A Framework for the Box Calculus

Konstantin Devyatov, Gudmund Grov, Greg Michaelson

1 Introduction

The Hume box calculus was first introduced in TFP 2007 [3] as part of Grov's PhD thesis [2]. The calculus is used to transform programs written in the Hume language [5], which is a finite state automata language build on top of a purely functional language (which can be seen as a set of boxes connected by wires). The originally motivation for the calculus was in order to statically guarantee time and space properties, as shown in [4], and it has later been extended to a method for multi-core programming [7].

The box calculus has not been implemented yet, and so far each proofs has been by hand. At the most abstract level, each rule of the calculus can be seen as a graph rewrite rule, and encoding the calculus in a graph rewrite system therefore seems like a natural approach.

Here, we describe a first step towards a box calculus tool by encoding Hume programs in the Quantomatic graph rewrite system [6]. This task has included 3 steps:

- Develop a Quantomatic theory where a graph represents a Hume program.
- Based on an existing Hume parser, develop a translator from Hume source code into a graph of this theory.
- Dually, generate a code generator, that generates Hume source code from such a graph.

A user will then write a Hume program. Import this into our box calculus tool, then rewrite the program using the calculus, and generate the new updated code.

2 Hume & the Box Calculus

The project's aim is to encode the Hume programming language and the Box Calculus as a theory of the Quantomatic graph rewrite system.

2.1 Hume

Hume (Higher-order Unified Meta-Environment) is a strongly typed, mostly-functional language with an integrated tool set for developing, proving and assessing concurrent, safety-critical systems. Hume aims to extend the frontiers of language design for resource-limited systems, including real-time embedded and safety-critical systems, by introducing new levels of abstraction and provability.

To illustate consider the following Hume program:

```
stream input from "std_in"; --declare stream "input" from standard input
stream output to "std_out"; --declare stream "output" to standard output

add1 x = x + 1; --declare function "add1", that increments an argument integer
mult2 x = x * 2; --declare function "mult2", that multiplies an argument integer by 2

box composed --declaration of box "composed"
 in (n::int 64) --list of inputs, "n" of type "int 64"
 out (n'::int 64) --list of outputs, "n'" of type "int 64"
```

```
 match --matching declaration
  (n) -> (mult2(add1(n))); --take "n", add 1 to it and multiply it by 2

wire composed --wire declaration, conneting inputs and outputs to other boxes and streams
 (input) --connect first input ("n") to stream "input"
 (output); --connect first output ("n'") to stream "output"
```

It takes in an integer `n` from standard input, then pushes the result `(n+1)*2` to the standard output, using the functions `add1` and `mult2`.

2.2 Hume Box Calculus

Next consider the following Hume programs:

```
stream input from "std_in"; --declare stream "input" from standard input
stream output to "std_out"; --declare stream "output" to standard output

add1 x = x + 1; --declare function "add1", that increments an argument integer
mult2 x = x * 2; --declare function "mult2", that multiplies an argument integer by 2

box addOne --declaration of box "addOne"
 in (n::int 64) --list of inputs, "n" of type "int 64"
 out (n'::int 64) --list of outputs, "n'" of type "int 64"
 match
  (n) -> (add1(n)); --take "n", add 1 to it

box multTwo -- declaration of box "multTwo"
 in (n::int 64) --list of inputs, "n" of type "int 64"
 out (n'::int 64) --list of outputs, "n'" of type "int 64"
 match
  (n) -> (mult2(n)); --take "n", multiply it by 2

wire addOne --wire declaration, conneting inputs and outputs to other boxes and streams
 (input) --connect first input ("n") to stream "input"
 (multTwo.n); --connect first output ("n'") to variable "n" of box "multTwo"

wire multTwo
 (addOne.n') --connect "n" to variable "n'" of box "addOne"
 (output); --connect "n'" to stream "output"
```

Notice that here the `composed` box from the previous program has been split into two sequential boxes: `addOne` followed by `multTwo`. What has happened is that the composition of the `add1` and `mult1` functions from the `composed` box has been lifted to the box level. This has been achieved by a rule called 'vertical de-composition' [3].

2.3 Quantomatic

Quantomatic implements the theory of 'string graphs' [1]. It follows the double-pushout approach to graph rewriting, and has the novelty that "dangling edges" are supported. For more details we refer to [1].

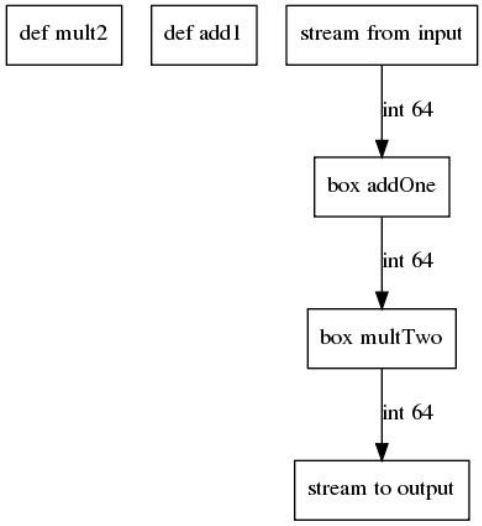

Figure 1: decomposition program graph

3 Representation

Any Hume program could be seen as a set of boxes and wires, that connect them. For example, our second program can be seen in Figure 1.

The Hume parser will give us an Abstract Syntax Tree (AST). As the Hume parser is written in Haskell, and Quantomatic in PolyML, the first thing we did was to create an ML file from Haskell, which we then could import in PolyML. The AST of the second program looks as follows:

```
[
 AST.FromStream (AST.Var "input",AST.SString "std_in"),
 AST.ToStream (AST.Var "output",AST.SString "std_out"),

 AST.Def
 ( AST.Var "add1", (* function name *)
  [AST.Var "x"], (* argument list *)
   AST.Apply (AST.Binop "+",[AST.Var "x",AST.IInt 1]) (* expression *)
 ),
 AST.Def
 ( AST.Var "mult2",
  [AST.Var "x"],
   AST.Apply (AST.Binop "*",[AST.Var "x",AST.IInt 2])
 ),

 AST.Box
 ( AST.Var "addOne", (* name of the box *)
  [AST.LVar (AST.TInt (AST.TSInt 64),"n")], (* list of typed input variables *)
  [AST.LVar (AST.TInt (AST.TSInt 64),"n'")], (* list of typed output variables *)
   AST.Match [([AST.Var "n"], [AST.Apply (AST.Var "add1",[AST.Var "n"])])]
     (* list of matches *)
```

```
 ),

 AST.Box
 ( AST.Var "multTwo",
  [AST.LVar (AST.TInt (AST.TSInt 64),"n")],
  [AST.LVar (AST.TInt (AST.TSInt 64),"n'")],
   AST.Match [([AST.Var "n"],[AST.Apply (AST.Var "mult2",[AST.Var "n"])])]
 ),

 AST.Wire
 ( AST.Var "addOne", (* name of the box to be wired *)
  [AST.Connect (AST.Var "input")], (* list of input sources *)
  [AST.Link (AST.Var "multTwo",AST.Var "n")] (*list of output destinations *)
 ),
 AST.Wire
 ( AST.Var "multTwo",
  [AST.Link (AST.Var "addOne",AST.Var "n'")],
  [AST.Connect (AST.Var "output")]
 )
];
```

To develop a Quantomatic theory one first need to define the data that goes on the vertices and edges. Then the above AST needs to be translated into such a graph (and translated back to generate code). Below we detail the representations of each entity of a graph:

- A box is represented as a vertex of rectangular shape with label corresponding to "box " + its name (see fig. 2). Data stored inside the vertex (not shown in a dot graph) is box's AST in full. So for the "addOne" box the data stored would be:

```
AST.Box
( AST.Var "addOne", (* name of the box *)
 [AST.LVar (AST.TInt (AST.TSInt 64),"n")], (* list of typed input variables *)
 [AST.LVar (AST.TInt (AST.TSInt 64),"n'")], (* list of typed output variables *)
  AST.Match [([AST.Var "n"], [AST.Apply (AST.Var "add1",[AST.Var "n"])])]
    (* list of matches *)
)
```

- Streams are also represented as vertices with labels showing names of stream variables (fig. 3). Data stored inside the vertex is stream's AST in full. Example of "stream to input":

```
AST.FromStream
( AST.Var "input", (* name of stream variable *)
  AST.SString "std_in" (* actual stream source *)
)
```

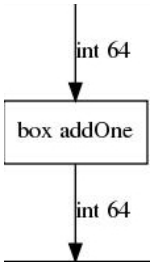

Figure 2: addOne box

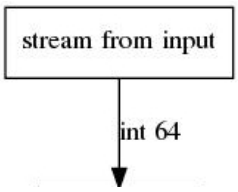

Figure 3: stream from input

- To be able to use function definitions, they also have to be stored a graph. Here they are stored as vertices, labelled with their variable names. These vertices have no edges coming in or out of them (see fig. 4).

```
AST.Def
( AST.Var "mult2", (* function name *)
 [AST.Var "x"], (* list of arguments *)
  AST.Apply (* expression *)
  ( AST.Binop "*", (* expression operator *)
   [AST.Var "x",AST.IInt 2] (* list of expression arguments *)
  )
)
```

- A wire is represented as an edge between two vertices, with label corresponding to its type (see fig. 2). Note that one wire declaration usually connected several variables, hence it would be represented as several edges. The reason, why edges have to hold data, is to be able to go back from graph to Hume code without a loss of data. Data held by edges is of the following type:

```
{source:string, destination:string, type:string, value:int option}
  (* a record format supports elaboration *)
```

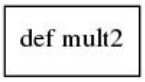

Figure 4: function definitions mult2

The first three parameters are simple enough, being string, but the last one deserves a brief explanation. To be able to differentiate between initiated wires (keyword ”initially”) and wires, waiting for data to be passed through, option type has been added to integer in the value field.

4 Parsing & Printing

Here we describe the details of the parser of Hume programs and printing of graphs into Hume programs.

4.1 Assumptions

1. We only work with correct Hume programs.
2. Only input values could have an initial value (use of keyword ”initially”).
3. This version of the theory supports only integer (int 64) data to be passed through boxes.

4.2 General Graph Properties:

1. Three types of vertices:
   (a) box
   (b) stream
   (c) definition
2. Three types of connections (definitions have no connecting edges):
   (a) stream-to-box
   (b) box-to-box
   (c) box-to-stream

4.3 Hume-to-Graph: CRT - graph

Algorithmic:

1. Takes in a concrete syntax tree of a Hume program.
2. Sort CRT: in order to draw a full graph, the CRT has to be in the following order: definitions-¿boxes/streams-¿edges. This is because boxes and streams are used by edges, and definitions are used by boxes (although not in the graph).
3. Construct a box list: while sorting CRT, all boxes should also be stored separately with smaller set of data: (name, inputs, outputs). This will be necessary in order to draw wires.
4. Draws boxes: streams and boxes hold their concrete syntax tree representation.
5. Draws wires: the algorithm for this is quite complex, because wire declarations hold limited information: name of a box,input and output connections. However, the connections are given in a neat format. Lets go back to the decomposition example, in particular ”multTwo” box and its wire declarations:

```
AST.Box
( AST.Var "multTwo",
 [AST.LVar (AST.TInt (AST.TSInt 64),"n")],
 [AST.LVar (AST.TInt (AST.TSInt 64),"n'")],
  AST.Match [([AST.Var "n"],[AST.Apply (AST.Var "mult2",[AST.Var "n"])])]
),
AST.Wire
( AST.Var "multTwo",
 [AST.Link (AST.Var "addOne",AST.Var "n'")],
    (* corresponds to edge "addOne.n'" -> "multTwo.n" *)
 [AST.Connect (AST.Var "output")]
    (* corresponds to edge "multTwo.n'" -> "output" *)
)
```

The declaration's input and output connections (in lists) give us exact source and destination addresses, respectively. But to figure out the other end of the connection (the "multTwo" box end), we have to compare order of variables in the box declaration to the order in the wire declaration. Below is a detail algorithms of drawing wires:

(a) Find inputs wires: parses through the list of inputs and converts them all to edges.

(b) Find other ends of wires: First the current box is found, from box list constructed before, while sorting CRT. Then the top input is pushed off the list. This ensures that duplication of edges will not be allowed.

(c) Construct edge: from the gathered data (source,destination, type and value), an edge record could be constructed.

(d) Draw input wires: Edges that represent input wires are then drawn on a graph one by one. As could be seen from General Graph Properties (2), all connections, but type (c), have box as destination. This means that the list of inputs contains most of edges.

(e) Draws wires outputs: a simplified process is gone through for output wires - only wires that are going to a stream are drawn (this follows from the General Graph Properties (2)). Below is the example of the edge data constructed, both input and output, from the "multTwo" box:

```
[
 {source = "addOne.n'", destination = "multTwo.n",
    type = "int 64", value = NONE },
 {source = "multTwo.n'", destination = "output",
    type = "int 64", value = NONE }
]
```

4.4 Graph-to-Hume: graph - CRT

Algorithmic:

1. Go through each box: make a list of box names and take one at a time (current box).

2. Sieving: go through vertices to find edges corresponding to their inputs and outputs, one edge (a-¿b) is an input edge for a and an output one - for b.

3. Find input edges: go through each edge and find all those that have destination parameter set to the current box; return an input list of edges.
4. Find output edges: the same as previous, but look in the source parameter; return an output list of edges.
5. Formatting: strip only requires information from the edges. Input edges need source, type and value; output edges need destination, type and value.
6. Wrapping: put the data in CRT order, sprinkled with required keywords.
7. Finishing: combine the vertex and edge CRT lists; return.

5 Conclusions and Future Work

Here we have described a theory for the Box Calculus in Quantomatic and shown how to translate between such graphs and the Hume source code. Therefore, the following future work has been identified:

1. Develop JSON interface in order to utilise Scala front end of the Quantomatic graph rewrite system.
2. Refactor and simplify the current project's source code in order for it to be used a tool.
3. Develop a set of rules and tactics for Hume Box Calculus, supported by the this project.

Addressing these will be the topic of the first author's honours project.

Acknowledgements

This work has been funding the EPSRC platform grant 'The Integration and Interaction of Multiple Mathematical Reasoning Processes' (EP/J001058). In this project the following software was used:

- Hume program to ML Abstract Syntax Tree parser by Greg Michaelson
- Quantomatic graph rewrite system [6].

References

[1] Lucas Dixon and Aleks Kissinger. Open graphs and monoidal theories. *CoRR*, abs/1011.4114, 2010.

[2] G. Grov. *Reasoning about correctness properties of a coordination language*. PhD thesis, Heriot-Watt University, 2009.

[3] G. Grov and G. Michaelson. Towards a Box Calculus for Hierarchical Hume. In M. Morazan, editor, *Trends in Functional Programming Volume 8*, pages 71–88. Intellect, 2008.

[4] G. Grov and G. Michaelson. Hume box calculus: robust system development through software transformation. *Higher Order Symbolic Computing*, July 2011. DOI 10.1007/s10990-011-9067-y.

[5] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *In Proc. of GPCE'03*, pages 37–56. LNCS 2830, Sep. 2003.

[6] A. Kissinger, A. Merry, L. Dixon, R. Duncan, M. Soloviev, and B. Frot. Quantomatic. https://sites.google.com/site/quantomatic/, 2011.

[7] G. Michaelson and G. Grov. Reasoning about multi-process systems with the box calculus. In *Proceedings of 4th Central European Summer Functional Programming School on (CEFP 2011)*. Springer, 2012.