

Towards a Hume IDE

Chunxu Liu and Greg Michaelson
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Scotland, EH14 4AS2s
chunxu@macs.hw.ac.uk or
greg@macs.hw.ac.uk

Abstract The Java-based Hume IDE provides a classic edit/run/debug text-based interactive programming environment, augmented with the Hume Diagram Tool(HDT) which displays two-dimensional box/wiring diagrams from Hume source programs. Diagram layout may be modified interactively and wires may be inspected to identify associated types.

Here, we present an overview of the IDE, focusing on the HDT. We also present a preliminary performance evaluation of the Hume-Java compiler; another major Hume IDE component.

1 Introduction

Hume[MK02] is a strongly typed programming language based on concurrent finite state machines with transitions defined through pattern matching and recursive functions. It is designed to support a high level of expressive power and provide strong guarantees of dynamic behavioural properties such as execution time and space usage.

Hume consists of the expression language and coordination languages, which share a common declaration language.

The expression language is a purely functional, recursive language with strict semantics. The properties of determinism, termination and bounded time and space behaviour may be statically proved by type system and semantics.

The coordination language is a finite state language. It describe multiple, interacting, re-entrant processes built from the purely functional expression layer. It has statically provable properties. These properties include both process equivalence and safety properties such as the absence of deadlock, live-lock or resource starvation. The basic units of the coordination language are boxes, with pattern matching transitions between inputs and outputs, and wires, that link boxes to each other and the external environment. The coordination language also provides exception-handling facilities including

x	y	c	z	c'
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

Figure 1: One Bit Full Adder

timeouts and system exceptions, and is responsible for interaction with the external environment through streams, associated with files or standard I/O, and ports, associated with devices.

The declaration language enables the introduction of constants, type aliases, data types and functions for use in the expression and coordination language.

A Hume program consists of one or more boxes with inputs and outputs. When a program is running, each box repeatedly tries to get its inputs and match one of its patterns. If match is succeeds, it will calculate the value of the associated expression for its outputs.

We are developing an experimental Java-based IDE for Hume which includes a Hume to Java compiler and visualisation support for coordination constructs. In the following sections, we provide a motivating overview of Hume, and discuss the design of the Hume Diagram Tool and its use within the IDE. Finally we present a preliminary performance evaluation of the Hume to Java compiler, compared with other Hume implementations.

2 Diagramming Hume Programs

Consider a one bit full adder based on the truth table shown in Figure 1.

The following program for the full adder is written in the HW-Hume language subset aimed at hardware descriptions. It is based on the classic combination of two half adders with an OR, where each half adder is composed of an AND and an XOR.

```
1 type Bit = int 1;
2 type Next = (Bit,Bit,Bit);
```

Lines 1 and 2 introduce type aliases for one bit and a tuple of three bits.

```
3 box gen
4 in (t::Next)
```

```

5 out (t'::Next,x,y,c::Bit)
6 match
7 (0,0,0) -> ((0,0,1),0,0,0) |
8 (0,0,1) -> ((0,1,0),0,1,0) |
9 (0,1,0) -> ((0,1,1),1,0,0) |
10 (0,1,1) -> ((1,0,0),1,1,0) |
11 (1,0,0) -> ((1,0,1),0,0,1) |
12 (1,0,1) -> ((1,1,0),0,1,1) |
13 (1,1,0) -> ((1,1,1),1,0,1) |
14 (1,1,1) -> ((0,0,0),1,1,1);

```

Lines 3 to 14 define a box to generate and log test cases.

```

15 template fanout
16 in (x,y::Bit)
17 out (x1,y1,x2,y2::Bit)
18 match
19 (x,y) -> (x,y,x,y);

20 instantiate fanout as f*2;

```

Lines 15 to 19 define a box template to fanout two copies of its inputs and line 20 instantiates two copies.

```

21 template xor
22 in (x,y::Bit)
23 out (z::Bit)
24 match
25 (0,0) -> 0 |
26 (0,1) -> 1 |
27 (1,0) -> 1 |
28 (1,1) -> 0;

29 instantiate xor as x*2;

```

Lines 21 to 28 define an XOR box template and line 29 instantiates two copies.

```

30 template and
31 in (x,y::Bit)
32 out (z::Bit)
33 match
34 (0,0) -> 0 |
35 (0,1) -> 0 |
36 (1,0) -> 0 |

```

```
37 (1,1) -> 1;
```

```
38 instantiate and as a*2;
```

Lines 30 to 38 define an AND box template and line 39 instantiates two copies.

```
39 box or
40 in (x,y::Bit)
41 out (z::Bit)
42 match
43 (0,0) -> 0 |
44 (0,1) -> 1 |
45 (1,0) -> 1 |
46 (1,1) -> 1;
```

Lines 39 to 46 define an OR box.

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

Lines 47 to 51 define a box to display the program output for each test case.

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

```
53 wire gen (gen.t' initially (0,0,0)) (gen.t,f1.x,f1.y,f2.x);
54 wire f1 (gen.x,gen.y) (x1.x,x1.y,a1.x,a1.y);
55 wire x1(f1.x1,f1.y1)(f2.y);
56 wire a1 (f1.x2,f1.y2)(or.x);
57 wire f2 (gen.c,x1.z) (x2.x,x2.y,a2.x,a2.y);
58 wire x2(f2.x1,f2.y1)(show.s);
59 wire a2 (f2.x2,f2.y2)(or.y);
60 wire or (a1.z,a2.z) (show.c);
61 wire show (x2.z,or.z) (output);
```

Line 52 defines a stream linked to standard output. Finally lines 53 to 61 wire the boxes to each other and to the stream.

As is clear from this example, it rapidly becomes difficult to understand the linear text of Hume programs composed of more than one or two boxes. Comprehending individual boxes is relatively straightforward but envisaging the wiring of a whole program is somewhat more demanding. Incidentally, if templates had not been used, this simple example would involve details of ten boxes and take up around 100 lines of text.

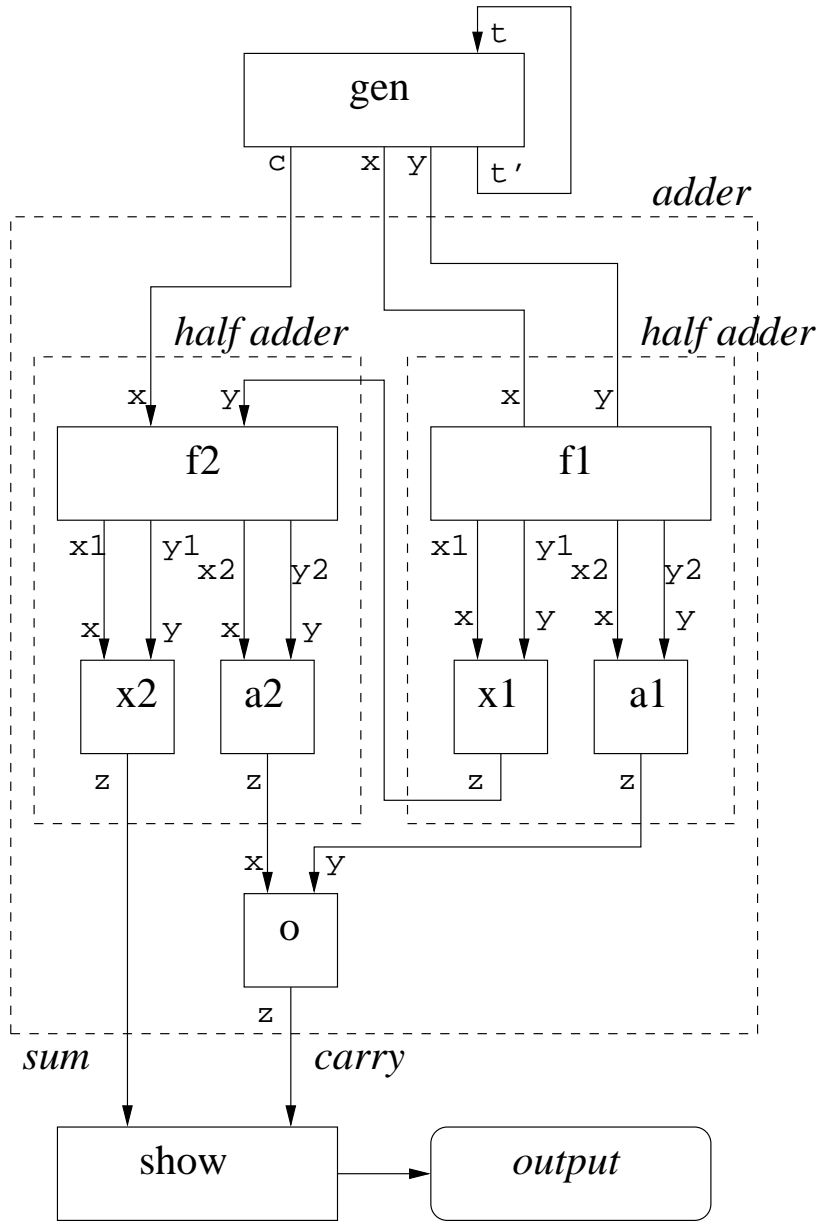


Figure 2: Full Adder

Hume programmers¹ have evolved ad-hoc diagrams for developing and explaining multi-box programs. Such diagrams tend to correspond to the coordination language, focusing on box wiring and eliding most detail of pattern matches and types. For the example above, Figure 2 is typical.

However, drawing such figures by hand is incredibly time consuming. A major motivation for the Hume IDE was to support at least diagrammatic visualisation of extant Hume programs.

3 IDE and Diagram Design

Text-based interactive development environments (IDEs) for programming languages, such as JBuilder [LSSL02] for Java, have a long pedigree. Typically, these support interactive program entry, editing, execution and trace-based debugging.

There is also a long history of interactive visualisation and diagramming aids for programming languages but until recently few have found much currency. Contributing factors may be that:

- there are no standards for representing languages diagrammatically;
- diagrams make poor use of screen space compared with text;
- icon choice for atomic constructs tends to be ad-hoc and obscure;
- with a diagram/text distinction for overview/fine detail, moving between the mouse for diagrams and the keyboard for code text detail is unwieldy.

However, UML now offers a language independent standard for object-oriented program design based on class diagrams. Tools directly supporting UML, such as Rational Rose [BB02], or augmented with UML support through plug-ins, such as Eclipse[Hol04], now enjoy wide use for OO software development.

Curiously, there have been many experimental visualisation tools and IDEs for functional languages[Yan01], but none seem to be in general use. Of course, where imperative language implementations have huge commercial user bases and are substantial sources of profit in their own right, functional languages tend to be developed in research environments and depend on enthusiasts for support and maintenance. Nonetheless, perhaps real functional programmers use `vi`?

We decided that our Hume IDE would be based on the classic text-based edit/execute/debug model, augmented with diagramming support for the coordination language. We also decided to focus at first on displaying extant

¹Yes, plural...

programs as diagrams and to consider program generation from diagrams at a later stage.

When we started building the IDE, there were two Hume implementations, the reference interpreter and the abstract machine compiler/interpreter, which shared the same Haskell front end (`hparse`) for lexical analysis, parsing and elaboration of box and wiring macros. Thus, we decided to adopt the Hume abstract syntax from this front end as the basis for program manipulation and representation in the IDE.

The Hume Diagram Tool(HDT), implemented in JavaCC[Col] and Java, is driven by the BNF for the `hparse` ADT. From the text for an ADT, it generates an object-based internal representation with a node for each non-terminals symbol of the BNF. From this internal representation, the HDT builds objects to store each box and its wire information. Then, HDT calculates box and wire display positions, and draws a box diagram in a window.

Diagrams are based on coordination language constructs only, with a labelled rectangle for each box and stream. To minimise clutter, multiple wires between one box and another are elided to a single wire, and inputs and outputs are not labelled or typed. However, wires may be selected to display input/output names and types in a separate window. Because it is very difficult to automatically avoid lines crossing, we allow a user to drag the box to other positions so that line crossing is minimised and to clarify the logical relations between boxes.

4 IDE Use

When running the IDE, the window shown in Figure 3 appears.



Figure 3: The Hume IDE window

We can input Hume source or open an existing file by selecting File—Open or the Open button. After we open a file (`adder.hume`), the window in Figure 4 appears.

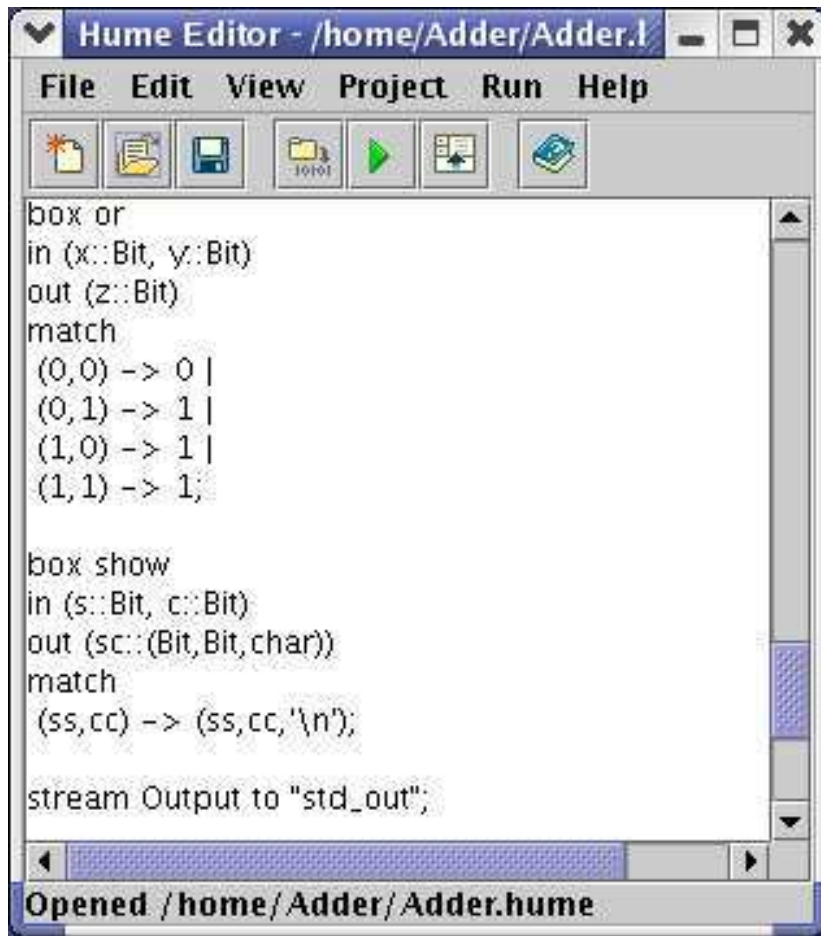


Figure 4: The Hume Diagram Tool window with a Hume file

Clicking Project—Box Picture draws the Hume box picture, as shown in Figure 5.

We can drag boxes to other positions so that we can see the box picture more easily, as shown in Figure 6. The HDT will automatically swap wire positions to minimise wire cross-over.

If we want to look at the wire information, we can select the vertical part of a wire line. This will display type and connectivity details, as shown in Figure 7.

Finally, selecting Run—Run compiles the current Hume program to Java, compiles this to JVM code and runs it, displaying output in a new window. In profiling mode, when the program halts after a given number of execution cycles, the IDE displays statistics for each box about how often it was runnable, blocked or failed to match, cumulative and average execution times, and the longest delay for any input to be consumed, as shown in

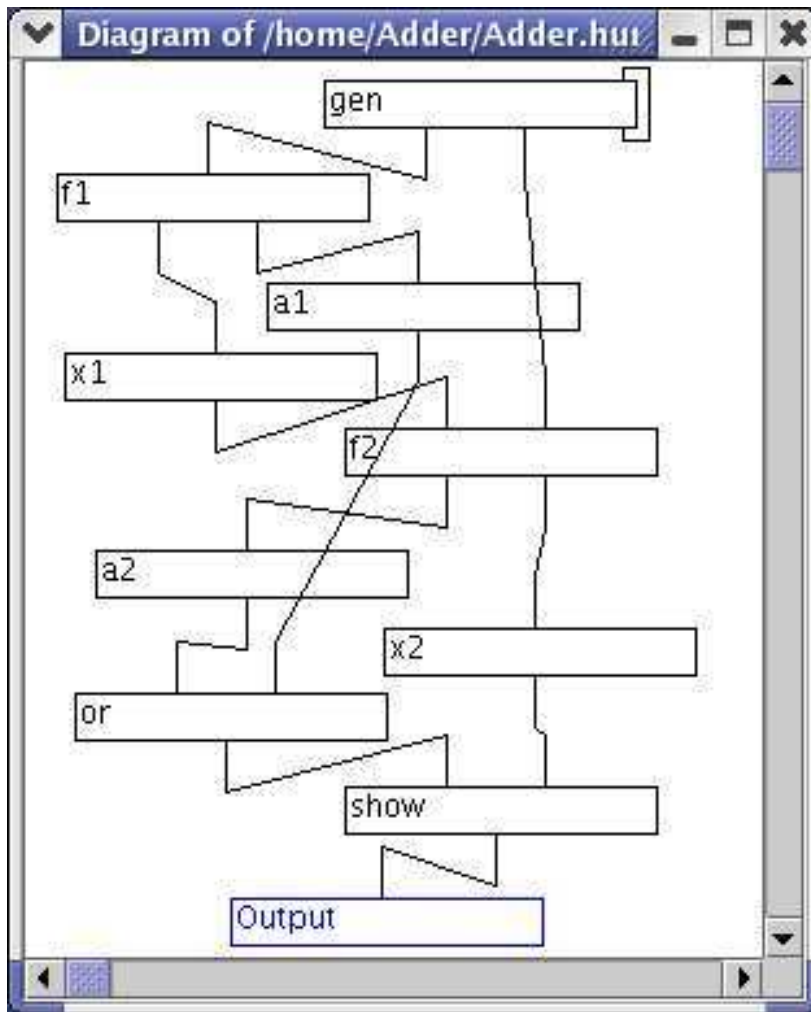


Figure 5: Box Diagram

Figure 8.

Finally, support is also provided for IDE and compiler development. When we finish editing the source file, we can select Project—Hume Tree to show the Hume syntax tree. It will save the file automatically, if we have modified the file. It then calls the Hume to Java parser and displays the syntax tree as shown in Figure 9. Similarly, selecting Project—Parser Tree generates and displays the `hparse` output syntax tree, as shown in Figure 10.

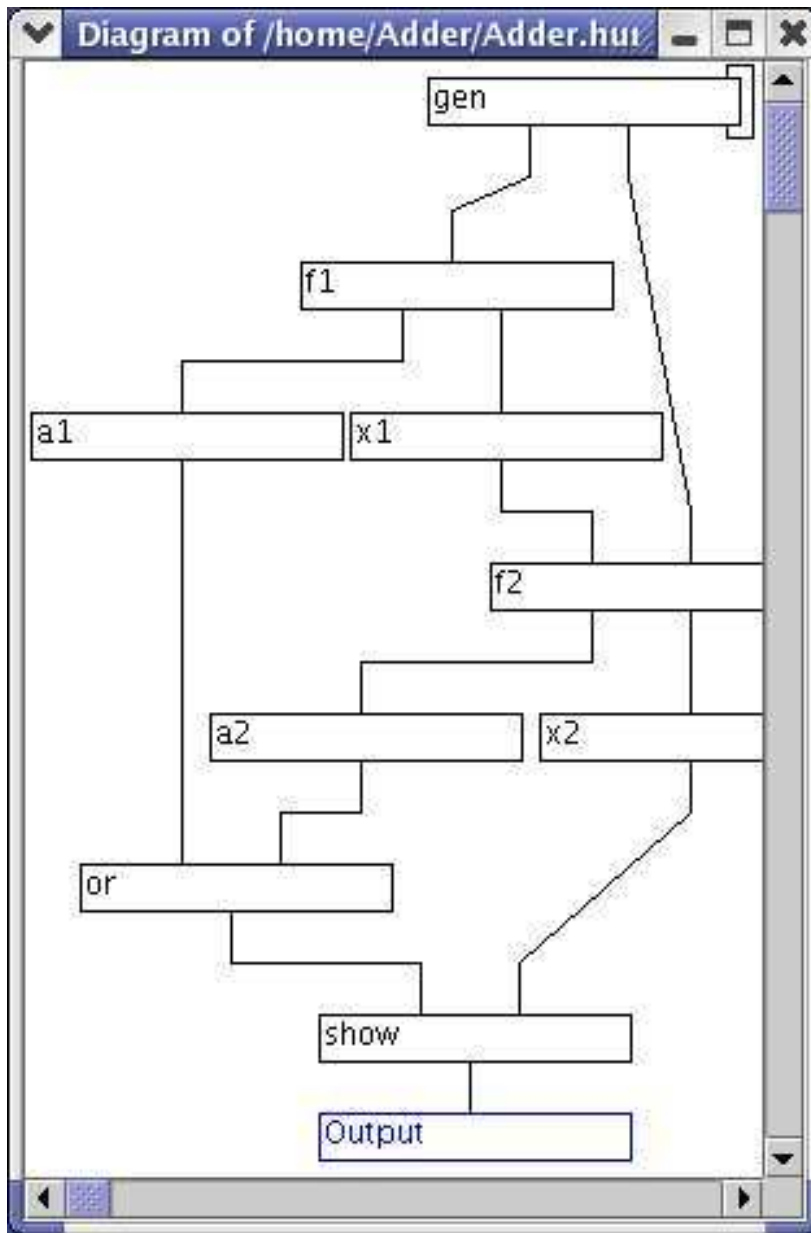


Figure 6: Modified Box Diagram

5 Hume Implementation Comparison

As an initial evaluation of the HW-Hume to Java compiler[LM04], we compare it with the reference interpreter and the abstract machine compiler/interpreter. The reference interpreter is written in Haskell and the `hami` abstract machine

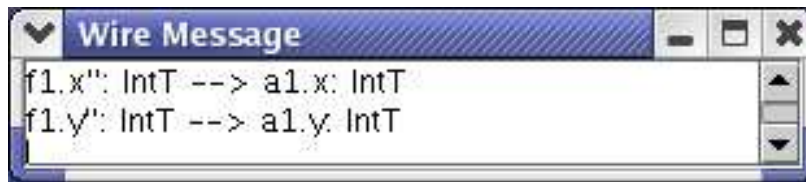


Figure 7: Wire information

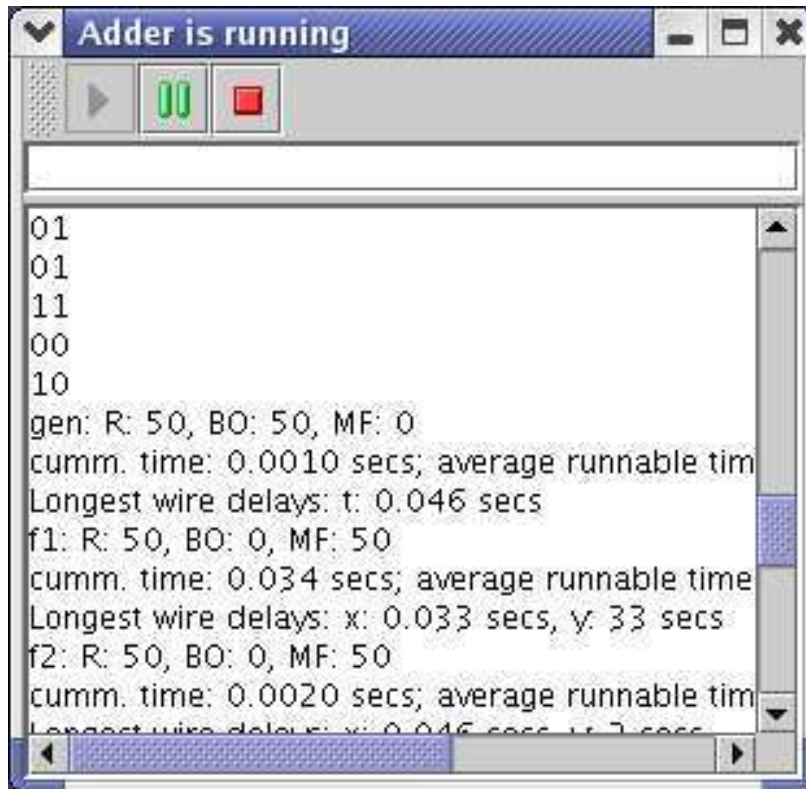


Figure 8: Run window

interpreter in C. Ultimately, both are compiled with `gcc`.

First we contrast the size of the full adder program in elaborated Hume, abstract machine code (Ham) generated by the `phamc` compiler, Java generated by the Hume to Java compiler, and Java Virtual Machine code (JVM) generated by the JDK compiler:

Language	lines	characters	chars/Hume chars
Hume	92	1.5K	1.0
Ham	759	8.8K	5.9
Java	1750	48K	32.0
JVM		36K	24.0

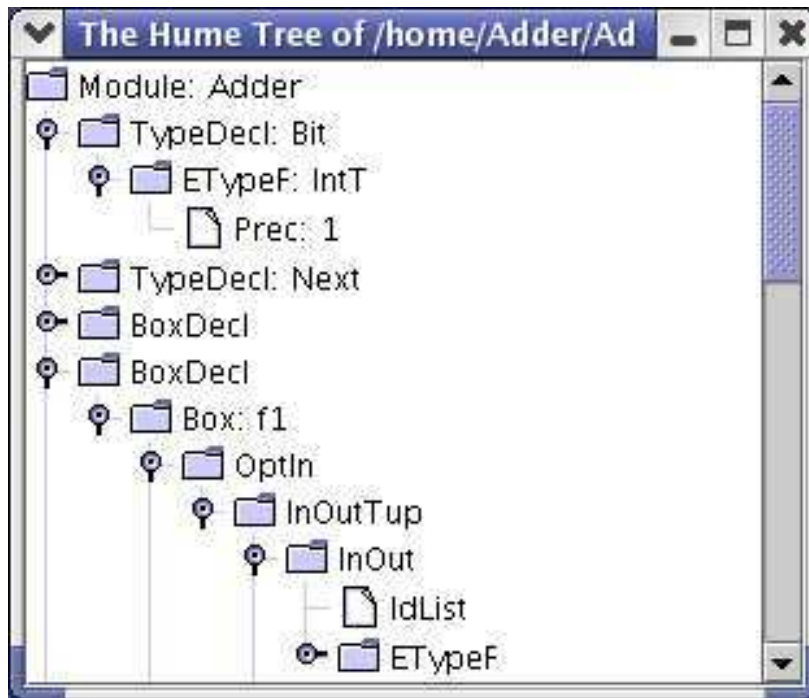


Figure 9: Syntax tree

The original Hume is considerably more compact than Ham, Java or JVM code. The relatively naive Java translation is 32 times the size of the Hume. However, this includes standard generic classes for representing Hume constructs.

Running the adder for 10,000 execution cycles on the different implementations, on a 1.6GHz Pentium 4 with 256 MB memory, gave the times:

System	time (secs)	% of interpreter
Interpreter	7.93	100.0
hami	2.17	27.4
JVM	3.45	43.5

The Ham is the fastest implementation, at nearly four times as fast as the interpreter, with Hume to Java around two and a half times as fast. However, the Hume to Java is a relatively naive translation, with considerable scope for optimisation.

6 Related Compiler Work

A number of projects have used Java or Java byte code as the target for functional languages. Thus, MLj[BKR98] and IncH[dLZ00] are compilers to

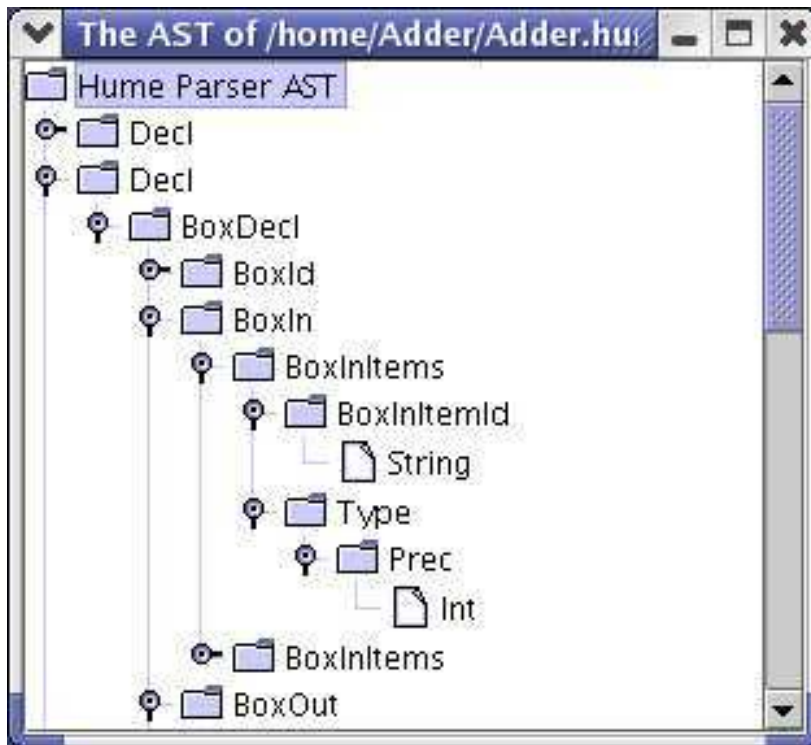


Figure 10: `hparse` output tree

Java byte code for Standard ML and Hope respectively. Similarly, Wakeling discusses the compilation of Haskell to Java byte-code[Wak97].

Kawa[Bot98] is a compiler from Scheme Lisp to Java. Perhaps the closest to our work is Koser et al's SML2Java[KLV03], which compiles Standard ML to Java source.

Note that these languages all correspond to the Hume expression language.

7 Future Work

The current compiler only support HW-Hume. We are steadily extending it to handle full Hume. We also plan to use the JIT just-in-time compiler for Java with Hume to Java compiler output.

At present, the HDT is driven by `hparse` output. We plan to extend it to work directly with Hume source via the parser from the Hume to Java compiler. Finally, in the longer term we plan to explore the generation of Hume from diagrams drawn and augmented with code within the IDE. We also plan to animate Hume diagrams with run-time trace and profiling information, to aid program development and debugging.

Acknowledgements

We would like to thank Kevin Hammond and Robert Pointon for their help and advice. Chunxu Liu is supported by a scholarship from Heriot-Watt University. This research is also supported by EU FP6-510255 EmBounded.

References

- [BB02] W. Boggs and M. Boggs. *Master UML with Rational Rose 2002*. Sybex, 2002.
- [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.
- [Col] Inc. CollabNet. Java Compiler Compiler (JavaCC) - The Java Parser Generator. <http://javacc.dev.java.net/>.
- [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.
- [Hol04] S. Holzner. *Eclipse*. O'Reilly, 2004.
- [KLV03] J. Koser, H. Larsen, and J. Vaughan. SML2Java: a source to source translator. In *Proceedings of DP-Cool, PLI'03, Uppsala, Sweden*, 2003.
- [LM04] C. Liu and G. Michaelson. Translating Hume to Java. In H-W. Loidl, editor, *Draft Proceedings of 5th Symposium on Trends in Functional Programming, Ludwig-Maximillians-Universität, Munich, Germany*, pages 113–128, November 2004.
- [LSSL02] M. Landy, S. Siddiqui, J. Swisher, and M. Lundy. *Borland JBuilder Developer's Guide*. Sams, 2002.
- [MK02] G. Michaelson and K. Hammond. The Hume Language Definition and Report, Version 0.2. Technical report, Heriot-Watt University and University of St Andrews, January 2002.
- [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.
- [Yan01] J. Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2001.