considered as consisting of a triple

$$\langle \theta, \eta, bcs \rangle :$$

θ is the wire heap; η is the box heap disjoint from θ; while bcs is a list of box configurations. Our rules in [14] focuses on Hierarchical Hume. Here, the wire heap contains all first-level wires as illustrated in Fig. 6(a). To enable more flexibility, we have instead used the representation in Fig. 6(b), i.e. allowing all wires except those connected to external devices to change. This also enables better support for transforming programs without hierarchies. The

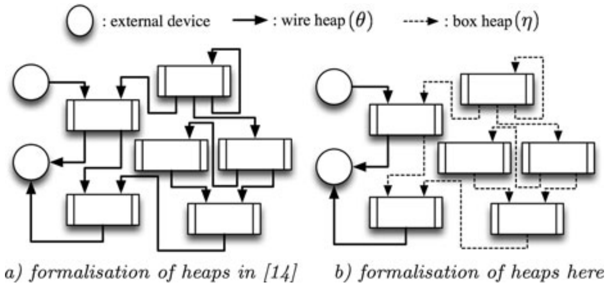

: external device
: wire heap (θ)
: box heap (η)
a) formalisation of heaps in [14]
b) formalisation of heaps here

Fig. 6 Wire heap θ and box heap η differences between [14] and this formalisation

distinction between θ and η is important in the formulation of the box calculus since only η is allowed to change. Note that, to ease the presentation we have omitted features of the Hume semantics that are not relevant for this discussion.

A heap is represented as a type H, which is a mapping from locations, of type *Locs*, to values or locations. To make this (partial) map into a total function, a *nil* element is also added. Thus, a heap H has the type:

$$H = Locs \to Vals \cup Locs \cup \{nil\}.$$

We abstract over the Hume type system, and assume all values (independent of types) are in the set *Vals*. However, note that the Hume type system is comparable to that of Haskell and Standard ML.

Let *BCS* be the type of a box configuration and [*BCS*] a list of box configurations. A program configuration has then the type:

$$\langle \theta, \eta, bcs \rangle :: H \times H \times [BCS].$$

Below we will abbreviate this type by *PC*:

$$PC = H \times H \times [BCS].$$

Each box configuration consists of the elements

$$\langle id, iws, ows, we, rs, ii, io, ibcs \rangle :$$

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; *we* is a wire environment linking wire identifiers to the correct element of the correct list (note that input and output wire names in a box must be distinct); *rs* is a list of matches. The three last elements are empty (lists) if the box is not nested: *ii* is a list of locations of internal input wires; *io* is a list of locations of internal output wires; and *ibcs* is a list of box configurations of internal (nested) boxes.

Let *Id* be the identifier type (e.g. string), and *RS* the type of a Hume match. A match consist of a pattern (type *Patt*) and expression (type *Expr*), where match can either be ignored, consumed, ignored but consumed if existing, a variable, a constructor or a constant:

$$Patt = \{*, \_, \_*\} \cup Vars \cup Constr \cup Consts.$$

Expr, *Vars*, *Constr* and *Consts* have the obvious meaning. Thus, a match has the type

$$RS = Patt \times Expr.$$

The wire environment is of type *WE*, it maps a variable name (type *Id*) to a list and an element of that list. To simplify we number the lists (e.g. 0 for *iws* and 1 for *ows*), thus the result is a pair of naturals. To achieve totality, *nil* is returned if the given variable is not valid:

$$WE = Id \to (\mathbb{N} \times \mathbb{N}) \cup \{nil\}.$$

A box configuration has then the type:

$$BCS = Id \times [Locs] \times [Locs] \times WE \times [RS] \times [Locs] \times [Locs] \times [Locs].$$

run_{bcs} represents one execution cycle of the program including the super step. It is a predicate on two heaps $\langle\langle\theta, \eta\rangle, \langle\theta', \eta'\rangle\rangle$ where $\langle\theta, \eta\rangle$ is a 'before heap' and $\langle\theta', \eta'\rangle$ an 'after heap'. 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

$$\langle\theta, \eta, bcs\rangle \vdash \mathbf{Rule}(X_1, \ldots, X_n) \Downarrow \langle\theta', \eta', bcs'\rangle.$$

This is read as "**Rule** with parameters $X_1, \ldots, X_n$ will, under the configuration $\langle\theta, \eta, bcs\rangle$ create the configuration $\langle\theta', \eta', bcs'\rangle$". This can also be written:

$$\langle\theta', \eta', bcs'\rangle = (\mathbf{Rule}(X_1, \ldots, X_n))(\langle\theta, \eta, bcs\rangle)$$

meaning that the program configuration is an implicit argument of **Rule**. Let T_i be the type of X_i above. **Rule** has then the type:

$$\mathbf{Rule} :: T_1 \times \cdots \times T_n \to PC \to PC.$$

Semantically, a *strategy* is the same as a rule—with the intention that it represents something "higher level". The plan is to have a (closed) set of primitive rules. This set can be used for theoretical results, like completeness. From this set of primitive rules, higher-level strategies are derived and these will be used in actual program transformation. The use of inference rules and tactics in LCF-style theorem proving can be seen as an analogy to rules and strategies.[2]

A *function*, on the other hand, does not change the program configuration. Furthermore, in many cases a function does not need all the program configuration. Thus, the configuration is not an implicit argument as the case is for a rule. Instead, a function will always produce a result. For example, let T be the result type of **Function**. It will then have the type:

$$\mathbf{Function}(X_1, \ldots, X_n) :: T_1 \times \cdots \times T_n \to T$$

3.1 A framework for correctness verification

The Temporal Logic of Action (TLA) [24] allows us to separate computation and coordination for reasoning, and fits well into the Hume framework [15, 19]. The TLA specification for the Hume program triple $\langle\theta, \eta, bcs\rangle$ is

$$\exists\eta : Init_\theta \wedge Init_\eta \wedge \Box(run_{bcs}).$$

[2]Technically, an inference rule and a tactic have different types, so a more correct analogy would be an inference rule wrapped by a tactic, and a "normal" tactic.

$\exists\eta$ denotes that η is *hidden*. $Init_\theta \wedge Init_\eta$ holds the initial values for the two heaps. The key to proving transformations is allowing steps that do not change the state space $\langle\theta, \eta\rangle$. Such steps are called *stuttering steps*, and are internal actions. Validity of formula should not depend on those, and such formulas are said to be *invariant under stuttering*. Thus, run_{bcs} will henceforth denote that if *bcs* does not hold between a before and an after state of an action, then $\langle\theta, \eta\rangle$ is left unchanged. run_{bcs} must hold throughout execution and has therefore been prefixed by the temporal 'always' operator '$\Box$'.

A transformation of $\langle\theta, \eta, bcs\rangle$ into $\langle\theta', \eta', bcs'\rangle$ (by application of a rule) is here seen as a *refinement*, or in other words an *implementation*. Here,

$$\big(\exists\eta : Init_\theta \wedge Init_\eta \wedge \Box(run_{bcs})\big). \tag{1}$$

can be seen as specifying the behaviour which

$$\big(\exists\eta' : Init_{\theta'} \wedge Init_{\eta'} \wedge \Box(run_{bcs'})\big) \tag{2}$$

must preserve. Thus, (2) is a correct transformation of (1) if (2) implements (1). A proof of implementation in TLA, is represented by implication. Thus, the transformation proof is formalised as:

$$\big(\exists\eta' : Init_{\theta'} \wedge Init_{\eta'} \wedge \Box(run_{bcs'})\big) \Rightarrow \big(\exists\eta : Init_\theta \wedge Init_\eta \wedge \Box(run_{bcs})\big). \tag{3}$$

The introduction/elimination rules for $\exists$ is similar to those for $\exists$ in standard predicate logic—however $\exists$ has a more complex semantics since it has to be invariant under stuttering. Hence, we must find a witness for η and introduce a Skolem constant for η' in (3). The witness is called a *refinement mapping*, and $\overline{F}$ is used for F under this refinement mapping, i.e. after the application of the introduction rule for $\exists$. We follow this syntax, and let $\overline{\eta}$ be this witness. For simplicity, we use $\overline{\eta'}$ for the Skolem constant as well. $run_{\overline{bcs}}$ ($run_{\overline{bcs'}}$) represents running *bcs* (*bcs'*) with $\overline{\eta}$ ($\overline{\eta'}$) replacing η (η').

To give a simple example, lets say there are two wires x and y in the box heap η,[3] and x and y are connected to the same boxes (with the same direction), and always have the same values. Thus, y is a duplicate of x and may be removed. A result of applying such a duplication elimination rule is that in the prior heap η, we have e.g. that $\eta(x) = \eta(y)$. However, since x has been eliminated in η', it will always be the case that $\eta'(y) = nil$, regardless of x. Thus, the witness of η in (3) is

$$\overline{\eta} = \big(\lambda z.\ \texttt{if}\ z = y\ \texttt{then}\ \overline{\eta}'(x)\ \texttt{else}\ \overline{\eta}'(z)\big).$$

Note that since η' is bound in the hypothesis, the $\exists$ elimination rule will introduce a Skolem constant, which we assume is $\overline{\eta}'$. In most cases, the witnesses are much more complex, and often requires the introduction of *auxiliary variables* [1]. This example has illustrated the distinction of the box and wire heaps for the calculus—and the importance of reducing the wire heap.

The proof of the correctness of the rules derived in the next section have a complex underlying TLA machinery. Full details are beyond the scope of this paper. In particular, hierarchies must be flattened for us to prove that a transformation is in fact correct, i.e. global and local steps are no longer separated at this level.

[3] For simplicity, let x and y be locations and not identifiers. Remember, the box heap contains wires that are not connected to external devices.

We then use an induction principle to prove (3) above: initially, the new heaps must be stronger than before the transformation[4]—and all actions updating the heaps must be stronger than the actions before the transformation:

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

where

$$\big(\langle \theta', \overline{\eta}' \rangle \Rightarrow_S \langle \theta, \overline{\eta} \rangle\big) \Leftrightarrow \big(\forall x \in \mathbf{dom}(\theta), y \in \mathbf{dom}(\overline{\eta}) : \ \theta(x) = \theta'(x) \wedge \overline{\eta}(y) = \overline{\eta}'(y)\big),$$

and

$\mathbf{dom}(h) :: H \to \mathbb{P}(Locs)$
Returns the domain of heap h, i.e. $\mathbf{dom}(h) = \{l \mid h(l) \neq nil\}$. $\mathbb{P}$ is the powerset.

Note that the primed components are the translated ones. Further, $\Rightarrow_T$ is an instantiation of standard TLA rules for our purpose. Thus, its soundness therefore follows from the soundness of TLA. An important feature, which underpins the calculus, is the transitivity of $\Rightarrow_T$:

Theorem 1 (transitivity) $\langle \theta, \eta, bcs \rangle \Rightarrow_T \langle \theta', \eta', bcs' \rangle$ *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. □

4 Rule & strategy derivations

Here, we formally derive some rules and strategies of the calculus. A summary of functions used are listed in Appendix A while all rules and strategies that are applied in this paper are listed in Appendix B. Details, such as pre-conditions, have been elided from 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 postfixed by 'E'.

In the rule derivation we give an informal graphical representation of the impact of the rule. In the graphical representation we do not show any potential siblings or parents of relevant boxes.

4.1 Derivation of **HieI**

The first rule we consider, **HieI**, introduces hierarchy by nesting one box B inside another box A with name N:

$\mathbf{HieI}(B, N) :: Id \times Id \to PC \to PC$
Replaces box B by N which only holds B.

[4] A heap H' is stronger than heap H if all properties P of H' also holds for H:

$$(H' \ stronger\ than\ H) \Leftrightarrow \big(\forall P.P(H') \Rightarrow P(H)\big).$$

The following functions and rules are used to derive it:

Gen_Rules$(rs) :: [RS] \to [RS]$
Returns a *generalisation* of the rule set *rs*. This generalisation works by replacing pattern variables by '_', while the rest is unchanged. In an expression, everything but '$*$' is replaced by '_', and all function calls are removed. Furthermore, any duplicate matches as a result of this are omitted.

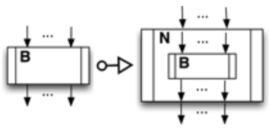
Get_Box$(B, bcs) :: Id \times [BCS] \to BCS$
Returns box configuration with box id B from list *bcs*.

HeapLocs_Copy$([l_1, \dots, l_n], H_1, H_2) :: [Locs] \times H \times H \to H \times [Locs]$
A deep copy is made of the locations $[l_1, \dots, l_n]$ of heap H_1 into heap H_2. A pair of the updated H_2 together with the locations of the copied elements is returned.

Replace$\big([A_1, \dots, A_n], [B_1, \dots, B_m]\big) :: [Id] \times [Id] \to PC \to PC$
Replaces boxes $A_1, \dots, A_n$ by $B_1, \dots, B_m$.

From the types we can see that **Replace** is the only rule—the others are functions. This rule introduces a *bounded context* for B, only consisting of N (internally) and B. By applying this rule we can ignore the top level timing dependencies when transforming B. 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**:[5]

$$\frac{\begin{array}{c}\langle B, iws, ows, rs, iw, ow, ibcs\rangle = \textbf{Get\_Box}(B, bcs)\\ \langle \eta_1, niw\rangle = \textbf{HeapLocs\_Copy}(iws, \theta, \eta)\\ \langle \eta_2, now\rangle = \textbf{HeapLocs\_Copy}(ows, \eta_1, \eta_1)\\ \langle \theta, \eta, bcs\rangle \vdash \textbf{Replace\_BoxHeap}(\eta_2) \Downarrow \langle \theta, \eta'', bcs\rangle\\ B' = \langle B, niw, now, rs, iw, ow, ibcs\rangle \qquad irs = \textbf{Gen\_Rules}(rs)\\ A = \big\langle N, iws, ows, irs, niw, now, [B']\big\rangle\\ \langle \theta, \eta'', bcs\rangle \vdash \textbf{Replace}\big([A], [B]\big) \Downarrow \langle \theta, \eta', bcs'\rangle\end{array}}{\langle \theta, \eta, bcs\rangle \vdash \textbf{HieI}(B, N) \Downarrow \langle \theta, \eta', bcs'\rangle}$$



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

Theorem 2 (**HieI** correctness)

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

Proof Since we only extend $\overline{\eta}$ and do not change θ, $\langle \theta', \overline{\eta'}\rangle \Rightarrow_S \langle \theta, \overline{\eta}\rangle$ holds (it follows from $\langle \theta', \eta'\rangle \Rightarrow_S \langle \theta, \eta\rangle$). In *bcs*, B is replaced by A. Since A's rule set generalises B's the

[5]This rule only applies to top-level boxes. The only difference with nested boxes, is that the source and destination heap for the **HeapLocs_Copy** function would both be η. As an alternative, we could have abstracted over the heap and computed which source heap should have been given to **HeapLocs_Copy**—it is easy to check if the box is nested using *bcs*. This is also the case for use of **HeapLocs_WireFree/HeapLocs_BoxFree** and **Replace_WireHeap/Replace_BoxHeap**. However, we believe that the more concrete approach taken here eases the reading.

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. □

4.2 Derivation of **HieE**

The second derivation is for **HieE**. This dual of **HieI** eliminates a layer of a hierarchy:

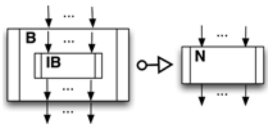
$\mathbf{HieE}(B, N) :: Id \times Id \rightarrow PC \rightarrow PC$
Replaces B with it's (only) child box and names this new box N.

The following functions and rules not already discussed are used to derive it:

$\mathbf{HeapLocs\_BoxFree}([l_1, \ldots, l_n]) :: [Locs] \rightarrow PC \rightarrow PC$
Frees locations $[l_1, \ldots, l_n]$ from the box heap.
$\mathbf{len}(L) :: [\alpha] \rightarrow \mathbb{N}$
Returns the length of list L. α indicate type variable, thus the function is polymorphic.
$\mathbf{Match\_eq}(rs_1, rs_2) :: [RS] \times [RS] \rightarrow \mathbb{B}$
Holds if the patterns of rs_1 and rs_1 match and consume the same inputs.
$\mathbf{Terminates\_eq}(B_1, B_2, bcs) :: Id \times Id \times [BCS] \rightarrow \mathbb{B}$
Holds if box B_1 and B_2 have the same termination conditions.

HieE assumes that one box is nested by another box, with no internal wires that are not connected to the parent. Moreover, the matching (and wire consumption) of the parent box must "behave similiarly" as the child box (achieved by **Match_eq**). The internal wires of the parent box, as well as the parent box, are then removed—and the (external) wires of the parent box are wired directly to the inner box:

$$\frac{\begin{array}{c}\langle B, iws, ows, rs, iw, ow, [IB]\rangle = \mathbf{Get\_Box}(B, bcs)\\ \langle IB, iiws, iows, irs, iiw, iow, ibxc\rangle = \mathbf{Get\_Box}(IB, bcs)\\ \mathbf{Match\_eq}(rs, irs) \wedge \mathbf{Terminates\_eq}(B, IB, bcs)\\ \mathbf{len}(iws) = \mathbf{len}(iiws) \wedge \mathbf{len}(ows) = \mathbf{len}(iows)\\ A = \big\langle N, iws, ows, irs, iiw, iow, ibxc\big\rangle\\ \langle\theta, \eta, bcs\rangle \vdash \mathbf{HeapLocs\_BoxFree}(iiws) \Downarrow \langle\theta, \eta'', bcs\rangle\\ \langle\theta, \eta'', bcs\rangle \vdash \mathbf{HeapLocs\_BoxFree}(iows) \Downarrow \langle\theta, \eta', bcs\rangle\\ \langle\theta, \eta', bcs\rangle \vdash \mathbf{Replace}\big([A], [B]\big) \Downarrow \langle\theta, \eta', bcs'\rangle\end{array}}{\langle\theta, \eta, bcs\rangle \vdash \mathbf{HieE}(B, N) \Downarrow \langle\theta, \eta', bcs'\rangle}$$



Theorem 3 (**HieE** correctness)

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

Proof Firstly, θ is not changed. The witness $\overline{\eta}$ on the other hand, is more complicated than in **HieI**—since part of the heap is freed. The proof would follow the same approach as in [12, pp. 107–109], where an ad-hoc transformation was verified by model checking (the witness still had to be provided manually).

Now, the correctness of this transformation follows from the observation that (the freed) *iiws* mimic *iws*, and (the freed) *ows* mimic *iows*. This is explored in the witness. However, *iws* is consumed when *iiws* is written—while *iows* is consumed when *ows* is written. Thus, we need to introduce auxiliary (ghost) variables [1] to the (specification of the) "before" program, which holds the "old values" of e.g. *iws*. In $\overline{\eta}$, the witness is then over the auxiliary variables rather than the actual variables.

Initially, $\langle\theta', \overline{\eta}'\rangle \Rightarrow_S \langle\theta, \overline{\eta}\rangle$ holds since all nested wires are empty, or with a known (constant) initial value. Since **Match_eq**(*rs*,*irs*), **Terminates_eq**(B, IB) and the computation of the outer box is achieved by the inner box, it is clear that the matching, computation and termination will be the same. With the witness for the freed inner variables described above it is then clear that $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ holds. □

4.3 Derivation of **HCompI**

In the third derivation, for **HCompI**, two non-nested boxes, A and B, are horizontally composed into a new box called N:

HCompI(A, B, N) :: $Id \times Id \times Id \rightarrow PC \rightarrow PC$
Horizontally composes box A and box B into N. There is an implicit renaming of variables in the patterns of B if there are name clashes with variables of A.

The following new functions and rules are used in the derivation:

is_Blocked(C, B) :: $PC \times Id \rightarrow \mathbb{B}$
Holds if box B cannot be executed.

mutually_exclusive(*rs*) :: $[RS] \rightarrow \mathbb{B}$
Holds if the patterns of rule set *rs* are mutually exclusive.

project($[(p_1 \rightarrow e_1), \ldots, (p_n \rightarrow e_n)], [(p'_1 \rightarrow e'_1), \ldots, (p'_m \rightarrow e'_m)]$) :: $[RS] \times [RS] \rightarrow [RS]$
Pairwise combines each pattern p_i and p'_j with e_i and e'_j where $i \in 1..n$ and $j \in 1..m$. For example,
project($[p_1 \rightarrow e_1], [p_2 \rightarrow e_2, p_3 \rightarrow e_3]$) $= [(p_1, p_2) \rightarrow (e_1, e_2), (p_1, p_3) \rightarrow (e_1, e_3)]$

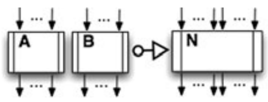
$L_1 @ L_2 :: [\alpha] \times [\alpha] \rightarrow [\alpha]$
Concat list L_1 in front of list L_2.

$\Box_H F :: (PC \rightarrow \mathbb{B}) \rightarrow PC \rightarrow \mathbb{B}$
This represents the temporal 'always' operator, denoting that $F :: PC \rightarrow \mathbb{B}$ must hold henceforth, abstracted over the program specification. Due to the temporal property, a possible world semantics is used, thus F is a predicate over a program configuration. Translated into TLA, the meaning becomes:

$$(\lambda\langle hw,hb,bcs\rangle.\ \Box_H F) \Leftrightarrow (\lambda\langle hw,hb,bcs\rangle.\ Init_{hw} \wedge Init_{hb} \wedge \Box(run_{bcs}) \Rightarrow \Box F)$$

A and B must always have the same *Blocked* status, since N will be *Blocked* if either of them is. 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 B's 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 fails to pattern match the inputs, and vice verse. This is captured by postfixing the composed rule set below 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:

$$\frac{\begin{array}{c}\langle A, iws_A, ows_A, rs_A, [], [], []\rangle = \mathbf{Get\_box}(A, bcs)\\ \langle B, iws_B, ows_B, rs_B, [], [], []\rangle = \mathbf{Get\_box}(B, bcs)\\ \Box_H\big(\lambda\langle hw,hb,\_\rangle.\ \mathbf{is\_Blocked}(\langle hw,hb,bcs\rangle, A) = \mathbf{is\_Blocked}(\langle hw,hb,bcs\rangle, B)\big)(\langle\theta, \eta, bcs\rangle)\\ \mathbf{mutually\_exclusive}(rs_A) \qquad \mathbf{mutually\_exclusive}(rs_B)\\ iws = iws_A @ iws_B \qquad ows = ows_A @ ows_B\\ n_A = \mathbf{len}(iws_A) \quad m_A = \mathbf{len}(ows_A)\\ n_B = \mathbf{len}(iws_B) \quad m_B = \mathbf{len}(ows_B)\\ *_A = [\underbrace{\langle *, \ldots, *\rangle}_{n_A} \to \underbrace{\langle *, \ldots, *\rangle}_{m_A}] \quad *_B = [\underbrace{\langle *, \ldots, *\rangle}_{n_B} \to \underbrace{\langle *, \ldots, *\rangle}_{m_B}]\\ rs = \mathbf{project}(rs_A, rs_B)\ @\ \mathbf{project}(rs_A, *_B)\ @\ \mathbf{project}(*_A, rs_B)\\ N' = \langle N, iws, ows, rs, [], [], []\rangle\\ \langle\theta, \eta, bcs\rangle \vdash \mathbf{Replace}([N'], [A, B]) \Downarrow \langle\theta, \eta, bcs'\rangle\end{array}}{\langle\theta, \eta, bcs\rangle \vdash \mathbf{HCompI}(A, B, N) \Downarrow \langle\theta, \eta, bcs'\rangle}$$



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 establishment of the *Blocked* status requires a temporal invariance proof. This has therefore been prefixed by the temporal 'always' operator '$\Box_H$'. Note that during execution the Hume coordination layer is static—thus program execution only changes the values on the heap (remember, that all values are on the heap due to the cost model). However, $\Box_H$ has a more general definition, and in this case the *bcs* input is ignored.

Theorem 4 (**HCompI** correctness)

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

Proof There is no nesting, hence $\overline{\eta} = \eta$. Further, it is obvious that $\theta' = \theta$ and $\eta' = \eta$, thus $\langle\theta', \overline{\eta'}\rangle_S \Rightarrow \langle\theta, \overline{\eta}\rangle$. The proof of $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ is by case-analysis on the "execution state" of A and B: Since

$$\Box_H\big(\lambda\langle hw,hb,\_\rangle.\ \mathbf{is\_Blocked}(\langle hw,hb,bcs\rangle, A) = \mathbf{is\_Blocked}(\langle hw,hb,bcs\rangle, B)\big)(\langle\theta, \eta, bcs\rangle)$$

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 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. $\Box$

4.4 Derivation of **VCompE**

The next two rules are complicated since they change the timing behaviour, as explained above. The first rule **VCompE**, performs vertical decomposition to produce two (non-nested) boxes:

VCompE$(B, N, [o_1, \ldots, o_n], M, [i_1, \ldots, i_n]) :: Id \times Id \times [Id] \times Id \times [Id] \to PC \to PC$
Vertically de-composes box B into two sequentially composed boxes N and M, where N has B's inputs and $[o_1, \ldots, o_n]$ as outputs, and M has $[i_1, \ldots, i_n]$ as inputs and B's outputs.

The following functions that have not been discussed yet, are required:

Alloc$([S_1, \ldots, S_n], H) :: [\mathbb{N}] \times H \to [Locs] \times H$
Accepts as input a list of natural numbers and a heap. Each number represent the size of a data type. Note that for mutable data structures, such as lists, this will be the size in the initial program configuration. For each such S_i $(i \in (1..n))$, a heap region of size S_i is allocated, and the list of newly allocated locations $[l_1, \ldots, l_n]$ is returned, together with the new heap.
NoTime_Dependency$(B, bcs) :: Id \times [BCS] \to \mathbb{B}$
There is no "time dependency" from box B. This implies that we can add identity boxes on the inputs and outputs of B, without this having any effect on the functional behaviour of the Hume program. We use a restrictive form of this predicate where we assume that **NoTime_Dependency**(B, bcs) holds iff:

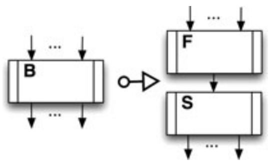
– there are no *s in any patterns of boxes in bcs which the output of B is directly or indirectly wired to, and
– B is not directly connected to an output in heap θ.

Out_Type_Size$(e) :: Expr \to \mathbb{N}$
Returns the return type of an expression/function.
Patt_of_Expr$(e) :: Expr \to Patt$
Returns a pattern from an expression. The pattern will consist of (distinct) variable(s) to be bound.

Note that with the restrictive definition of **NoTime_Dependency**, there may be cases where the above definitions does not hold, but the potential "bad" behaviour can be ruled out with an invariant or some dynamic properties. Finding a less restrictive definition is a subject for future work.

The main two side conditions of the rule are: (1) the (flat) box contains one match, with an expression of the form $g(f(e))$, i.e. in a pure functional setting the expression is a composition of (at least) two functions; (2 there is no timing dependency on the output of the box being decomposed. This means that if it takes one extra/less step (or any extra/fewer steps in general) to compute the result, then this will not have any effect on the functional behaviour of the other boxes. The rule is derived as follows:

$$\frac{\begin{array}{c}\langle B, iws, ows, [patt \to g(f(e))], [], [], [] \rangle = \textbf{Get\_box}(B, bcs) \\ \textbf{NoTime\_Dependency}(B, bcs) \\ p' = \textbf{Patt\_of\_Expr}(f(e)) \\ \langle [l], \eta_1 \rangle = \textbf{Alloc}([\textbf{Out\_Type\_Size}(f(e))], \eta) \\ \langle \theta, \eta, bcs \rangle \vdash \textbf{Replace\_BoxHeap}(\eta_1) \Downarrow \langle \theta, \eta', bcs \rangle \\ F = \langle F, iws, [l], [patt \to f(e)], [], [], [] \rangle \\ S = \langle S, [l], ows, [p' \to g(p')], [], [], [] \rangle \\ \langle \theta, \eta', bcs \rangle \vdash \textbf{Replace}([B], [F, S]) \Downarrow \langle \theta, \eta', bcs' \rangle \end{array}}{\langle \theta, \eta, bcs \rangle \vdash \textbf{VCompE}(B, F, S) \Downarrow \langle \theta, \eta', bcs' \rangle}$$



Theorem 5 (**VCompE** correctness)

$$\begin{array}{ll} \textit{If} & \langle \theta, \eta, bcs\rangle \vdash \textbf{VCompE}(B, F, S) \Downarrow \langle \theta', \eta', bcs'\rangle \\ \textit{then} & \qquad \langle \theta', \eta', bcs'\rangle \Rightarrow_T \langle \theta, \eta, bcs\rangle \end{array}$$

Proof This rule is function decomposition lifted to the box level. The matching/consume property of the new F/S component is that of F—which has the same pattern as B, and the output/blocking properties are that of S—which is the same resulting expression as in B (under an invariant on the intermediate step). The connection between them is ensured to be correct by **Patt_of_Expr**, while **Out_Type_Size** ensures the heap allocates the correct amount of memory. The main difficulty in the proof is the timing issue: F/S will require 2 steps to produce the output which B only requires 1 step to produce. Thus, it will also block and unblock one step later.

The key to correctness is, as in the proof of **HieE**, to find a witness which can abstract away the timing delay. We will here argue for why such witness exists with the given assumptions, drawing upon experiences in [12, pp. 107–109], where a transformation was mechanically verified (model checked) following a similar approach. Firstly, note that it is simpler than **HieE** in the sense that no heap space is freed—however, the (possible) delay parts are (much more) complicated.

Again, the proof relies on the introduction of auxiliary (ghost) variables [1], which records the changes of the variables. This "log" must also take care of the blocked status when recording the values. In the witness, this log is used instead of the actual program/wire variables.

This approach relies on the fact that the same values will appear on all variables (and in the same order)—the only deviation is *when* they appear. This is ensured by the **NoTime_Dependency** predicate as follows. All variables are updated by boxes which are pure functions without side-effects. Since all boxes except B are left unchanged, all variables except those from S are the same. Following an inductive principle, we assume that all variables so far have been written in the same order and the same value is written (note that we cannot simply use the Cartesian product of all variables due to the timing issue). The only way a box can then produce a different result, is if a different match is triggered. Since this is deterministic, it can only occur with different output. Now, since the values are guaranteed (by the induction hypothesis) to be the same, the only case left is that another match triggers due to a missing input value. This can only happen if the pattern contains $*$ (if not it will simply be a *Matchfail*), which violates the **NoTime_Dependency** assumption.

Thus, since θ is not changed, η is just extended, thus $\langle \theta', \overline{\eta'}\rangle \Rightarrow \langle \theta, \overline{\eta}\rangle$. Since functionally the result will always be the same, and using the auxiliary variables and witness above, $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ also holds. □

4.5 Derivation of **VCompI**

The next rule **VCompI** is the dual of **VCompE**, and introduces composition from two boxes.

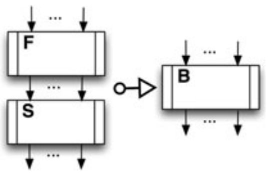
VCompI$(F, S, B) :: Id \times Id \times Id \to PC \to PC$
Vertically composes boxes F and S into box B. It assumes that F is directly followed by S. The input of the new box B is the input of F and the output is the output of S.

It requires the following new functions and rules

is_Empty$(H, L) :: H \times [Loc] \rightarrow \mathbb{B}$
  Holds if $\forall l \in L.H(l) = nil$.
WiresE$(A, xs) :: Id \times Id \rightarrow PC \rightarrow PC$
  Eliminates wires xs of box A.

and is derived as follows:

$$\frac{\begin{array}{c} F = \langle F, iws_F, ows_F, [patt_F \rightarrow f(e)], [], [], [] \rangle \\ S = \langle S, iws_S, ows_S, [patt_S \rightarrow g(patt_S)], [], [], [] \rangle \\ \textbf{NoTime\_Dependency}(S, bcs) \\ iws_S = ows_F \wedge \textbf{is\_Empty}(\eta, ows_F) \\ N = \langle N, iws_F, ows_S, [patt_F \rightarrow g(f(e))], [], [], [] \rangle \\ \langle \theta, \eta', bcs \rangle \vdash \textbf{Replace}([F, S], [B]) \Downarrow \langle \theta, \eta, bcs' \rangle \\ \langle \theta, \eta, bcs' \rangle \vdash \textbf{WiresE}(F, ows_F) \Downarrow \langle \theta, \eta', bcs' \rangle \end{array}}{\langle \theta, \eta, bcs \rangle \vdash \textbf{VCompI}(F, S, B) \Downarrow \langle \theta, \eta', bcs' \rangle}$$



Theorem 6 (**VCompI** correctness)

$$\begin{array}{ll} \textit{If} & \langle \theta, \eta, bcs \rangle \vdash \textbf{VCompI}(F, S, B) \Downarrow \langle \theta', \eta', bcs' \rangle \\ \textit{then} & \qquad \langle \theta', \eta', bcs' \rangle \Rightarrow_T \langle \theta, \eta, bcs \rangle \end{array}$$

Proof The proof follows the same approach as the proof of **VCompE**. In addition, as in **HieE**, parts of the heap are freed—thus the proof of **VCompE** is combined with aspect of the proof of **HieE**. The functional correctness of the box-level function composition is direct (since g is a function directly on the input pattern). The timing issue is handled as in **VCompE**—with addition of handling the freed variables ows_F (which are the same heap locations as iws_S). Again, this is handled by auxiliary variables, which "remember" the input, and compute $f(iws_F)$—which will become the result of box F, and later written to ows_F. Now, the intermediate step that updates ows_F is removed in the new configuration. To "simulate" this, a notion of auxiliary variables called *stuttering variables* [1], which introduces steps that leaves all but the auxiliary variables unchanged is used. Using them, the intermediate step is achieved—since only the auxiliary "log" is changed here—and no program variables. The rest of the witness is as in **VCompE**. Note that **is_Empty**(η, ows_F) asserts that the wires connecting F and S are initially empty. □

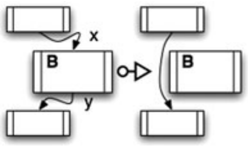
4.6 Derivation of **ThreadE**

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 (see discussion on the distinction between rules and strategies in Sect. 3). An example of a strategy, although still rather low-level, is the elimination of *threading* **ThreadE**:

ThreadE$(B, x, y) :: Id \times Id \times Id \rightarrow PC \rightarrow PC$
  Replaces threading of input x and output y through box B by direct wire.

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 eliminated, 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 the thread is eliminated. This must again not have any impact on the context. Since the rule is derived from other rules these precondition can be ignored as 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**:

$$\frac{\langle\theta,\eta,bcs\rangle \vdash \mathbf{HCompE}(B,[x],[y],Id,B) \Downarrow \langle\theta_1,\eta_1,bcs_1\rangle \quad \langle\theta_1,\eta_1,bcs_1\rangle \vdash \mathbf{IdE}(Id) \Downarrow \langle\theta',\eta',bcs'\rangle}{\langle\theta,\eta,bcs\rangle \vdash \mathbf{ThreadE}(B,x,y) \Downarrow \langle\theta',\eta',bcs'\rangle}$$



It uses the identity elimination rules, which has not been previously used:

IdE$(B) :: Id \rightarrow PC \rightarrow PC$
Eliminates identity box B.

The correctness proof for strategies are trivial since they only rely on the transitivity theorem (Theorem 1):

Theorem 7 (**IdE** correctness)

$$\begin{array}{ll} \textit{If} & \langle\theta,\eta,bcs\rangle \vdash \mathbf{ThreadE}(B,x,y) \Downarrow \langle\theta',\eta',bcs'\rangle \\ \textit{then} & \langle\theta',\eta',bcs'\rangle \Rightarrow_T \langle\theta,\eta,bcs\rangle \end{array}$$

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

5 Examples

We now illustrate the calculus by applying it to two simple examples based on single bit adder circuits. The first example shows the decomposition of a half adder box into a simple binary tree of three elementary logic gate boxes. The second example shows the decomposition of a full adder box into two half adders and an elementary logic gate, but with staged mutual dependencies. The examples are chosen to illustrate both the deployment of a characteristic range of base rules, and the use of strategies.

5.1 Example 1: half-adder

First we apply the box calculus to the half adder example above. We will do so stepwise, and a graphical representation of each of these steps is shown in Fig. 7. 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.

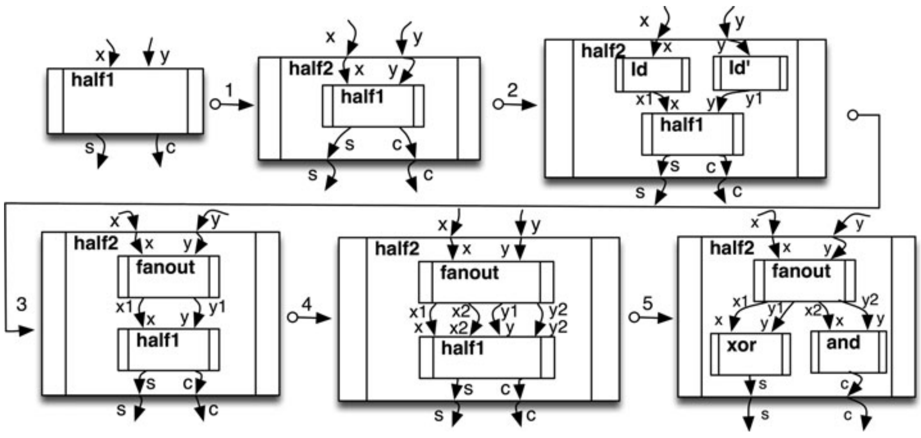


Fig. 7 Transformation of half adder

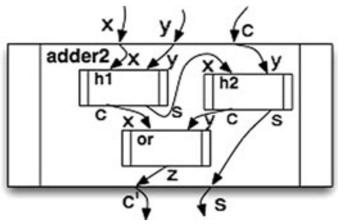
1. Since the transformation has a forward direction we start with the box shown in Fig. 4a. First rule **HieI**(`half1`, `half2`) which replaces box `half1` with a box `half2` that simply nests it.
2. Since there is no '$*$' in the context nested by `half2` there are no dependencies. We can therefore 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 which 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 sets 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 these boxes is exactly the same as the `xor` while the second is the same as the `and` box of Fig. 4b/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.

```
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 Adders 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

Fig. 8 Full adders in Hierarchical Hume

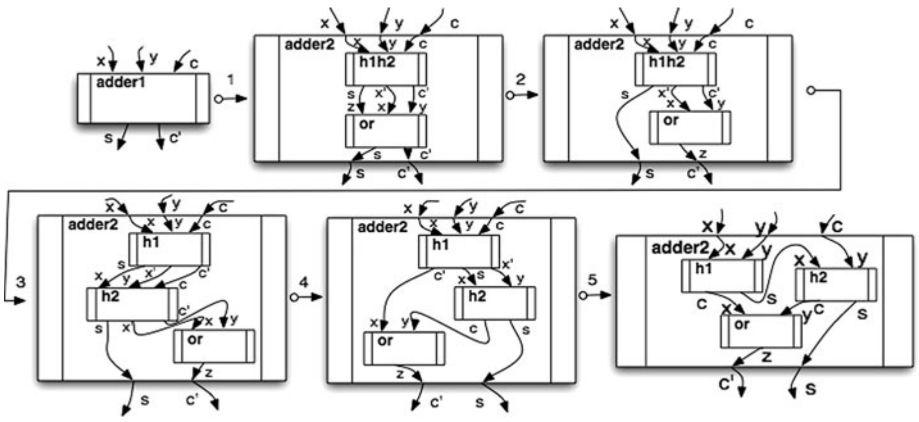


Fig. 9 Transformation of full adder

5.2 Example 2: a full adder

The second example is more complex: A full adder represented as a truth table (Fig. 8a) is transformed into a representation using two half adders and an OR gate (Fig. 8b/c). Again, the transformation is step-by-step and each step is graphically illustrated in Fig. 9:

1. The transformation starts with `adder1` from Fig. 8a. 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`):

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

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

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

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

Fig. 10 Auxiliary functions used in full adder transformation

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

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 Fig. 10. 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
 in (z,x,y::Bit)
 out (s,c'::Bit)
match
   (0,0,0) -> (0,0) |
   (1,0,0) -> (1,0) |
   (0,0,1) -> (0,1) |
   (1,0,1) -> (1,1) ...;
```

```
box or
 in (x,y::Bit)
 out (c'::Bit)
match
   (0,0) -> 0 |
   (0,0) -> 0 |
   (0,1) -> 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 Fig. 8c.

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 Fig. 10: **ReplaceExpr**(`adder2.h1h2`, `1`, `gg` · `ff(a,b,c)`). Since the context does 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
 in (x,y,c::Bit)
 out (s,x',c'::Bit)
match
   (0,0,0) -> (0,0,0) |
   (0,0,1) -> (0,1,0) |
   (0,1,0) -> (1,0,0) |
   (0,1,1) -> (1,1,0) ...;
```

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

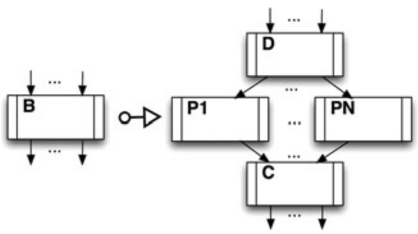
5. The transformation of `h1` follows the same pattern as `h2` (and `or`): First the case expression is removed, followed by a variable introduction and threading elimination: **Unfold**(`adder2.h1`, `ff`), **CaseE**(`adder2.h1`, `1`), **MatchVarI**(`adder2.h1`, `c`, `x'`) and **ThreadE**(`adder2.h2`,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

6 Strategies for transforming Hume for parallelism

Hume boxes offer considerable potential as loci of concurrent execution. However, unsystematic parallelisation is well known to produce disappointing initial results, subsequently requiring considerable fine tuning of code. We see the box calculus as offering a principled way to guide the construction of multi-box programs with known properties.

We next present an overview of the derivation of a general divide and conquer transformation for exposing horizontal parallelism from a single box. This is a well known variant of standard transformations for higher-order program manipulation and skeleton deployment.

Graphically, this transformation can be seen as



where box B (with no time dependency) is transformed into a divide (D) and conquer (C) component, where $P1$ to PN performs the computation. We will now illustrate how to achieve this in the calculus by applying a (binary) divide and conquer strategy, followed by two flattening steps, which combines the divide and conquer boxes, respectively. We will illustrate this by creating a system of four parallel boxes, but the same approach can be applied to create more parallel boxes.

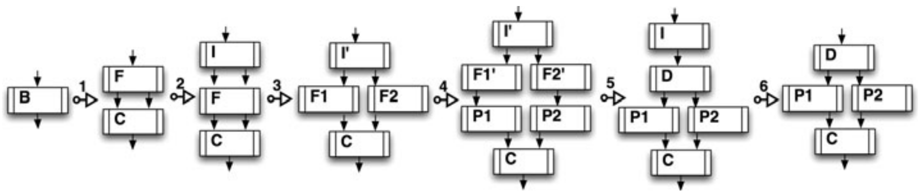
We assume there is a *divide* function, which divides an input type into two "sub-types", e.g. this can be a list or vector divided in the middle (where the former will have the same type). Moreover, we can normalise $x \to \mathit{divide}\ x$ to use the projection function l and r for each resulting partition $\mathit{divide}\ x = (l\ x, r\ x)$. Next, there is a dual conquer function *conq* which combines the result, and a function f such that the following distributivity law holds:

$$\mathbf{Dist\_Div\_Conq}(f) \equiv (f\ x = f(\mathit{conq}(l\ x)\ (r\ x)) = \mathit{conq}\ (l(f\ x))\ (r(f\ x)))$$

One example of such f is the mapping a function g over a list x. Further, we assume that there are no timing dependency on the boxes being transformed.

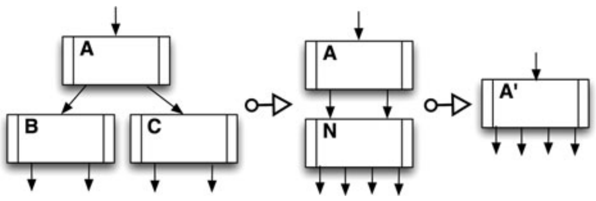
6.1 A binary divide & conquer strategy

The first strategy transforms a box B into a divide box B, two boxes $P1$ and $P2$ which performs the computation, and a conquer box C which merges the results. We have previously derived this strategy in [16] using the box calculus—however, for completeness, the full strategy is shown in Fig. 11. First, the expression of B is refactored following **Dist_Div_Conq**. This is followed by vertically composing out the *conquer* function into D, while the rest of the box is in F (1). To horizontally decompose $P1$ and $P2$ in a later step, an identity box is introduced with two duplicate wires (2). The F box can now be



$$\begin{array}{c}
\langle B, [iw], [ow], [x \to f\ x], [], [], [] \rangle = \mathbf{Get\_Box}(B, bcs) \\
\mathbf{Dist\_Div\_Conq}(f) \\
B' = \langle B, [iw], [ow], [x \to \mathit{conq}\ (l(f\ x))\ (r(f\ x))], [], [], [] \rangle \\
\langle \theta, \eta, bcs \rangle \vdash \mathbf{Replace}(B, B') \Downarrow \langle \theta, \eta, bcs'' \rangle \\
(1)\ \langle \theta, \eta, bcs'' \rangle \vdash \mathbf{VCompE}(B', F, [a, b], C, [a, b]) \Downarrow \langle \theta, \eta_1, bcs_1 \rangle \\
\langle \theta, \eta_1, bcs_1 \rangle \vdash \mathbf{IdI}(F, iw, I) \Downarrow \langle \theta, \eta_2, bcs_2 \rangle \\
(2)\ \langle \theta, \eta_2, bcs_2 \rangle \vdash \mathbf{DupI}(I, v', v'', F, ow, ow') \Downarrow \langle \theta, \eta_3, bcs_3 \rangle \\
(3)\ \langle \theta, \eta_3, bcs_3 \rangle \vdash \mathbf{HCompE}(F, [v'], [ow], F1, F2) \Downarrow \langle \theta, \eta_4, bcs_4 \rangle \\
\langle \theta, \eta_4, bcs_4 \rangle \vdash \mathbf{VCompE}(F1, F1', [o1], P1, [i1]) \Downarrow \langle \theta, \eta_5, bcs_5 \rangle \\
(4)\ \langle \theta, \eta_5, bcs_5 \rangle \vdash \mathbf{VCompE}(F2, F2', [o2], P2, [i2]) \Downarrow \langle \theta, \eta_6, bcs_6 \rangle \\
\langle \theta, \eta_6, bcs_6 \rangle \vdash \mathbf{HCompI}(F1', F2', D) \Downarrow \langle \theta, \eta_7, bcs_7 \rangle \\
(5) \langle \theta, \eta_7, bcs_7 \rangle \vdash \mathbf{DupE}(D, i1, i2) \Downarrow \langle \theta, \eta_8, bcs_8 \rangle \\
(6) \langle \theta, \eta_8, bcs_8 \rangle \vdash \mathbf{IdE}(I) \Downarrow \langle \theta, \eta', bcs' \rangle \\
\hline
\langle \theta, \eta, bcs \rangle \vdash \mathbf{DivConq}(B, D, P1, P2, C) \Downarrow \langle \theta, \eta', bcs' \rangle
\end{array}$$

Fig. 11 Derivation of the **DivConq** strategy



$$\frac{\begin{array}{c}\langle A, iws_A, [ow_{A1}, ow_{A2}], [x \to (l\ x, r\ x)], [], [], []\rangle = \textbf{Get\_Box}(A, bcs)\\ \langle B, [ow_{A1}], [ow_{B1}, ow_{B2}], [y \to (l\ y, r\ y)], [], [], []\rangle = \textbf{Get\_Box}(B, bcs)\\ \langle C, [ow_{A2}], [ow_{C1}, ow_{C2}], [z \to (l\ z, r\ z)], [], [], []\rangle = \textbf{Get\_Box}(C, bcs)\\ \langle \theta, \eta, bcs\rangle \vdash \textbf{HComplI}(B, C, N) \Downarrow \langle \theta, \eta'', bcs''\rangle\\ \langle \theta, \eta'', bcs'\rangle \vdash \textbf{MatchE}(N, 3) \Downarrow \langle \theta, \eta'', bcs'''\rangle\\ \langle \theta, \eta'', bcs'''\rangle \vdash \textbf{MatchE}(N, 2) \Downarrow \langle \theta, \eta'', bcs''''\rangle\\ \langle \theta, \eta'', bcs''''\rangle \vdash \textbf{VComplI}(A, N, A') \Downarrow \langle \theta, \eta', bcs'\rangle\end{array}}{\langle \theta, \eta, bcs\rangle \vdash \textbf{FlattenDiv}(A, B, C, A') \Downarrow \langle \theta, \eta', bcs'\rangle}$$

Fig. 12 Derivation of the **FlattenDiv** strategy

horizontally decomposed into $F1$ and $F2$ (3), and the divide (projection) functions are then vertically moved into $F1'$ and $F2'$, thus giving the $P1$ and $P2$ boxes (4). $F1'$ and $F2'$ are then horizontally combined into the divide box D (5), and the identity box I can be elided (6). Please see [16] for a more detailed discussion.

6.2 A flattening strategy for dividing

DivConq will create two parallel boxes. To achieve four, **DivConq** can be applied again on the resulting $P1$ and $P2$ boxes. This will result in a divide box, with both outputs wired into another divide box. Figure 12 derives the **FlattenDiv** strategy which flattens these three boxes into one. box. The correctness of this transformation follows from the transitivity of $\Rightarrow_T$. Firstly, the A, B and C boxes must apply the (normalised) *divide* function, and have the correct wiring (as shown in the diagram). B and C are first (horisontally) combined, as shown in the intermediate step. **HComplI** will introduce additional $(y, *) \to \cdots$ and $(*, z) \to \cdots$ matches. These can safely be eliminated, since either both or none of the inputs will be available (assured by A). In this intermediate step, N will have the following match:

$$(y, z) \to (l\ y, r\ y, l\ z, r\ z)$$

In the final step A and N are combined using the vertical composition rule—the new box A' will then have the following match:

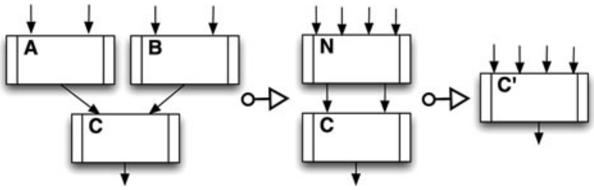
$$x \to (l\ (l\ x), r\ (l\ x), l\ (r\ x), r\ (r\ x))$$

6.3 A flattening strategy for conquering

As for dividing, two nested application of **DivConq** will create a conquer box with outputs connected to another two conquer boxes. Figure 13 derives the **FlattenConq** strategy—which is behaves for conquer as **FlattenDiv** behaves for divide. The intermediate N box will here have the following match:

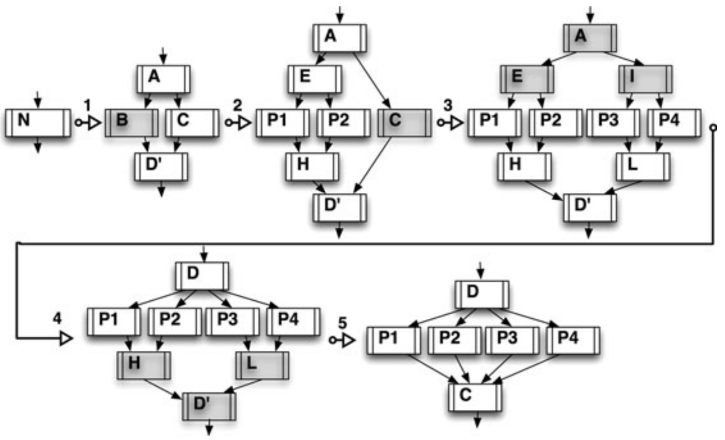
$$(a, b, c, d) \to (conq\ a\ b, conq\ c\ d)$$

while the final C' box has the match:



$$\dfrac{\begin{array}{c}\langle A, [iw_{A1}, iw_{A2}], [ow_A], [(a,b) \to conq\ a\ b], [], [].[]\rangle = \mathbf{Get\_Box}(A, bcs)\\ \langle B, [iw_{B1}, iw_{B2}], [ow_B], [(c,d) \to conq\ c\ d], [], [].[]\rangle = \mathbf{Get\_Box}(B, bcs)\\ \langle C, [ow_A, ow_B], [ow_C], [(e,f) \to conq\ e\ f], [], [].[]\rangle = \mathbf{Get\_Box}(C, bcs)\\ \langle \theta, \eta, bcs\rangle \vdash \mathbf{HCompI}(A, B, N) \Downarrow \langle \theta, \eta'', bcs''\rangle\\ \langle \theta, \eta'', bcs'\rangle \vdash \mathbf{MatchE}(N, 3) \Downarrow \langle \theta, \eta'', bcs'''\rangle\\ \langle \theta, \eta'', bcs'''\rangle \vdash \mathbf{MatchE}(N, 2) \Downarrow \langle \theta, \eta'', bcs''''\rangle\\ \langle \theta, \eta'', bcs''''\rangle \vdash \mathbf{VCompI}(N, C, C') \Downarrow \langle \theta, \eta', bcs'\rangle\end{array}}{\langle \theta, \eta, bcs\rangle \vdash \mathbf{FlattenConq}(A, B, C, C') \Downarrow \langle \theta, \eta', bcs'\rangle}$$

Fig. 13 Derivation of **FlattenConq** strategy



$$\dfrac{\begin{array}{c}(1)\ \langle \theta, \eta, bcs\rangle \vdash \mathbf{DivConq}(N, A, B, C, D') \Downarrow \langle \theta, \eta'', bcs''\rangle\\ (2)\ \langle \theta, \eta'', bcs''\rangle \vdash \mathbf{DivConq}(B, H, P1, P2, H) \Downarrow \langle \theta, \eta''', bcs'''\rangle\\ (3)\ \langle \theta, \eta''', bcs'''\rangle \vdash \mathbf{DivConq}(C, I, P3, P4, L) \Downarrow \langle \theta, \eta'''', bcs''''\rangle\\ (4)\ \langle \theta, \eta'''', bcs''''\rangle \vdash \mathbf{FlattenDiv}(A, E, I, D) \Downarrow \langle \theta, \eta''''', bcs'''''\rangle\\ (5)\ \langle \theta, \eta''''', bcs'''''\rangle \vdash \mathbf{FlattenConq}(H, L, D', C) \Downarrow \langle \theta, \eta', bcs'\rangle\end{array}}{\langle \theta, \eta, bcs\rangle \vdash \mathbf{Para4}(N, D, P1, P2, P3, P4, C) \Downarrow \langle \theta, \eta', bcs'\rangle}$$

Fig. 14 Derivation of **Para4** strategy

$$(a, b, c, d) \to conq(conq\ a\ b)\ (conq\ c\ d)$$

6.4 A divide & conquer based strategy for 4 parallel boxes

We can now combine the three derived strategies to create a strategy to paralellise four boxes. This **Para4** strategy is derived in Fig. 14. To ease the reading of the associated diagram, we have highlighted the box(es) which are transformed in the following step. As clear from the discussion above, the **DivConq** strategy is first applied to the original box N (1). The same strategy is then applied to the "left computation box" (2) and "right computation box" (3). A, E and I are all then divide boxes, and **FlattenDiv** is then applied, creating the box D (4). Finally, in (5) the **FlattenConq** strategy is applied to the H, L and D' (conquer) boxes, creating box C.

6.5 Applying the parallelising transformation

The Hume tool set is built around the Hume Abstract Machine (HAM) which provides a locus for resource analysis as well as implementation. We have been experimenting with a simple shared memory, MIMD implementation of the standard HAM interpreter [2] using OpenMP to implement each box in a separate thread for first stage execution on multiple cores. If all boxes have similar run times then the implementation offers good speedup but most programs will not offer such regularity of box behaviour. Thus, the slowest concurrent box will severely limit the overall effectiveness of this naive approach. However, by systematically applying the above transformations, which have known behavioural consequences, it may be possible to produce multi box systems where each concurrent box has uniform behaviour.

So far we have explored two canonical cases. First of all, we have investigated a top level map over a list. Thus, for:

```
map f [] = [];
map f (h:t) = f h:map f t;
```

it is well known that for `map f l`, if:

```
l == left++right
```

then:

```
map f l == map f (left++right) == (map f left)++(map f right)
```

Thus, we may split a `map` argument list into a concatenation of equal length subsections in a divide phase, send each to a separate box for concurrent execution, and then concatenate the results in the conquer phase.

We have also investigated the slightly more elaborate linear iteration:

```
apply i j f (h:t) =
 if i>j
 then []
 else f i h:apply (i+1) j f t;
```

For `apply 1 (length l) f l`, it is straightforward to establish that if:

```
l == left++right
```

then:

```
apply 1 (length l) f l ==
apply 1 (length left+length right) f (left++right) ==
(apply 1 (length left) f left)++
(apply (length left+1) (length l) f right)
```

As with `map`, we may split an `apply` argument list into a concatenation of equal length subsections with appropriate ranges in a divide phase, send each to a separate box for concurrent execution, and once again concatenate the results in the conquer phase.

Early results reported in [2], of a simple Fibonacci test, and in [16], of a list-based matrix multiplication, show consistent near-linear speedup with up to eight boxes on up to eight cores.

7 Related work

A Hume *transformation* is a strategy. Following Visser [33] 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 computation to coordination, i.e. a translation from the expression layer of a box into the coordination layer, between components of a nested box. This has also been illustrated by our examples, where relatively rich single-box truth-tables are dissolved into configurations of simpler boxes. A full transformation can therefore be seen as the form of program translation termed *program synthesis* from a computation 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 Bird-Meertens Formalism [5] and calculational programming [21], 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 resemblance to the form of program rephrasing called *program refactoring* [10].

Just as Hume integrates a finite state coordination language with a functional transition computation 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 the box calculus for Petri nets [9] as well as process calculi [4]. The computation aspects resemble classic functional programming techniques including curry/uncurry, fold/unfold [7] and functional refactoring [25]. Hence, a full transformation can be seen as a program translation, consisting of several program rephrasing steps. There is also some correspondence to Morgan's refinement rules [30]—although in sequential rather than (controlled) concurrent setting.

In principle the transformation proofs could have been achieved using (observational) bisimulations in a process algebra like CSP [20]. However, it is not possible to construct 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 combination [31]. 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 [29]. Different strategies for general formal verification of Hume programs are first discussed in [11]. TLA is first used to verify programs in [19], while Hierarchical Hume and linear recursion to box iteration with respect to scheduling are discussed in [15].

The calculus has yet not been fully formalised or mechanised. However, our experience with embedding (Hierarchical) Hume in TLA and Isabelle, and in ad-hoc mechanical transformation verification of both flat and Hierarchical Hume [12, 13], gives us confidence that full formalisation and mechanisation are achievable.

8 Conclusions and future work

We have presented a first approach towards a box calculus for Hume and Hierarchical Hume programs, which introduces correct transformation by construction, semi-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, as well as more generic parallelisation strategy.

Hierarchical Hume enables us to elide the potentially global impact of what should be localised program changes by providing a framework for the identification and isolation of distinct sub-systems within a program. Furthermore, the calculus supports the systematic transformation of program components through the introduction, modification, elimination, composition and separation of boxes and wires. A major strength of the calculus is that it combines changes to computation aspects within boxes with those to coordination aspects between boxes.

Our work 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 formalisation of the calculus will mainly require an extension 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 speculate 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.

In developing the box calculus, our next step is to identify a sufficient set of rules which is adequate for the classes of transformations between and within levels. We then seek to elaborate new transformation strategies that may be used in particular to optimise pragmatic properties of Hume programs. For example, we have shown that it is possible to apply the box calculus to derive a generalised transformation for parallelising Hume boxes. However, its application is clearly restricted to regular cases where splitting a computation can guarantee balance across all processors. Far more sophisticated strategies are required to establish or restore balance in irregular cases.

We have previously formalised TLA in the Isabelle theorem prover, and built both a shallow embedding of Hume and Hierarchical Hume on top of this [12]. The calculus, in turn, could be built on the shallow embedding, where each rule becomes a tactic, or a higher-level *proof plan*. The disadvantage of this is the complicated witnesses used for rules which changes the timing behaviour. These will have to be found and verified for each example. A better solution would be to use a deep embedding, where each rule will only need to be derived once. This approach is discussed below.

We envisage two further desirable developments of the box calculus. First of all, the examples considered above, of application of low-level rules, and of derivation of high-level strategies, require deep manipulation of coordination and expression layers. This suggests that scalable program manipulation with the box calculus would benefit strongly from fully automatic transformation or considerable automated support for user directed transformation.

A fundamental requirement for automatic transformation is to characterise some goal to which transformation is directed. One way of achieving this is to introduce a measure through some well-founded relation. For example, this could be that computation always moves from the expression layer into the coordination layer, to reduce complexity within a box and enhance the possibility of accurate intra-box costing. A second approach is to derive high-level generic strategies. This was illustrated in Sect. 6 where a strategy for parallel box execution was introduced. We think that, in the first instance, it would be acceptable to derive such high-level generic strategies manually. Strategy application could then again either be automatic, or by the user selecting strategies manually—the latter would depend on a good graphical user interface.

Independent of the two approaches, the automatic application of rules or strategies, reduces to checking the preconditions of rules. The strict topology and execution model of

Hume programs provide strong guidance in structuring proofs, and we therefore believe we can implement special purpose LCF-style tactics, or possibly *proof plans* (as in e.g. [8, 26]), to prove preconditions. To illustrate, [12] implements a special purpose (post facto) transformation tactic for Hierarchical Hume in Isabelle/HOL.

Moreover, we expect many precondition, such as comparing termination conditions and generalising rule set, to have algorithmic solution. We may also be able to rely on techniques such as data-flow analysis, to establish algorithmically (some cases of) properties like **NoTime_Dependency**. Additionally properties such as checking *Blocked* status could by achieved by model checking: for example, we discuss model checking of Hume program using TLA in [19]. Moreover, recent advances with SMT solvers should improve this further, due to the finiteness of the Hume coordination layer. Thus, we believe we can achieve a high degree of automation for this part of the process.

The second development is to directly augment the box calculus with the Hume cost models to implement the notion of *costing by construction*. In costing by construction, box calculus rules are augmented with cost judgements, so applying a rule to a program construct of known cost produces a changed program with a known changed cost. Thus, in principle, a program of required cost might be constructed from scratch by applying successive rules from a single, empty, unwired box of base cost.

In turn, costing by constructing could enable *cost-oriented program refinement*, similar to Aldinucci et al.'s [3] exploration of cost-directed parallelisation. The objective here would be to modify an original program in a principled manner to minimise cost or, less stringently, to meet cost requirements. Once again, this would benefit strongly from tool support. One potential difficulty with automatic cost oriented refinement is that it may be necessary to go through intermediate stages where costs temporarily increase. This is directly analogous to the need in correctness-oriented transformation, alluded to above, to go through stages where program correctness is temporarily compromised. This suggests that computer aid might be a worthwhile first goal towards full automation of box calculus based transformation for both correctness and cost.

Acknowledgements This paper is an elaboration of work first presented in [2, 14, 16].

This work has been supported by the EU FP6 EmBounded project (2005-8) and the UK EPSRC Islay project (EP/F030592, EP/F030657, EP/F03072, and EP/F031017) "Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds" (2008-11). Gudmund Grov was supported by a Heriot-Watt University James Watt Scholarship. He has also been employed by UK EPSRC Platform grants EP/E005713/1 and EP/E035329/1 (Newcastle University) during this work. He is currently employed by UK EPSRC grant EP/H024204/1 "AI4FM: the use of AI to automate proof search in Formal Methods" (2010-2014).

We would like to thank Robert Pointon and Andrew Ireland for valuable discussions of transformation and Hierarchical Hume. We would also like to thank Kevin Hammond, who built the original HAM interpreter and conducted the first multi-core HAM experiments, and Abyd Al Zain who built the current multi-core HAM interpreter.

Finally, we would like to than the anonymous referees for their constructive suggestions for improvements.

Appendix A: Summary of functions

Alloc$([S_1, \dots, S_n], H) :: [\mathbb{N}] \times H \to [Locs] \times H$

Accepts as input a list of natural numbers and a heap. Each number represent the size of a data type. Note that for mutable data structures, such as lists, this will be the size in the initial program configuration. For each such S_i $(i \in (1..n))$, a heap region of size S_i is allocated, and the list of newly allocated locations $[l_1, \dots, l_n]$ is returned, together with the new heap.

dom$(h) :: H \rightarrow \mathbb{P}(Locs)$
Returns the domain of heap h, i.e. **dom**$(h) = \{l \mid h(l) \neq nil\}$. $\mathbb{P}$ is the powerset.

Gen_Rules$(rs) :: [RS] \rightarrow [RS]$
Returns a *generalisation* of the rule set *rs*. This generalisation works by replacing pattern variables by '_', while the rest is unchanged. In an expression, everything but '$*$' is replaced by '_', and all function calls are removed. Furthermore, any duplicate matches as a result of this are omitted.

Get_Box$(B, bcs) :: Id \times [BCS] \rightarrow BCS$
Returns box configuration with box id B from list *bcs*.

HeapLocs_Copy$([l_1, \dots, l_n], H_1, H_2) :: [Locs] \times H \times H \rightarrow H \times [Locs]$
A deep copy of the locations $[l_1, \dots, l_n]$ of heap H_1 into heap H_2. A pair of the updated H_2 together with the locations of the copied elements is returned.

is_Blocked$(C, B) :: PC \times Id \rightarrow \mathbb{B}$
Holds if box B cannot be executed.

is_Empty$(H, L) :: H \times [Loc] \rightarrow \mathbb{B}$
Holds if $\forall l \in L . H(l) = nil$.

len$(L) :: [\alpha] \rightarrow \mathbb{N}$
Returns the length of list L. α indicate type variable, thus the function is polymorphic.

Match_eq$(rs_1, rs_2) :: [RS] \times [RS] \rightarrow \mathbb{B}$
Holds if the patterns of rs_1 and rs_1 match and consume the same inputs.

mutually_exclusive$(rs) :: [RS] \rightarrow \mathbb{B}$
Holds if the patterns of rule set *rs* are mutually exclusive.

NoTime_Dependency$(B, bcs) :: Id \times [BCS] \rightarrow \mathbb{B}$
There is no "time dependency" from box B. This implies that we can add identity boxes on the inputs and outputs of B, without this having any effect on the functional behaviour of the Hume program. We use a restrictive form of this predicate where we assume that **NoTime_Dependency**(B, bcs) holds iff:

- there are no *s in any patterns of boxes in *bcs* which the output of B is directly or indirectly wired to, and
- B is not directly connected to an output in heap θ.

Out_Type_Size$(e) :: Expr \rightarrow \mathbb{N}$
Returns the return type of an expression/function.

Patt_of_Expr$(e) :: Expr \rightarrow Patt$
Returns a pattern from an expression. The pattern will consist of (distinct) variable(s) to be bound.

project$([(p_1 \rightarrow e_1), \dots, (p_n \rightarrow e_n)], [(p'_1 \rightarrow e'_1), \dots, (p'_m \rightarrow e'_m)]) :: [RS] \times [RS] \rightarrow [RS]$
Pairwise combines each pattern p_i and p'_j with e_i and e'_j where $i \in 1..n$ and $j \in 1..m$. For example,
project$([p_1 \rightarrow e_1], [p_2 \rightarrow e_2, p_3 \rightarrow e_3]) = [(p_1, p_2) \rightarrow (e_1, e_2), (p_1, p_3) \rightarrow (e_1, e_3)]$

Terminates_eq$(B_1, B_2, bcs) :: Id \times Id \times [BCS] \rightarrow \mathbb{B}$
Holds if box B_1 and B_2 have the same termination conditions.

$L_1 @ L_2 :: [\alpha] \times [\alpha] \rightarrow [\alpha]$
Concat list L_1 in front of list L_2.

$\Box_H F :: (PC \rightarrow \mathbb{B}) \rightarrow PC \rightarrow \mathbb{B}$
This represents the temporal 'always' operator, denoting that $F :: PC \rightarrow \mathbb{B}$ must hold henceforth, abstracted over the program specification. Due to the temporal property, a possible world semantics is used, thus F is a predicate over a program configuration. Translated into TLA, the meaning becomes:

$$(\lambda \langle hw,hb,bcs \rangle .\ \Box_H F) \Leftrightarrow (\lambda \langle hw,hb,bcs \rangle .\ Init_{hw} \wedge Init_{hb} \wedge \Box(run_{bcs}) \Rightarrow \Box F)$$

Appendix B: Summary of rules and strategies

CaseE$(B, i) :: Id \times \mathbb{N} \rightarrow PC \rightarrow PC$
Moves case expression in match i of box B into B's rule set (note that we do not need bcs since this is an implicit argument).

CaseI$(B, i, j) :: Id \times \mathbb{N} \times \mathbb{N} \rightarrow PC \rightarrow PC$
Replaces match i to j in box B by a case-expression.

DupI$(A, x, x', B, y, y') :: Id \times Id \times Id \times Id \times Id \times Id \rightarrow PC \rightarrow PC$
Duplicates wire connecting x of box A and y of B, with wire names x' (of A) and y' (of B) respectively.

HCompE$(B, [i_1, \dots, i_n], [o_1, \dots, o_m], X, Y) :: Id \times [Id] \times [Id] \times Id \times Id \rightarrow PC \rightarrow PC$
Horizontally de-composes box B into boxes X and Y, where X has inputs $[i_1, \dots, i_n]$ and outputs $[o_1, \dots, o_m]$. Y will have the inputs/output of B not in $[i_1, \dots, i_n]/[o_1, \dots, o_m]$.

HCompI$(A, B, N) :: Id \times Id \times Id \rightarrow PC \rightarrow PC$
Horizontally composes box A and box B into N. There is an implicit renaming of variables in the patterns of B if there are name clashes with variables of A.

HeapLocs_WireFree$([l_1, \dots, l_n]) :: [Locs] \rightarrow PC \rightarrow PC$
Frees locations $[l_1, \cdots l_n]$ from the wire heap.

HeapLocs_BoxFree$([l_1, \dots, l_n]) :: [Locs] \rightarrow PC \rightarrow PC$
Frees locations $[l_1, \dots, l_n]$ from the box heap.

HieE$(B, N) :: Id \times Id \rightarrow PC \rightarrow PC$
Replaces B with it's (only) child box and names this new box N.

HieI$(B, N) :: Id \times Id \rightarrow PC \rightarrow PC$
Replaces box B by N which only holds B.

IdE$(B) :: Id \rightarrow PC \rightarrow PC$
Eliminates identity box B.

IdI$(B, v, N) :: Id \times Id \times Id \rightarrow PC \rightarrow PC$
Introduces an identity box N to wire connected to v of box B.

MatchE$(B, n) :: Id \rightarrow \mathbb{N} \rightarrow PC$
Eliminates match n of box B.

MatchVarI$(B, i, o) :: Id \times Id \times Id \rightarrow PC \rightarrow PC$
Replaces constants in inputs i and output o by a variable.

Rename$(A, N) :: Id \times Id \rightarrow PC \rightarrow PC$
Renames box A to N.

Replace$\big([A_1, \dots, A_n], [B_1, \dots, B_m]\big) :: [Id] \times [Id] \rightarrow PC \rightarrow PC$
Replaces boxes $A_1, \dots, A_n$ by $B_1, \dots, B_m$.

ReplaceBoxHeap$(H) :: H \rightarrow PC \rightarrow PC$
Replaces the box heap by H.

ReplaceExpr$(A, n, e) :: Id \times \mathbb{N} \times Expr \rightarrow PC \rightarrow PC$
The expression of match n of box A is replaced by e.

ReplaceWireHeap$(H) :: H \rightarrow PC \rightarrow PC$
Replaces the wire heap by H.

ThreadE$(B, x, y) :: Id \times Id \times Id \rightarrow PC \rightarrow PC$
Removes threading of input x and output y through box B.

Unfold$(B, n, f) :: Id \times Id \times Id \rightarrow PC \rightarrow PC$
Unfolds function f in match n of box B.

VCompE$(B, N, [o_1, \dots, o_n], M, [i_1, \dots, i_n]) :: Id \times Id \times [Id] \times Id \times [Id] \rightarrow PC \rightarrow PC$

Vertically de-composes box B into two sequentially composed boxes N and M, where N has B's inputs and $[o_1, \ldots, o_n]$ as outputs, and M has $[i_1, \ldots, i_n]$ as inputs and B's outputs.

VCompl$(F, S, B) :: Id \times Id \times Id \to PC \to PC$

Vertically composes boxes F and S into box B. It assumes that F is directly followed by S. The input of the new box B is the input of F and the output is the output of S.

VRename$(A, x, N) :: Id \times Id \times Id \to PC \to PC$

Renames wire x of box A to N.

WireE$(A, x) :: Id \times Id \to PC \to PC$

Eliminates wire x of box.

WiresE$(A, xs) :: Id \times Id \to PC \to PC$

Eliminates wires xs of box A.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991)
2. Al Zain, A., Michaelson, G., Hammond, K.: Multi-core parallelisation for Hume. In: Horvath, Z., Zsok, V., Achten, P., Koopman, P. (eds.) Tenth Symposium on Trends in Functional Programming, Komarno, Slovakia, 2–4 June 2009, pp. 131–142 (2009)
3. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework. Int. J. Parallel Emerg. Distrib. Syst. **16**(2), 87–121 (2001)
4. Baeten, J.C.M.: A brief history of process algebra. Theor. Comput. Sci. **335**(2–3), 131–146 (2005)
5. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall, New York (1997)
6. Breitinger, S., Loogen, R., Ortega Mallén, Y., Peña Marí, R.: Eden—the paradise of functional concurrent programming. In: EuroPar'96—European Conference on Parallel Processing, Lyon, France, August. LNCS, vol. 1123, pp. 710–713. Springer, Berlin (1996)
7. Burstall, R., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
8. Cook, A., Ireland, A., Michaelson, G.J., Scaife, N.: Discovering applications of higher order functions through proof planning. J. Form. Asp. Comput. **17**(1), 38–57 (2005)
9. Devillers, R., Klaudel, H., Riemann, R.-C.: General parameterised refinement and recursion for the M-net calculus. Theor. Comput. Sci. **300**(1–3), 259–300 (2003)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
11. Grov, G.: Verifying the correctness of Hume programs—an approach combining algorithmic and deductive reasoning. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE-05), pp. 444–447. ACM Press, New York (2005)
12. Grov, G.: Reasoning About Corectness Properties of a Coordination Language. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University (2009)
13. Grov, G., Ireland, A.: Towards automated property discovery within Hume. In: Ireland, A., Kovacs, L. (eds.) 2nd International Workshop on Invariant Generation (WING'09), pp. 45–59 (2009)
14. Grov, G., Michaelson, G.: Towards a box calculus for hierarchical Hume. In: Morazon, M. (ed.) Trends in Functional Programming, vol. 8, pp. 71–88 (2008)
15. Grov, G., Pointon, R., Michaelson, G., Ireland, A.: Preserving coordination properties when transforming concurrent system components. In: Coordination Models, Languages and Applications Track of the 23rd Annual ACM Symposium on Applied Computing, 1515 Broadway, New York, March 2008, vol. 1, pp. 126–127. The Association for Computing Machinery, Inc., New York (2008)
16. Grov, G., Michaelson, G., Al Zain, A.: Multi-core parallelisation of Hume through structured transformation. In: Draft Proceedings of 21st International Symposium on Implementation and Application of Functional Languages, Seton-Hall University, New Jersey, September (2009)
17. Hammond, K., Michaelson, G.J.: 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, pp. 37–56. Springer, Berlin (2003)
18. Hammond, K., Ferdinand, C., Heckmann, R., Dyckhoff, R., Hoffmann, M., Jost, S., Loidl, H.-W., Michaelson, G., Pointon, R., Scaife, N., Sérot, J., Wallace, A.: Towards formally verifiable resource bounds for real-time embedded systems. In: Proc. Workshop on Innovative Techniques for Certification of Embedded Systems (2006)

19. Hammond, K., Grov, G., Michaelson, G., Ireland, A.: 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. LNCS, vol. 4449, pp. 91–107. Springer, Berlin (2007)
20. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs (1985)
21. Hutton, G., Wright, J.: Calculating an exceptional machine. In: Loidl, H.-W. (ed.) Trends in Functional Programming, vol. 5, pp. 49–64 (2006)
22. Jost, S.: Formal Hume semantics. EmBounded Project Deliverable (2008). Deliverable D12. Available at http://www.embounded.org/
23. Kesseler, M.H.G.: Constructing skeletons in clean: the bare bones. In: HPFC'95—Conference on High Performance Functional Computing, Denver, CO, April 10–12, pp. 182–192 (1995)
24. Lamport, L.: The temporal logic of actions. ACM Toplas **16**(3), 872–923 (1994)
25. Li, H., Thompson, S.: A comparative study of refactoring Haskell and Erlang programs. In: Proceedings of 6th IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, USA, September, pp. 197–206 (2006)
26. Madden, P.: Automated Program Transformation Through Proof Transformation. PhD thesis, University of Edinburgh (1991)
27. Manna, Z.: Mathematical Theory of Computing. McGraw-Hill, New York (1974)
28. Michaelson, G., Scaife, N., Bristow, P., King, P.: Nested algorithmic skeletons from higher order functions. Parallel Algorithms Appl. **16**, 181–206 (2001). Special Issue on High Level Models and Languages for Parallel Processing
29. Michaelson, G., Hammond, K., Sérot, J.: The finite state-ness of FSM-Hume. In: Trends in Functional Programming, vol. 4, pp. 19–28. Intellect, Bristol (2004)
30. Morgan, C.: Programming from Specifications. Prentice-Hall, New York (1990)
31. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Form. Asp. Comput. **17**(4) (2005)
32. Trinder, P.W., Hammond, K., Loidl, H.-W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. J. Funct. Program. **8**(1), 23–60 (1998)
33. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symb. Comput. **40**(1), 831–873 (2005). Special issue on Reduction Strategies in Rewriting and Programming