Chapter 1

Translating Hume to Java

Chunxu Liu[1], and Greg Michaelson[2]

Abstract Progress towards the compilation of Hume to Java is presented.

1.1 INTRODUCTION

Hume[MHJ04] is a formally motivated language intended for application domains with strong bounds on resource consumption and response. A Hume program consists of concurrent *boxes* connected by *wires* between their *inputs* and *outputs*. Boxes, which never terminate, repeatedly apply *patterns* to *match* inputs and generate outputs. Boxes are defined in the Hume *coordination language*. Output generation is expressed in a functional style in the strict Hume *expression language*. Both coordination and expression languages share a common type system and may employ auxiliary data and function definitions.

Hume has been designed as a *layered* language where the types permitted on wires and the expressiveness of output generation determine the layer's decidability properties. Full Hume, has rich polymorphic types and general recursion. PR-Hume restricts expressions to primitive recursion (i.e. bounded minimisation). HO-Hume only has higher-order functions for repetition. FSM-Hume has finite types (fixed size integers, floats; fixed length vectors) and no recursion. HW-Hume has only bits and tuples.

We are exploring the translation of Hume into the contemporary object oriented language Java. While Java has many design and implementation infelicities, it has a number of pragmatic attractions as a target language. First of all, core Java has very wide cross platform and cross system support through the implementation independent Java Virtual Machine (JVM). Thus, core Java enjoys a very high degree of portability. Secondly, there are very substantial class libraries for Java.

[1]School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Scotland, EH14 4AS, `chunxu@macs.hw.ac.uk`

[2]School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Scotland, EH14 4AS, `greg@macs.hw.ac.uk`

In principle, compilation to Java should ease exploitation of such libraries from Hume.

The following sections present our current progress. Following an overview of Hume and the introduction of an illustrative Hume example, we discuss the representation of Hume types, expressions and boxes in Java. We then briefly survey the status of our Hume to Java compiler, and consider related and future work.

1.2 HUME LANGUAGE OVERVIEW

Hume provides the following core types:

- fixed precision characters, integers, reals and words, and booleans;
- fixed width tuples of mixed type;
- fixed width vectors of single type;
- variable length lists of single type;
- user defined constructed types.

with associated operators.

Hume also provides the following expression constructs:

- arithmetic and boolean expressions;
- conditional expressions;
- case expression with pattern matching;
- local definitions;
- recursive functions defined by case;
- exceptions;
- timeouts.

Hume boxes specify:

- box name;
- names and types of inputs and outputs;
- input match patterns with associated outputs expressions;
- exception handling;
- timeout handling.

Finally, Hume wires specify:

- connections between boxes;

- connections from boxes to input/outputs streams and ports;
- initial values for wires;

The macro notation enables the compile-time generalisation and repeated instantiation of boxes and wiring.

1.3 EXAMPLE: FULL ADDER

We will present our approach to translating Hume to Java through the example of a single bit full adder shown in Figure 1.1.

Note that the half adder is actually written in HW-Hume, the Hume layers oriented to hardware modelling. In HW-Hume, types, patterns and values are restricted to possibly nested tuples of bits. However, this example is adequate for explaining the central features of our translation approach.

The full adder is composed of two virtual half adders, which are each formed from a fanout box:

```
type Bit = word 1;

template fanout
in (x,y::Bit)
out (x1,y1,x2,y2::Bit)
match
 (x,y) -> (x,y,x,y);
```

an xor box:

```
template xor
in (x,y::Bit)
out (z::Bit)
match
 (0,0) -> 0 |
 (0,1) -> 1 |
 (1,0) -> 1 |
 (1,1) -> 0;
```

and an and box:

```
template and
in (x,y::Bit)
out (z::Bit)
match
 (0,0) -> 0 |
 (0,1) -> 0 |
 (1,0) -> 0 |
 (1,1) -> 1;
```

As two copies of each are required, they are defined as *templates* for replication:

```
instantiate fanout as f*2;
instantiate xor as x*2;
instantiate and as a*2;
```

To test the adder, the box `gen` successively generates all possible three bit binary combinations on its outputs `c`, `x` and `y`:

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

box gen
in (t::Next)
out (t'::Next,x,y,c::Bit)
match
 (0,0,0) -> ((0,0,1),0,0,0) |
 (0,0,1) -> ((0,1,0),0,1,0) |
 (0,1,0) -> ((0,1,1),1,0,0) |
 (0,1,1) -> ((1,0,0),1,1,0) |
 (1,0,0) -> ((1,0,1),0,0,1) |
 (1,0,1) -> ((1,1,0),0,1,1) |
 (1,1,0) -> ((1,1,1),1,0,1) |
 (1,1,1) -> ((0,0,0),1,1,1);
```

Note that feedback between `t` and `t'` is used to control the box's state.

First of all, `gen`'s `x` and `y` outputs are fed to a virtual *half adder*. The *fanout* box `f1` duplicates its inputs as `x1/y1` and `x2/y2`s. `x1/y1` are x-ored by box `x1` to give the sum and and-ed by box `a1` to give the carry. The sum from `x1`'s output is wired with the carry from `gen` to another virtual half adder. The second fanout box `f2` duplicates these values. One pair are x-ored by box `x2` to give the final sum. The other pair are and-ed by box `a2`, and the result is or-ed with the carry from `a1` by box `o` to give the final carry:

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

The final sum and carry are then fed to box `show` which is wired to the stream `output` which is associated with standard output:

```
box show
in (s,c::Bit)
out (sc::(Bit,Bit,char))
match
 (s,c) -> (s,c,'\n');
```

```
stream output to "std_out";
```

The adder's components are wired together by position:

```
wire gen (gen.t' initially (0,0,0)) (gen.t,f1.x,f1.y,f2.x);
wire f1 (gen.x,gen.y) (x1.x,x1.y,a1.x,a1.y);
wire x1(f1.x1,f1.y1)(f2.y);
wire a1 (f1.x2,f1.y2)(or.x);
wire f2 (gen.c,x1.z) (x2.x,x2.y,a2.x,a2.y);
wire x2(f2.x1,f2.y1)(show.s);
wire a2 (f2.x2,f2.y2)(or.y);
wire or (a1.z,a2.z) (show.c);
wire show (x2.z,or.z) (output);
```

Note that `gen`'s `t` input is initialised to `(0,0,0)`.

1.4 TRANSLATING HUME TO JAVA

1.4.1 Hume Types in Java

All Hume types in Java are sub-classes of the root `HT` class. Thus, all types effectively have boxed representations. Base Hume types are represented by common classes with values from the corresponding base Java types. Hume tuples are represented by type-specific classes with fields of `HT` for the elements. Similarly, a Hume vector is represented by a type-specific class with an array of `HT`. Hume constructed types are represented by type-specific classes which are sub-typed for the constructor values. Every class has methods to return required fields and to convert an instance to a string, as well as a constructor method.

For example, in the adder, we represent `type Bit = word 1;` by:

```
public final class HTBit extends HT
{ private boolean contents;
  public HTBit(boolean contents) {...}
  public String toString() {...}
  public boolean getContents() {...}
}
```

and `type Next = (Bit,Bit,Bit)` by:

```
public class HTTupleNext extends HT
{ private HT contents1;
  private HT contents2;
  private HT contents3;
  public HTTupleNext(HT contents1, HT contents2, HT contents3){...}
  public String toString() {...}
  public HT getContents1() {...}
  public HT getContents2() {...}
```

```
  public HT getContents3() {...}
}
```

Constants are instances of the corresponding classes, for example:

```
public abstract class Constant
{  public static HTBit BIT0 = new HTBit(false);
   public static HTBit BIT1 = new HTBit(true);
   public static HTTupleNext BIT000 = new HTTupleNext(BIT0,BIT0,BIT0);
   public static HTTupleNext BIT001 = new HTTupleNext(BIT0,BIT0,BIT1);
   ...
}
```

Note that `Constant` is declared as `abstract`: it is a global class that is never instantiated.

1.4.2 Hume Expression Language in Java

Every class for a Hume base type has methods corresponding to the associated type operations. Simple and conditional expression translation follows the obvious naive correspondence with the equivalent Java constructs. Case expressions with pattern matches are translated into the equivalent nested branching Java conditional constructs. However, no Java variables are created for local variables introduced by patterns: such local variables are aliases for the associated values which are manipulated directly through appropriate fields of the equivalent Java `HT` values.

The adder example only contains trivial expressions so we illustrate expression translation with the insert function from the Hume 'people sort' example, designed to expose a large number of language features in a small program for development purposes:

```
type Record   = (vector 1 .. 3 of char, int 64);

insert n [] = [n];
insert n (h: t) = if n<h
                  then n:h:t
                  else h:insert n t;
```

Note that comparison is overloaded in Hume so if `(h:t)` is a list of `Record` then < will perform element wise comparison on the `Records` `n` and `h`.

Given appropriate classes a `Record` and list of `Record`, `insert` translates to:

```
  public static HT insert(HTTupleRecord n, HTTupleRecordList l)
  {  HTTupleRecordList list;
     if (l.equals(Constant.HTTupleRecordListNULL))
      list = new HTTupleRecordList(n, l);
     else
```

```
    if (n.compareTo(l.getHead()) < 0)
     list = new HTTupleRecordList(n, l);
    else
    {  list = l;
       list.setTail((HTTupleRecordList) insert(n, l.getTail()));
    }
    return list;
  }
```

1.4.3 Hume Coordination Language in Java

The Hume execution model is based on non-terminating, round-robin, one-shot scheduling of boxes. Execution proceeds in a series of cycles as follows:

```
for each box
  state ← RUNNABLE
forever
  for each box
    if state!=BLOCKED then
      copy values from input wires
      if some pattern matches input values then
        generate associated output values
        consume values from input wires
        state ← SUCCESS
      else
        state ← MATCHFAIL
    if state!=MATCHFAIL then
      if previous values on output wires not consumed then
        state ← BLOCKED
      else
        put output values on output wires
        state ← RUNNABLE
```

Wires are based on a common class:

```
public class Wire
{  private HT value;
   public HT look() {...}
   public void reset() {...}
   public void put(HT value) {...}
   public boolean isFull() {...}
}
```

Boxes are based on a general class:

```
public abstract class HBox
{  protected abstract void getInput();
```

```
  protected abstract void putOutput();
  protected abstract boolean checkOutput();
  protected abstract boolean match(int currentNo);

  private final int RUNNABLE = 1,MATCHFAIL = 2,BLOCKED = 3;
  private int matchState = RUNNABLE;
  private int matchFirst = 0;
  private int matchMaxNo;
  private boolean matchFair;

  protected void setMatchMaxNo(int matchMaxNo) {...}
  protected void setMatchFair(boolean matchFair) {...}
  private boolean establish() {...}
```

The `match` method performs fair or unfair matching:

```
private boolean match()
{ int current = matchFirst;
  boolean matchSuccess = false;
  this.getInput();
  do
  { if(matchSuccess = this.match(current)) break;
    current++;
    if(current > this.matchMaxNo) current = 0;
  } while(current != this.matchFirst);
  if(this.matchFair && matchSuccess) {
  { this.matchFirst = current + 1;
    if(this.matchFirst > this.matchMaxNo) this.matchFirst = 0;
  }
  return matchSuccess;
}
```

The method `run` corresponds closely to the inner loops of the execution model schema above:

```
  public void run()
  { if(this.matchState!=BLOCKED)
     if(this.match()) this.matchState = RUNNABLE;
     else this.matchState = MATCHFAIL;
    if (this.matchState != MATCHFAIL)
     if (this.establish()) this.matchState = RUNNABLE;
     else this.matchState = BLOCKED;
  }
}
```

Here, the specific box's `match` finds a matching pattern and generates outputs, and its `establish` checks that old outputs have been consumed, consumes inputs and asserts new outputs.

Individual boxes are then formed by sub-classing `HBox`. For example, the `gen` box is as follows. First the wires and associated local buffers are declared:

```
public class HBoxgen extends HBox
{ private HT t; private Wire tWire;
  private HT t1; private Wire t1Wire;
  private HT x; private Wire xWire;
  private HT y; private Wire yWire;
  private HT c; private Wire cWire;
```

Next, the constructor is defined to establish the wires and other initial values:

```
  public HBoxgen(Wire tWire, Wire t1Wire, Wire xWire, Wire yWire,
                 Wire cWire, int matchMaxNo,boolean matchFair)
  {  this.tWire = tWire;
     this.t1Wire = t1Wire;
     this.xWire = xWire;
     this.yWire = yWire;
     this.cWire = cWire;
     this.setMatchMaxNo(matchMaxNo);
     this.setMatchFair(matchFair);
  }
```

The `HBox` wire I/O methods are implemented for the box's specific wires:

```
  protected void getInput()
  {  this.t = tWire.look(); }

  protected void putOutput()
  {  if (this.t1 != null) this.t1Wire.put(this.t1);
     if (this.x != null) this.xWire.put(this.x);
     if (this.y != null) this.yWire.put(this.y);
     if (this.c != null) this.cWire.put(this.c);
  }

  protected boolean checkOutput()
  {  if (this.t1 != null && this.t1Wire.isFull()) return false;
     if (this.x != null && this.xWire.isFull()) return false;
     if (this.y != null && this.yWire.isFull()) return false;
     if (this.c != null && this.cWire.isFull()) return false;
     return true;
  }
```

The `HBox` `match` method is also implemented to reflect the box's body; here a simple transcription from the corresponding Hume:

```
  protected boolean match(int currentNo)
  {  boolean matchSuccess = false;
```

```
    switch (currentNo) {
    {  case 0:
         if (this.t != null && this.t.equals(Constant.BIT000))
         {   matchSuccess = true;
             this.tWire.reset();
             this.t1 = Constant.BIT001;
             this.x = Constant.BIT0;
             this.y = Constant.BIT0;
             this.c = Constant.BIT0;
         }
         break;
       ...
    }
    return matchSuccess;
  }
}
```

Note that as a specific box is an instance of the corresponding class, this also provides a simple basis for an implementation of templates.

Note that streams are also sub-classed boxes.

Finally, the program class defines the wires and box instances:

```
public class ProgramAdder {
  Wire hw_gen_t1_to_gen_t = new Wire();
  ...
  HBoxgen hBoxgen =s
    new HBoxgen(hw_gen_t1_to_gen_t,...};
  ...
  HBox[] hBoxs = new HBox[] {hBoxgen,...};
```

initialises the wires:

```
  public ProgramAdder()
  {  hw_gen_t1_to_gen_t.put(Constant.BIT000); }
```

and runs the scheduler:

```
  public void run()
  {  while (true) {
      for (int i = 0; i < hBoxs.length; i++)
       hBoxs[i].run();
  }

  public static void main(String[] args)
  {  new ProgramAdder().run(); }
}
```

1.4.4 Current Status

We have hand translated a number of Hume examples into Java. Initially we experimented with realising Hume boxes as Java threads. However, threads lack a formal semantics so we next explored the use of JCSP, Welsh's Java embedding of CSP[Wel99], realising boxes as processes. However, this proved over elaborate given that Hume has a sequential scheduling policy, so we finally adopted our current approach of boxes as unprivileged objects. This has enabled us to clarify the correspondences between Hume and Java, and has provided a rich set of general classes for Hume types, boxes and wires, for use in our compiler.

We have constructed two parsers for Hume that construct Java abstract syntax trees, from Hume concrete syntax and from Hume abstract syntax as output by Hammond's parser. Both parsers were built using the JavaCC compiler-compiler. We are currently completing Java generation for boxes and wires, and anticipate being able to process HW-Hume shortly thereafter. We are embedding the compiler in a prototype IDE for Hume which currently supports the drawing of box diagrams from Hume programs.

1.4.5 Related Work

A number of projects have used Java or Java byte code as the target for functional languages. Thus, Kawa[Bot98] is a compiler from Scheme Lisp to Java, and MLj[BKR98] and IncH[dLZ00] are compilers to Java byte code for Standard ML and Hope respectively. Similarly, Wakeling discusses the compilation of Haskell to Java byte-code[Wak97]. Note that these languages all correspond to the Hume expression language.

Fekete and Richard[FR98] discuss the translation of the reactive language Esterel into Java. Esterel has concurrent constructs analogous to the Hume coordination language but with considerably weaker expressive power.

In work closely related to that discussed here[HM03], Hammond translated Michaelson's Hume version of Burns and Wellings' Ada mine pump system[BW01] into Java. Hammond's approach is similar to ours in defining a general Java class for wires. However, our approach differs in:

- using boxed representations of types that subclass a general type class, rather than directly using unboxed Java base types;
- creating boxes by sub-classing a general box class, rather than defining separate classes;
- basing box scheduling on sequential methods, rather than on concurrent threads.

1.5 FUTURE WORK

In developing Hume to Java translation, we also wish to satisfy a central objective in Hume's development, to establish that language processing tools are consistent with the Hume definition and meaning preserving ([MK02] p7). Thus, we wish

to adopt Stepney's approach to high integrity compilation [Ste93] and prove that the meaning of a compiled Hume program relative to Java's formal semantics is the same as the program's meaning in terms of the Hume formal semantics. If P_H is a Hume program, M_H is the Hume semantic function, T_J^H is a translator from Hume to Java and M_J is the Java semantic function, then we wish to prove:

$$M_H(P_H) = M_J(T_J^H(P_H))$$

Semantics of Hume are well defined[Ham01]. Several operational semantics for Java have been developed, for example Atalli et al's Typol in the Jitan environment[Ata98] and Nipkow's Jinja within Isabelle/HOL[Nip03].

Our current objective is to complete the HW-Hume to Java compiler and the formalisation of the translation of HW-Hume to Java. We will then start to explore the correctness of HW-Hume to Java translation while developing translation and compilation for higher Hume language layers.

ACKNOWLEDGEMENTS

Chunxu Liu is supported by a scholarship from Heriot-Watt University.

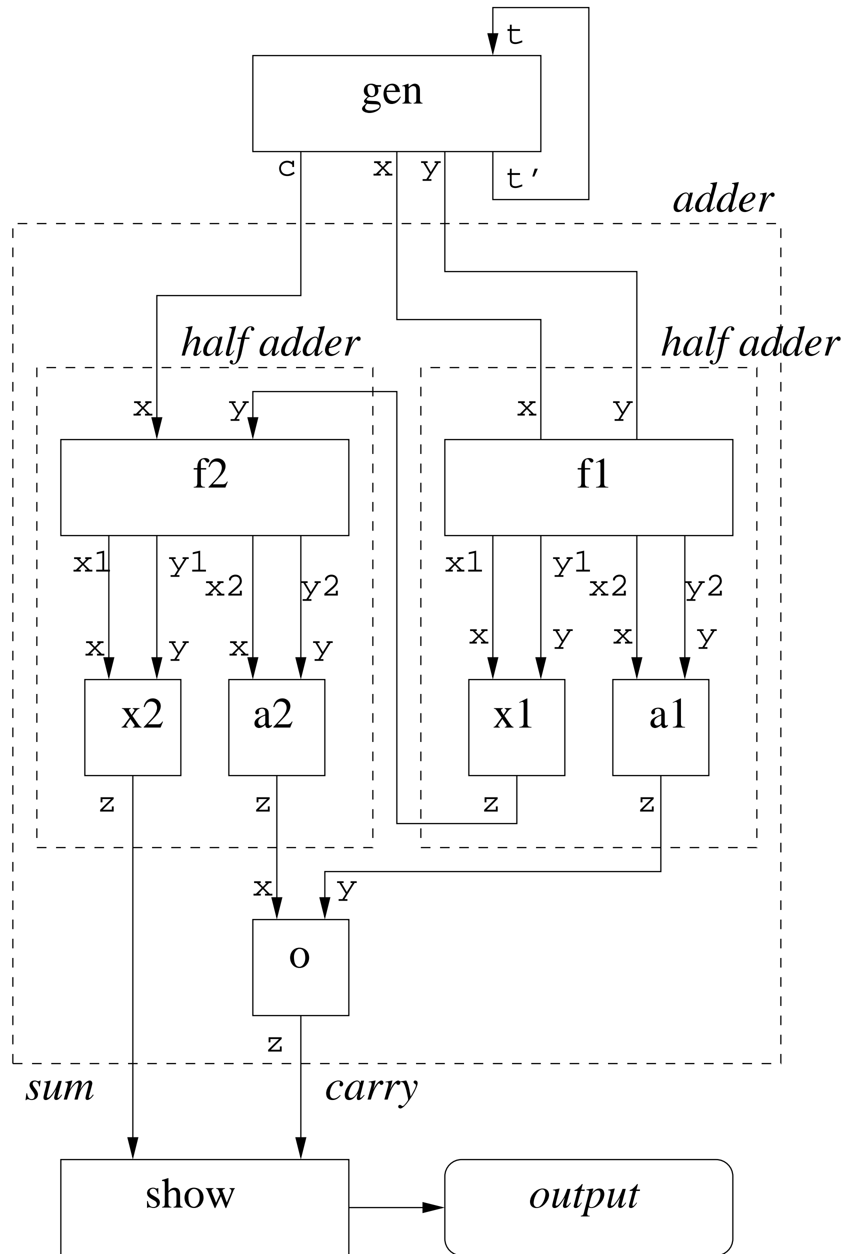

FIGURE 1.1. Full Adder

REFERENCES

[Ata98] I. Atalli. A Formal and Executable Semantics for Java. In *Proceedings of Workshop on Formal Underpinings of Java, OOPSLA'98, Vancouver, Canada*. Technical Report, Princeton University, 1998.

[BKR98] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java Bytecode. In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming, Baltimore, USA*. ACM Press, 1998.

[Bot98] P. Bothner. Kawa: Compiling Scheme to Java. In *Proceedings of Lisp Users Conference*, 1998.

[BW01] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley Longman, 2001.

[dLZ00] J. G. de Lamadrid and J. Zimmerman. The IncH Hope Compiler for the Java Virtual Machine. In M. Mohnen and P. Koopman, editors, *Draft Proceedings of 12th International Workshop on Implementation of Functional Languages, Aachen, Germany*, pages 347–362. Aachener Informatik-Berichte, 00-7, September 2000.

[FR98] J-D. Fekete and M Richard. Esterel Meets Java: Building Reactive Synchronous Programs in Java. Technical Report Rapport 98-3-info, Ecole des Mines de Nantes, December 1998.

[Ham01] K. Hammond. The Dynamic Properties of Hume: a Functionally-Based Concurrent Language with Bounded Time and Space Behaviour. In *Proc. Impl. of Funct. Langs. (IFL 2000), Aachen, Germany, Sept. 2000*, number 2011, pages pp. 122–139. Springer-Verlag Lecture Notes in Computer Science, 2001.

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

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

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

[Nip03] T. Nipkow. Jinja: Towards a Comprehensive Formal Semantics for a Java-like Language. In *Proceedings of Marktoberdorf Summer School*, 2003. To appear.

[Ste93] S. Stepney. *High Integrity Compilation: a case study*. Prentice-Hall, 1993.

[Wak97] D. Wakeling. A Haskell to Java Virtual Machine Code Compiler. In C. Clack, K. Hammond, and T. Davie, editors, *Proceedings of 9th International Workshop on Implementation of Functional Languages, St Andrews, Scotland*, pages 39–52. Springer, LNCS 1467, September 1997.

[Wel99] P. Welch. *Concurrent Programming in Java: Design Principles and Patterns(2nd Edition)*. Addison-Wesley, 1999.