A Haskell to Java Virtual Machine Code Compiler

David Wakeling

Department of Computer Science, University of Exeter,
Exeter, EX4 4PT, United Kingdom.
(web: `http://www.dcs.exeter.ac.uk/~david`)

Abstract. For some time now, we have been interested in using Haskell to program inexpensive embedded processors, such as those in SUN's new Java family. This paper describes our first attempt to produce a Haskell to Java Virtual Machine code compiler, based on a mapping between the G-machine and the Java Virtual Machine. Although this mapping looks good, it is not perfect, and our first results suggest that the compiled Java Virtual Machine code may be rather larger and slower than one might hope.

1 Introduction

For some time now, we have been interested in the efficient implementation of lazy functional programming languages on very small computers, such as those found in consumer electronics devices. So far, all of our implementations have assumed that next-generation products will be controlled by previous-generation RISC processors [Wak95]. But Java processors, with their compact instruction encoding, are an attractive alternative [SUN97]. This paper investigates whether these processors could successfully run lazy functional programs.

The paper has two parts. The first part points out the similarity between the virtual machine usually used to implement Java [LY96] and the Chalmers G-machine [Pey87], a virtual machine often used to implement lazy functional languages. Section 2 gives a quick tour of the Java Virtual Machine, Section 3 gives a quick tour of the G-machine, and Section 4 describes a mapping between the two virtual machines that can serve as the basis of a lazy functional language implementation. The second part assesses the effectiveness of the mapping, and suggests how it could be improved. Section 5 presents some benchmark figures, Section 6 discusses these figures, and Section 7 has some ideas for future improvements. Section 8 mentions some related work, and Section 9 concludes.

2 The Java Virtual Machine

In principle, Java could be compiled for any machine, but in practice it is usually compiled for a standard virtual machine. This section gives a quick tour of the Java Virtual Machine; more detail can be found in [LY96]. Throughout

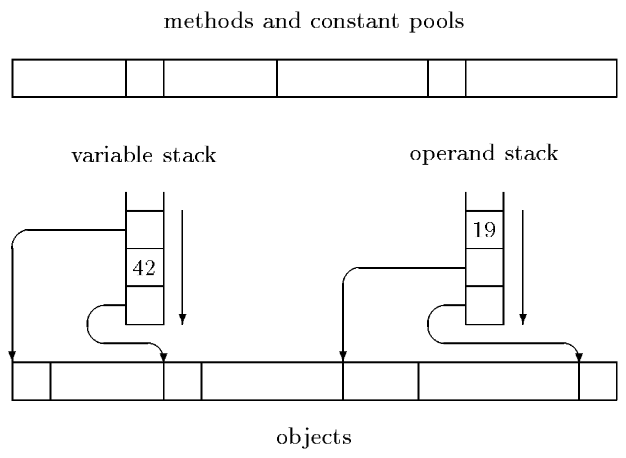

Fig. 1. The Java Virtual Machine.

this paper, Java source code and Java Virtual Machine code will be written in `typewriter` font.

As Figure 1 shows, the Java Virtual Machine is a stack-based virtual machine which works with *methods and constant pools*, *objects*, and two *stacks*.

2.1 Methods and Constant Pools

A Java program is organised into *classes*, each of which may have some *methods* (or functions) for performing computation. For every class, the Java Virtual Machine stores the virtual machine code for each method, and a *constant pool* of literals, such as numbers and strings, used by the methods. To ensure the binary portability of Java programs, the layout and byte-order of the stored form is precisely specified. Nevertheless, before the Java Virtual Machine runs any untrusted code it *verifies* it in an attempt to ensure that it is well-behaved.

2.2 Objects

As well as providing methods, classes also describe the structure of *objects*. An object is a record whose fields may be either scalar values, methods or references to other objects. There are virtual machine instructions for allocating a new object, for setting and getting the value of a field, and for invoking a method. But there is no instruction for disposing of an object. A *garbage collector* is assumed to run from time-to-time to recover the memory occupied by objects that are no longer in use.

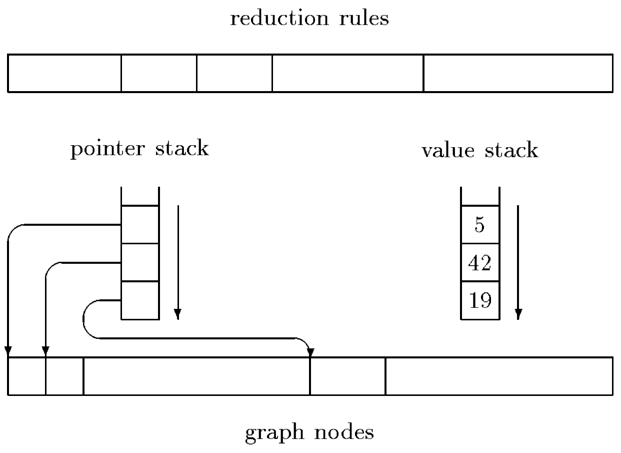

Fig. 2. The G-machine.

2.3 Stacks

The Java Virtual Machine stack is divided into *frames*, each of which stores the local state of a method invocation. In order to simplify our exposition, we have chosen to separate this stack into a *variable stack* from which space for actual parameters and local variables is allocated, and an *operand stack* from which space for the intermediate results of expression evaluations is allocated.

3 The G-machine

Many of the most successful lazy functional language implementations are based on some variant of the Chalmers G-machine. This section gives a quick tour of the G-machine; more detail can be found in [Pey87]. Throughout this paper, G-machine instructions will be written in upper-case SANS SERIF font.

As Figure 2 shows, the G-machine is a stack-based virtual machine which works with *graph nodes*, *reduction rules* and two *stacks*.

3.1 Graph Nodes

A lazy functional program is executed by evaluating an expression to normal form, printing the result as it becomes available. The G-machine represents the expression by a *graph*, and it evaluates it by *graph reduction*. As the graph is reduced, new nodes are attached to it and existing nodes are detached from it. From time-to-time, a *garbage collector* recovers the memory occupied by detached nodes.

3.2 Reduction Rules

Each function or primitive operation serves as a *reduction rule* for the graph. By way of example, Figure 3 describes simple functional programs as a set of reduction rules of varying arity. The body of each rule is an expression which can be either a function, an argument, an integer, an addition, a conditional, or an application. Continuing with the example, Figure 4 shows how the reduction rules can be compiled into G-machine code. Executing the G-machine code for a rule reduces the graph in the manner required by that rule.

$$
\begin{aligned}
p ::= {} & f_1 = \lambda x_{11} \cdots \lambda x_{1m_1}.e_1 \\
& \vdots \\
& f_n = \lambda x_{n1} \cdots \lambda x_{nm_n}.e_n \\
\\
e ::= {} & f \\
\mid {} & x \\
\mid {} & i \\
\mid {} & e_1 \ + \ e_2 \\
\mid {} & \textbf{if } e_1 \ e_2 \ e_3 \\
\mid {} & e_1 \ e_2
\end{aligned}
$$

Fig. 3. Program syntax.

3.3 Stacks

In order to make garbage collection easier, the G-machine stores the local state of a function application on two stacks, both of which can be divided into *frames*. The *pointer stack* stores pointers to argument and intermediate graph nodes, and the *value stack* stores unboxed constants.

4 A Mapping Between Virtual Machines

Figures 1 and 2 and the accompanying text are intended to suggest the mapping between the G-machine and the Java Virtual Machine discussed below.

4.1 Mapping Graph Nodes to Objects

The mapping of the G-machine's graph nodes to the Java Virtual Machine's objects is largely straightforward. It is natural to represent general graph nodes by a class, `N`, and particular graph nodes by subclasses of that class. Figure 5 shows the abstract class, `N`. Here, the `ev` method returns a node representing this

$$\mathcal{F}\llbracket f = \lambda x_1 \cdots \lambda x_n.e \rrbracket = \mathcal{R}\llbracket e \rrbracket\,[\,x_1 = n, x_2 = n-1 \cdots x_n = 1\,]\; n$$

$$\begin{array}{ll}
\mathcal{R}\llbracket e_1 + e_2 \rrbracket\,\rho\, n & = \mathcal{E}\llbracket e_1 + e_2 \rrbracket\,\rho\, n;\ \mathsf{UPDATE}\ (n+1);\ \mathsf{POP}\ n;\ \mathsf{UNWIND} \\
\mathcal{R}\llbracket \textbf{if}\, e_1\ e_2\ e_3 \rrbracket\,\rho\, n & = \mathcal{B}\llbracket e_1 \rrbracket\,\rho\, n\ \mathsf{JFALSE}\ l; \\
 & \qquad \mathcal{R}\llbracket e_2 \rrbracket\,\rho\, n; \\
 & \quad \mathsf{LABEL}\ l; \\
 & \qquad \mathcal{R}\llbracket e_3 \rrbracket\,\rho\, n \\
\mathcal{R}\llbracket e \rrbracket\,\rho\, n & = \mathcal{C}\llbracket e \rrbracket\,\rho\, n;\ \mathsf{UPDATE}\ (n+1);\ \mathsf{POP}\ n;\ \mathsf{UNWIND} \\
 & \\
\mathcal{E}\llbracket i \rrbracket\,\rho\, n & = \mathsf{PUSHINT}\ i \\
\mathcal{E}\llbracket e_1 + e_2 \rrbracket\,\rho\, n & = \mathcal{B}\llbracket e_1 + e_2 \rrbracket\,\rho\, n;\ \mathsf{MKINT} \\
\mathcal{E}\llbracket \textbf{if}\, e_1\ e_2\ e_3 \rrbracket\,\rho\, n & = \mathcal{B}\llbracket e_1 \rrbracket\,\rho\, n;\ \mathsf{JFALSE}\ l_1; \\
 & \qquad \mathcal{E}\llbracket e_2 \rrbracket\,\rho\, n;\ \mathsf{JMP}\ l_2; \\
 & \quad \mathsf{LABEL}\ l_1; \\
 & \qquad \mathcal{E}\llbracket e_3 \rrbracket\,\rho\, n; \\
 & \quad \mathsf{LABEL}\ l_2 \\
\mathcal{E}\llbracket e \rrbracket\,\rho\, n & = \mathcal{C}\llbracket e \rrbracket\,\rho\, n;\ \mathsf{EVAL} \\
 & \\
\mathcal{B}\llbracket i \rrbracket\,\rho\, n & = \mathsf{PUSHBASIC}\ i \\
\mathcal{B}\llbracket e_1 + e_2 \rrbracket\,\rho\, n & = \mathcal{B}\llbracket e_2 \rrbracket\,\rho\, n;\ \mathcal{B}\llbracket e_1 \rrbracket\,\rho\ (n+1);\ \mathsf{ADD} \\
\mathcal{E}\llbracket \textbf{if}\, e_1\ e_2\ e_3 \rrbracket\,\rho\, n & = \mathcal{B}\llbracket e_1 \rrbracket\,\rho\, n;\ \mathsf{JFALSE}\ l_1; \\
 & \qquad \mathcal{B}\llbracket e_2 \rrbracket\,\rho\, n;\ \mathsf{JMP}\ l_2; \\
 & \quad \mathsf{LABEL}\ l_1; \\
 & \qquad \mathcal{B}\llbracket e_3 \rrbracket\,\rho\, n; \\
 & \quad \mathsf{LABEL}\ l_2 \\
\mathcal{B}\llbracket e \rrbracket\,\rho\, n & = \mathcal{E}\llbracket e \rrbracket\,\rho\, n;\ \mathsf{GET} \\
 & \\
\mathcal{C}\llbracket f \rrbracket\,\rho\, n & = \mathsf{PUSHGLOBAL}\ f \\
\mathcal{C}\llbracket x \rrbracket\,\rho\, n & = \mathsf{PUSH}\ n - \rho(x) \\
\mathcal{C}\llbracket i \rrbracket\,\rho\, n & = \mathsf{PUSHINT}\ i \\
\mathcal{C}\llbracket e_1 + e_2 \rrbracket\,\rho\, n & = \mathcal{C}\llbracket e_2 \rrbracket\,\rho\, n;\ \mathcal{C}\llbracket e_1 \rrbracket\,\rho\ (n+1); \\
 & \quad \mathsf{PUSHGLOBAL}\ \mathrm{PLUS};\ \mathsf{MKAP};\ \mathsf{MKAP} \\
\mathcal{C}\llbracket \textbf{if}\, e_1\ e_2\ e_3 \rrbracket\,\rho\, n & = \mathcal{C}\llbracket e_3 \rrbracket\,\rho\, n;\ \mathcal{C}\llbracket e_2 \rrbracket\,\rho\ (n+1);\ \mathcal{C}\llbracket e_1 \rrbracket\,\rho\ (n+2); \\
 & \quad \mathsf{PUSHGLOBAL}\ \mathrm{COND};\ \mathsf{MKAP};\ \mathsf{MKAP};\ \mathsf{MKAP}; \\
\mathcal{C}\llbracket e_1\ e_2 \rrbracket\,\rho\, n & = \mathcal{C}\llbracket e_2 \rrbracket\,\rho\, n;\ \mathcal{C}\llbracket e_1 \rrbracket\,\rho\ (n+1);\ \mathsf{MKAP}
\end{array}$$

Fig. 4. Compilation schemes.

```
abstract public class N {
      public abstract N    ev();
      public abstract void uw();
      public abstract int  gt();
      public abstract N    rn();
      public abstract void ud(N g);
}
```

Fig. 5. The abstract graph node class, `N`.

node in normal form; the `uw` method initiates a further reduction if this node is not in normal form; the `gt` method returns the tag of this node; the `rn` method runs the code associated with this (function) node; and the `ud` method updates this (application) node with a node representing its normal form. Usually, an update involves overwriting the node with either a *copy* of the normal form node, or with an *indirection node* that points to it. Sadly, the Java Virtual Machine does not provide a way to copy one object over another, so one must update by indirection. Thus, each updatable node is represented by an object with an extra indirection field. The indirection field is initially `null`, and the update method sets it to reference a normal form node object. During graph reduction, the indirection field of a node must be followed if it is not `null`. As an example, Figure 6 gives Java code for the application node class `AP`.

Constant Applicative Forms (or CAFs) can also sometimes be a source of trouble for functional language implementors, but here they give no difficulty. There is a class for each CAF, and these classes are all subclasses of the graph node class. Like other updatable nodes, a CAF node object has an indirection field that is initially `null`, and is later set to reference a normal form node object. However, in this case the indirection field is a *class variable* shared between all instances of the CAF class, so that when one is evaluated, the others "feel the benefits".

For a while, we represented graph nodes by a more elaborate class structure. There were subclasses of the node class gathering together constant nodes, function nodes, updatable nodes, and so on. The intention was to gain security by using the Java Virtual Machine's type-checking to trap errors in the implementation of graph reduction, and to save space by having the methods for graph node operations in just the classes where they were needed. This did not work out well, however, because many extra Java Virtual Machine instructions were needed for type-checking. An update, for example, can only take place once it has been checked that the node to be updated is in the subclass of updatable nodes. In a correct implementation, of course, it always is.

4.2 Mapping Reduction Rules to Methods and Constant Pools

Figure 7 shows how G-machine instructions are mapped onto Java Virtual Machine instructions. For a rule of arity n whose G-machine code is gs, $\mathcal{T}\ [\![\ gs\ ]\!]\ n$ gives the Java Virtual Machine code for the `rn` method. As well as producing Java Virtual Machine instructions, some mappings add new entries to the class constant pool, seen here as a map between integers and entries.

4.3 Mapping the Stacks

The G-machine's value stack is easily mapped onto the Java Virtual Machine's operand stack. However, mapping the G-machine's pointer stack onto the Java Virtual Machine's variable stack is a bit harder. The `rn` method supposes that once an application spine has been unwound and rearranged, its arguments can

```
final public class AP extends N {
      N ind, f, a;

      public AP(N f, N a) {
            this.ind = null;
            this.f   = f;
            this.a   = a;
      }

      public N ev() {
            if (this.ind == null) {
                  return this.ind = RT.APev(this);
            } else {
                  return this.ind.ev();
            }
      }

      public void uw() {
            if (this.ind == null) {
                  RT.APuw(this);
            } else {
                  this.ind.uw();
            }
      }

      public N rn() {
            RT.Stop("A.rn()");
            return null;
      }

      public void ud(N g) {
            this.ind = g;
      }
}
```

Fig. 6. The application node class `AP`.

be efficiently accessed as local variables (see Figure 8). This is achieved by unwinding the spine onto a stack array in the run-time system, and then having some prologue code at the start of the `rn` method do the rearrangement whilst transferring the arguments from the stack array to the local variables. Figure 9 gives the methods for evaluating and unwinding an application spine.

$\mathcal{T}$ [[ADD.gs]] sp	$=$ `iadd`; $\mathcal{T}$ [[gs]] sp
$\mathcal{T}$ [[EVAL.gs]] sp	$=$ `aload` sp; `invokevirtual` i; `astore` sp; $\mathcal{T}$ [[gs]] sp
	$pool$ + [$i \mapsto$ `MethodRef N ev ()N`]
$\mathcal{T}$ [[GET.gs]] sp	$=$ `aload` sp; `invokevirtual` i; $\mathcal{T}$ [[gs]] $(sp-1)$
	$pool$ + [$i \mapsto$ `MethodRef N gt ()I`]
$\mathcal{T}$ [[JFALSE $l.gs$]] sp	$=$ `ifeq` l; $\mathcal{T}$ [[gs]] sp
$\mathcal{T}$ [[JMP $l.gs$]] sp	$=$ `goto` l; $\mathcal{T}$ [[gs]] sp
$\mathcal{T}$ [[MKAP.gs]] sp	$=$ `new` i; `dup`; `aload` sp; `aload` $(sp-1)$;
	`invokespecial` j; `astore` $(sp-1)$; $\mathcal{T}$ [[gs]] $(sp-1)$
	$pool$ + [$i \mapsto$ `Class AP`,
	$j \mapsto$ `MethodRef AP <init> (NN)V`]
$\mathcal{T}$ [[MKINT.gs]] sp	$=$ `new` i; `dup`; `dup2_x1`; `pop2`;
	`invokespecial` j; `astore` $(sp+1)$; $\mathcal{T}$ [[gs]] $(sp+1)$
	$pool$ + [$i \mapsto$ `Class INT`,
	$j \mapsto$ `MethodRef INT <init> (I)V`]
$\mathcal{T}$ [[POP $n.gs$]] sp	$= \mathcal{T}$ [[gs]] $(sp-n)$
$\mathcal{T}$ [[PUSH $n.gs$]] sp	$=$ `aload` $(sp-n)$; `astore` $(sp+1)$;
	$\mathcal{T}$ [[gs]] $(sp+1)$
$\mathcal{T}$ [[PUSHBASIC $i.gs$]] sp	$=$ `sipush` i; $\mathcal{T}$ [[gs]] sp
$\mathcal{T}$ [[PUSHGLOBAL $f.gs$]] sp	$=$ `new` i; `dup`; `invokespecial` j; `astore` $(sp+1)$;
	$\mathcal{T}$ [[gs]] $(sp+1)$
	$pool$ + [$i \mapsto$ `Class` f, $j \mapsto$ `MethodRef` i `<init> ()V`]
$\mathcal{T}$ [[PUSHINT $i.gs$]] sp	$= \mathcal{T}$ [[PUSHBASIC i.MKINT.gs]] sp
$\mathcal{T}$ [[UPDATE $n.gs$]] sp	$=$ `aload` $(sp-n)$; `aload` sp; `invokevirtual` i;
	$\mathcal{T}$ [[gs]] $(sp-1)$
	$pool$ + [$i \mapsto$ `MethodRef N ud (N)V`]
$\mathcal{T}$ [[UNWIND.gs]] sp	$=$ `aload` sp; `invokevirtual` i; $\mathcal{T}$ [[gs]] sp
	$pool$ + [$i \mapsto$ `MethodRef N uw (N)V`]

Fig. 7. Instruction translation scheme.

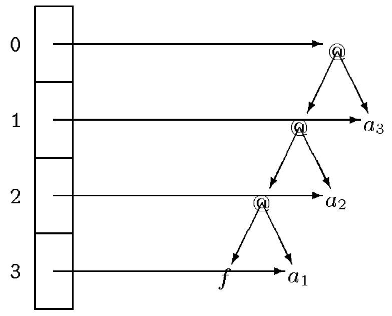

Fig. 8. Argument access via local variables.

```
public static N APev(AP a) {
        APuw(a);
        return S[ sp++ ];
}

public static void APuw(AP a) {
        S[ --sp ] = a;
        a.f.uw();
}
```

Fig. 9. Application node methods.

4.4 Tail Calls

A tail call occurs when the result of one function is given by a call to another function with exactly the right number of arguments supplied. In this case, an efficient implementation discards the frame of the first function before allocating that for the second. Unfortunately, the Java Virtual Machine is not required to implement tail calls like this, and so we must use a well-known trick to achieve the proper behaviour. The trick is a variant of the "UUO handler" invented for the Rabbit Scheme compiler [Ste78] and later popularised as the "tiny interpreter" by the Glasgow Haskell compiler [Jon92]. A tail call is made by returning a function node object whose `rn` is to be invoked, rather than by invoking this method directly. Execution is then controlled by the one-line tiny interpreter:

```
while (f != null) f = f.rn();
```

The UNWIND instruction eventually encounters a normal form, which it leaves on the top of the stack array and returns a `null` object. This implementation of tail calls looks costly because a temporary function node object must be created for each tail call. It can be avoided in the important special case when a function makes a tail-call to itself; here a `goto` instruction suffices. But in general it seems to be the best that can be done, given that there are no function pointers in the Java Virtual Machine.

5 Benchmarks

A compiler from Haskell to Java Virtual Machine code has been constructed using the ideas described above, based on version 0.9999.3 of the Chalmers HBC compiler. This compiler has been used to compile seven benchmark programs: `nfib30`, the unfashionable benchmarking function with type `Int -> Int` and argument 30; `calendar`, a program that formats 7 calendars; `clausify`, which converts propositional formulae to clausal form; `soda`, which performs a word search in a 20×30 grid; `infer`, a type-checker written in the monadic style;

`parser`, a Haskell parser; and `prolog`, a logic programming system solving the Towers of Hanoi problem for six discs.

All of the benchmark figures reported here were recorded on a SUN Ultra 140 workstation with 64Mbytes of memory, running the Solaris 2.5.1 operating system. The machine was running version 1.1.3 of the SUN Java Developer's Kit, which performs "just-in-time" compilation of Java Virtual Machine code to native code. Tables 1 and 2 compare three implementations: our compiler, an unmodified version 0.9999.3 Chalmers HBC compiler, and the Nottingham Hugs interpreter. For our compiler, program sizes are obtained by adding up the sizes of all the generated ".class" files; for the HBC compiler, they are as measured by the `size` command; for the Hugs interpreter, no program size figures are given because the G-machine code that it interprets cannot easily be extracted from the interactive environment. None of the sizes includes type classes and functions from the standard prelude, or support routines from the run-time system. It is not easy to pick-out bits of the prelude, and in any case none of the implementations treat it significantly differently from other Haskell code. Since our compiler has only a minimal run-time system, comparisons of run-time system size would not be fair. For our compiler and the HBC compiler, the timings are an average of those for five consecutive runs after an initial "warm up" run; for Hugs they are an average of five runs, each made immediately after starting the interpreter.

Benchmark	Our compiler Time (s)	Hugs Time (s)	HBC Time (s)
`nfib30`	50.0	106.7	1.5
`calendars`	9.6	4.8	0.1
`clausify`	14.2	7.4	0.5
`soda`	8.0	2.7	0.2
`infer`	35.7	8.2	0.9
`parser`	61.5	8.8	0.9
`prolog`	76.2	9.9	1.3

Table 1. Execution times.

6 Discussion

These benchmark results are disappointing. Our compiler produces programs that are between half and three quarters of the size of those produced by the ordinary HBC compiler, but run between 30 and 60 times more slowly. Indeed, the programs run between 2 and 9 times more slowly than with the Hugs interpreter.

From the point-of-view of embedded applications, the program size is the real concern because a hardware implementation of the Java Virtual Machine can (only) improve program speed. So why are programs so large? Mostly because

Benchmark	Our compiler	Hugs	HBC
	Size (bytes)	Size (bytes)	Size (bytes)
nfib30	2,729	—	2,545
calendars	28,517	—	46,540
clausify	24,152	—	33,693
soda	12,574	—	24,497
infer	204,807	—	222,335
parser	254,007	—	339,192
prolog	99,081	—	124,923

Table 2. Program sizes.

of the way that tail-calls are implemented. Recall that a tail-call is made by returning a function node object to the tiny interpreter. For each such object, there must be a class, and for each such class, there must be a ".class" file. This amounts to a ".class" file for every function and CAF in the program. Each of these has its own methods and constant pool, with no sharing possible between them. In typical programs, we have found that the byte code for Java Virtual Machine instructions accounts for only 40% of the space.

It would be unreasonable, of course, to expect the SUN Java Virtual Machine to run the programs that our compiler produces as fast as the SUN SPARC processor can run the programs that HBC produces. But it is reasonable to expect it to run them as fast as the Hugs interpreter does, because the G-machine used by our compiler is much more sophisticated than the one used by Hugs (for example, it has n-ary application nodes instead of just binary ones). So why are programs so slow? Consider the small program in Figure 10. Figure 10). Its result is the character 'a', the last of a list of 50,000. Computing

```
main = putChar (last legion)

legion = take 50000 (repeat 'a')
```

Fig. 10. A small, allocation-intensive benchmark.

this result involves (lazily) allocating 50,000 "cons" nodes to represent the list, and 50,000 application nodes to represent suspended applications of `take`. In addition, 50,000 function nodes are returned to the tiny interpreter. VSD runs the program in 41.4 seconds, and Hugs in just 3.1 seconds. Even allowing that Hugs does not have to return function nodes to a tiny interpreter, it seems that memory allocation/reclamation is an order of magnitude more expensive in the Java Virtual Machine than in the Hugs run-time system. A more sophisticated G-machine can get back some, but not all of this. Other small, allocation-intensive benchmarks give similar results.

Garbage collection is also a problem. Both the stack array and the local variables are potential sources of space-leaks because the Java Virtual Machine garbage collector cannot know that they are being used as a stack. Thus, it holds onto everything referenced from the array and variables, regardless of where the stack pointer is. This is serious. The input to the larger programs has had to be made smaller to avoid running out of memory because of space-leaks.

Three other points are worth mentioning. Firstly, the Java Virtual Machine uses dynamic linking: at the start of each program run, method references are resolved to memory addresses by loading files from disk. This costs, but not much because the computer's operating system caches ".class" files in memory during the "warm-up" run, and then does not have to access the disk again. Next, the Java Virtual Machine performs/requires run-time checks that are unnecessary for a strongly-typed functional language, such as Haskell. Although memory is never accessed through a `null` pointer, and the pattern-matching code that splits apart a pair never gets anything else, these things are checked anyway. Omitting check instructions can make programs upto 5% faster, but the Java Virtual Machine code is no longer verifiable. Finally, there are the run-time support routines. In the Hugs implementation they are written in C and compiled; in our implementation they are written in Java and interpreted. Compiled code, of course, runs much faster than interpreted code.

7 Future Work

One problem with our implementation is the mapping of the G-machine's pointer stack to a Java array in the run-time system and the Java Virtual Machine's local variables. Pointers must constantly be moved between the array and local variables, and space-leaks occur because the garbage collector cannot know that both are being used as a stack. To some extent, the garbage collection problem could be solved by mapping the pointer stack to a linked-list of frames instead of an array. Taking this idea further, the movement of pointers could also be avoided by using the $\langle \nu, \mathrm{G} \rangle$-machine [AJ89], where stack space is allocated as part of the graph nodes, instead of the ordinary G-machine. We have just built such an implementation, and it looks more promising.

Another problem is with SUN's implementation of the Java Virtual Machine. The high cost of memory allocation/reclamation is quite surprising, and tail calls are not dealt with properly by discarding the frame for one method invocation before allocating a frame for the next. Of course, one can hope that better commercial implementations will appear in future, but in the meantime it it might be interesting to fit out one of the free ones with more efficient storage management — perhaps using semispaces instead of mark/sweep — and a proper treatment of tail-calls.

8 Related Work

The instant success of Java has attracted the attention of others in the functional programming community, notably Odersky and Wadler [OW97]. Their Pizza implementation extends Java with parametric polymorphism, higher-order functions and algebraic data types in order to make these ideas more widely accessible. Pizza can be used as a functional programming language, and object-oriented programming language, or something in between.

Compilers for many other programming languages to Java Virtual Machine code are either available now, or are on the way. For example, Cygnus Support have a compiler called Kawa for Scheme, INRIA-Lorraine have a compiler called SmallEiffel for Eiffel, and Intermetrics have a compiler called Ada Magic for Ada 95.

9 Conclusions

In this paper we have investigated whether Java processors could be programmed with lazy functional languages in the context of embedded applications. Our work is based on a mapping between the G-machine and the Java Virtual Machine. In principle, the mapping is a good one, but in practice programs are not nearly as small or as fast as one might hope. This is largely due to the high cost of memory allocation/reclamation, and the difficulty of implementing tail-calls in the Java Virtual Machine. Despite our disappointing early results, we are not downhearted, and intend to continue developing a Haskell compiler for the Java Virtual Machine.

Acknowledgements

Our thanks as always to Lennart Augustsson and Thomas Johnsson, whose work on the HBC compiler forms the basis of our own.

This work was supported by the University of Exeter Research Fund, and by Canon Research Centre Europe Limited.

References

[AJ89] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle\nu$,g$\rangle$-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 202–213. ACM Press, 1989.

[Jon92] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, pages 127–202, April 1992.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine.* Addison-Wesley, September 1996.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 1997 ACM Symposium on the Principles of Programming Languages*, pages 146–149, January 1997.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Ste78] G. L. Steele. Rabbit: A Compiler for Scheme. Technical Report AI-TR-474, MIT Laboratory for Computer Science, 1978.

[SUN97] SUN Microsystems Inc. PicoJava I Microprocessor Core Architecture. Technical Report WPR-0014-01, SUN Microsystems, 1997.

[Wak95] D. Wakeling. A Throw-away Compiler for a Lazy Functional Language. In M. Takeichi and T. Ida, editors, *Fuji International Workshop on Functional and Logic Programming*, pages 287–300. World Scientific, July 1995.