

Compiling lazy functional programs for the Java Virtual Machine

David Wakeling

School of Engineering and Computer Science,
University of Exeter, Exeter, EX4 4PT,
United Kingdom.
(web: <http://www.dcs.exeter.ac.uk/~david>)

Abstract

In this paper, we show how lazy functional programs can be compiled for the Java Virtual Machine using a mapping between a version of the $\langle \nu, G \rangle$ -machine and the Java Virtual Machine. This mapping is elegant — the description is entirely straightforward — and efficient — using it, both code size and execution speed are of the same order of magnitude as those obtained with a traditional functional language bytecode interpreter. In future, our work could serve as the basis of an interface between Haskell and Java.

1 Introduction

For some time, we have been interested in the implementation of lazy functional languages on small computers, such as those found in consumer electronics devices. Upto now, we have assumed that next-generation products would use previous-generation RISC processors (Wakeling, 1998a). But Java processors, with their compact instruction encoding, are an attractive alternative (O'Connor & Tremblay, 1997). In this paper, we consider software rather than hardware implementations of the Java Virtual Machine, and show that lazy functional languages can be compiled for such implementations with code size and execution speed of the same order of magnitude as those obtained with a traditional functional language bytecode interpreter.

The paper is organised as follows. Section 2 describes the major difficulties in compiling lazy functional languages for the Java Virtual Machine. Section 3 outlines the architecture and instruction set of the $\langle \nu, G \rangle$ -machine, an abstract machine that allows us to overcome most of these difficulties. Section 4 shows how a core functional language can be compiled for this abstract machine. Section 5 outlines the architecture and instruction set of the $\langle \nu, G \rangle$ -machine. Section 6 shows how $\langle \nu, G \rangle$ -machine code can be converted to Java Virtual Machine code. Section 7 presents some benchmark figures for our compiler with the Sun implementation of the Java Virtual Machine, and Section 8 describes some optimisations to the compilation scheme. Section 9 considers some other implementations of the Java Virtual

Machine, and Section 10 compares our compiler with others for Standard ML and Java. Section 11 mentions some closely related work, and Section 12 suggests some possible future work. Section 13 concludes.

2 Implementation Difficulties

This section describes the major difficulties in compiling lazy functional languages for the Java Virtual Machine (Lindholm & Yellin, 1999).

2.1 *The Cost of Memory Allocation*

Our first attempts to compile lazy functional programs for the Java Virtual Machine showed that the cost of memory allocation could be a serious problem. We found that it was *an order of magnitude higher* in version 1.1 of the Sun Java Virtual Machine interpreter than in version 1.3 of the Hugs interpreter (Wakeling, 1997). Since lazy functional programs allocate so much memory for closures, this hurt.

So far, there has not been much incentive for Sun or others to reduce the cost of memory allocation. A Java Virtual Machine implementation is judged by how well it runs benchmarks written in Java, rather than in a lazy functional language, and current implementations usually do an *an order of magnitude less* memory allocation for Java benchmarks than they do for lazy functional ones. Consider, for example, version 3 of Pendragon’s Embedded CaffineMark(tm) suite. This suite does not even have a memory allocation test, and only 0.2% of the Java Virtual Machine instructions executed allocate a new object. But for Haskell programs from the Nofib suite (Partain, 1992) that we have compiled to Java Virtual Machine code, about 2.0% do so.

More realistic Java programs do more memory allocation, and Java Virtual Machine implementors are starting to recognise this by, for example, using generational garbage collectors. Nonetheless, memory allocation remains a bottleneck.

2.2 *Tail Recursion*

In functional programs, the result of one function application is often given by another with exactly the right number of arguments. This is known as *tail recursion*, and when it happens an implementation can save stack space by ensuring that the two function calls use the same frame. But the Java Virtual Machine specification does not require that tail recursive method invocations use the same stack frame. As a result, a straightforward implementation of a tail recursive functional program running on a conforming Java Virtual Machine implementation could easily overflow the stack.

A similar problem was encountered by Steele some years ago, whilst compiling Scheme to C (Steele, 1978). He solved it with the “UWO handler” — a tight loop that continuously called the next function according to the address returned by the previous one. In general, it seems that some variant of this technique must be

used with the Java Virtual Machine, although for small or directly tail recursive functions, inlining or a jump instruction could be used instead.

2.3 The Reduction Stack

Lazy functional languages are usually implemented using some form of *graph reduction* (Peyton Jones, 1987). Expressions are represented by *graphs* and evaluated by *reducing* these graphs to *normal form*. Central to most implementations of graph reduction is a *reduction stack* on which pointers to the graphs representing arguments accumulate before a function is applied. Unfortunately, it is hard to see how the Java Virtual Machine's stack can be used to implement the reduction stack. Even the most basic graph reduction operations, such as unwinding an application spine and then rearranging the pointers to the vertebrae, or determining how many pointers have been pushed, are problematic because the Java Virtual Machine lacks the necessary stack instructions. Using a large array to implement the reduction stack instead can be inefficient because of the cost of bounds-checked Java array accesses, and because of space leaks caused by the Java Virtual Machine's garbage collector not knowing that the array is being used as a stack, and so preserving everything reachable from it (Wakeling, 1997). It might be possible to avoid the space leaks by setting locations above the stack pointer to `null`, but that involves yet more costly array accesses. A less conventional implementation of the reduction stack seems to be needed — one that does not use the Java Virtual Machine's stack or cause space leaks.

2.4 Updating

Laziness involves evaluating a function application at most once. It is achieved by *updating* the root node of the graph representing the function application with the root node of the graph representing its result. Updating a node may be performed by overwriting it with either a new node, or with an indirection pointing to an existing node. Choosing how to update is a slightly tricky part of graph reduction (Peyton Jones, 1987). If the root node of the result graph is new and sufficiently small, then it should be constructed directly on top of the root node of the application graph. Otherwise, an indirection pointing to the root node of the result graph should be constructed on top of the root node of the application graph.

The obvious way for the Java Virtual Machine to represent the graph is by linked objects. However, the Java Virtual Machine provides no way to construct one object directly on top of another, and so updates must be managed in some other way.

3 The $\langle \nu, G \rangle$ -machine

Happily, a version of the $\langle \nu, G \rangle$ -machine (Augustsson & Johnsson, 1989) allows us to overcome most of the difficulties just mentioned. This abstract machine was intended for implementing graph reduction on parallel computers. Its design allows a smooth transition from single-threaded to multi-threaded graph reduction by

allowing more than one point of reduction in the graph, and by making the reduction stack a part of it. This section outlines the architecture and instruction set of our version of the $\langle \nu, G \rangle$ -machine.

3.1 Architecture

The $\langle \nu, G \rangle$ -machine represents expressions by graphs with four kinds of node (see Figure 1). A *constructed value node*, $\mathbf{V} k a$, represents the application of a data

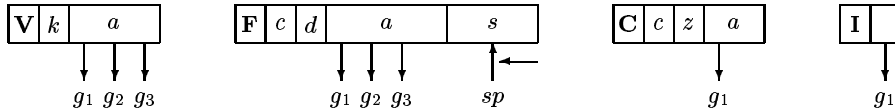


Fig. 1. Constructed value, frame, canonical application and indirection nodes.

value constructor to all of its arguments. The node stores the constructor number, k , and has slots for pointers to the graphs representing the arguments, a . Basic values, such as integers and characters, are represented by constructed value nodes with appropriate values of k and no argument slots. A *frame node*, $\mathbf{F} c d a s$, represents the application of a function to all of its arguments. The node stores the function's code, c , a dynamic link, d , back to the node that requested its reduction, and has slots for pointers to the graphs representing the arguments, a . Additional argument slots are also provided for *local variables*, and for a small *evaluation stack*, s , growing from the last slot of the node. These additional slots are used when the application is reduced. A *canonical application node*, $\mathbf{C} c z a$, represents the application of a function to the first few of its arguments. The node stores the function's code, c , the number of arguments that are missing, z , and has slots for pointers to graphs representing arguments, a (z of which are unused). An *indirection node*, $\mathbf{I} g$, may appear when a frame node is updated, as mentioned earlier.

The advantage of the $\langle \nu, G \rangle$ -machine architecture is that the small evaluation stacks of many frame nodes replace a single large reduction stack, making it possible to avoid space leaks. There are no argument/local variable slots in the original $\langle \nu, G \rangle$ -machine — the evaluation stack is used for everything. As we shall see later, though, the architecture of the Java Virtual Machine makes separating the argument/local variable and stack slots sensible.

3.2 Instruction Set

In order to present the instruction set of the $\langle \nu, G \rangle$ -machine, we need to introduce some notation for lists and graphs. An empty list is written $[]$, and a non-empty list with x as its *head* (first item) and xs as its *tail* (list of remaining items) is written $(x : xs)$. As a shorthand, $[x_1, x_2, \dots, x_n]$ can be written instead of $(x_1 : (x_2 : (\dots x_n : [])))$. The infix operator $++$ concatenates two lists. A graph is a

Instruction	Description
ALOAD i	push pointer from argument/local variable slot i
ASTORE i	pop pointer into argument/local variable slot i
CASE (l_1, l_2, \dots)	pop value and transfer control to appropriate label
DO n	apply an unknown function on the stack to n arguments
EVAL	reduce graph on top of stack to normal form
INITCAP $f_i^k m$	initialise m argument slots of a canonical application node
INITFRM f_i^k	initialise k argument slots of a frame node
INITVAL m	initialise m argument slots of a constructed value node
NUM	pop pointer to constructed value node; push its constructor number
NEWCAP f_i^k	push pointer to new f_i^k canonical application node
NEWFRM f_i^k	push pointer to new f_i^k frame node
NEWVAL m	push pointer to new m -argument constructed value node
POP	pop top stack value
RET	update frame node and return from evaluation
SPLIT $n m$	copy m constructed value node arguments to local variables from n

Table 1. A summary of the $\langle \nu, G \rangle$ -machine instructions.

mapping from node identifiers to nodes, written as

$$\left\{ \begin{array}{l} i_1 \mapsto g_1 \\ i_2 \mapsto g_2 \\ \vdots \\ i_n \mapsto g_n \end{array} \right\}$$

Lists will be used to describe sequences of $\langle \nu, G \rangle$ -machine instructions, as well as the contents of argument/local variable and evaluation stack slots. Graphs will appear as part of the *machine state*, written $\langle \nu, G \rangle$, where G is a graph, and ν is the node identifier of the frame node in G where reduction is currently taking place. Table 1 summarises the $\langle \nu, G \rangle$ -machine instructions. Below, one or more *state transition rules* of the form $\langle \nu_1, G_1 \rangle \Rightarrow \langle \nu_2, G_2 \rangle$ will be given for each. In these rules, graph nodes mentioned in G_1 but not G_2 are assumed to be unchanged, and those mentioned in G_2 but not G_1 are assumed to be new.

Figure 2 gives state transition rules for the various **NEW** and **INIT** instructions that allocate and initialise graph nodes (rules 1 —3, 4 —6). Here and elsewhere, f_i^k stands for a non-primitive function f that happens to be the i th of arity k defined in the program (although all of this information is not always needed in the state transition rules, it will be needed later to convert $\langle \nu, G \rangle$ -machine code to Java Virtual Machine code); $code_k(i)$ stands for the code of this function. A ‘?’ stands for an uninitialised value. Unlike the original $\langle \nu, G \rangle$ -machine, we separate node allocation from initialisation so as to fit better with the Java Virtual Machine, which separates object allocation from initialisation.

Figure 3 gives state transition rules for the **ALOAD** and **ASTORE** instructions that must be provided in our version of the $\langle \nu, G \rangle$ -machine to move pointers between the argument/local variable slots and the evaluation stack of a frame node (rules 7 and 8).

Figure 4 gives the state transition rules for the instructions that reduce a graph to normal form. The **EVAL** instruction examines the node at the top of the evalu-

$$\begin{aligned}
(1) \quad & \langle \nu, \{ \nu \mapsto \mathbf{F} (\text{NEWVAL } m : c) \text{ d a s } \} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} \text{ c d a } (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} ? [?_1, \dots, ?_m] \end{array} \right\} \rangle \\
(2) \quad & \langle \nu, \{ \nu \mapsto \mathbf{F} (\text{NEWFRM } f_i^k : c) \text{ d a s } \} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} \text{ c d a } (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{F} \text{ code}_k(i) ? [?_1, \dots, ?_k] ? \end{array} \right\} \rangle \\
(3) \quad & \langle \nu, \{ \nu \mapsto \mathbf{F} (\text{NEWCAP } f_i^k : c) \text{ d a s } \} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} \text{ c d a } (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} \text{ code}_k(i) ? [?_1, \dots, ?_k] \end{array} \right\} \rangle \\
(4) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{INITVAL } k \ m : c) \text{ d a } (g_m : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} ? [?_1, \dots, ?_m] \end{array} \right\} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} \text{ c d a } (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} \ k [g_1, \dots, g_m] \end{array} \right\} \rangle \\
(5) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{INITFRM } f_i^k : c_1) \text{ d a } (g_k : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{F} \text{ c}_2 ? [?_1, \dots, ?_k] ? \end{array} \right\} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} \text{ c}_1 \text{ d a } (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{F} \text{ c}_2 \ \emptyset [g_1, \dots, g_k] [] \end{array} \right\} \rangle \\
(6) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{INITCAP } f_i^k \ m : c_1) \text{ d a } (g_m : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} \text{ c}_2 ? [?_1, \dots, ?_k] \end{array} \right\} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} \text{ c}_1 \text{ d a } (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} \text{ c}_2 (k - m) [g_1, \dots, g_m] \end{array} \right\} \rangle
\end{aligned}$$

Fig. 2. The various NEW and INIT instructions.

$$\begin{aligned}
(7) \quad & \langle \nu, \{ \nu \mapsto \mathbf{F} (\text{ALOAD } i : c) \text{ d } [\dots, a_i, \dots] \text{ s } \} \rangle \\
& \Rightarrow \langle \nu, \{ \nu \mapsto \mathbf{F} \text{ c d } [\dots, a_i, \dots] (a_i : s) \} \rangle \\
(8) \quad & \langle \nu, \{ \nu \mapsto \mathbf{F} (\text{ASTORE } i : c) \text{ d } [\dots, a_i, \dots] (g : s) \} \rangle \\
& \Rightarrow \langle \nu, \{ \nu \mapsto \mathbf{F} \text{ c d } [\dots, g, \dots] \text{ s } \} \rangle
\end{aligned}$$

Fig. 3. The ALOAD and ASTORE instructions.

ation stack. If it is a constructed value node or a canonical application node then there is nothing to do (rules 9 and 10); if it is an indirection node, then the node pointed to is reduced instead (rule 11); if it is a frame node, then the current point of reduction is saved in the dynamic link of the frame node and then moved to it (rule 12). After the frame node has been reduced, the **EVAL** instruction is tried again. This iterative description of **EVAL** is different from the one in the original $\langle \nu, G \rangle$ -machine paper, and will form the basis of our “UO handler”. The **RET** instruction updates the node at the current point of reduction with an indirection to a result node taken from the evaluation stack. At the same time, it restores the current point of reduction from the dynamic link of the updated node and leaves a pointer to the result node on the evaluation stack of this node too (rule 13). Depending on the context, the original $\langle \nu, G \rangle$ -machine could either update by copying or by indirection. To accommodate the Java Virtual Machine, however, we always update by indirection.

Figure 5 gives state transition rules for the instructions that perform case-analysis.

$$\begin{aligned}
 (9) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : c) d a_1 (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} k a_2 \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} c d a_1 (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} k a_2 \end{array} \right\} \rangle \\
 (10) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : c_1) d a_1 (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} c_1 d a_1 (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \end{array} \right\} \rangle \\
 (11) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : c) d a (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{I} g \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : c) d a (g : s) \\ \nu_1 \mapsto \mathbf{I} g \end{array} \right\} \rangle \\
 (12) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : c_1) d a_1 (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{F} c_2 \emptyset a_2 [] \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu_1, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : c_1) d a_1 s \\ \nu_1 \mapsto \mathbf{F} c_2 \nu a_2 [] \end{array} \right\} \rangle \\
 (13) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{RET} : c_1) \nu_1 a_1 (g : s_1) \\ \nu_1 \mapsto \mathbf{F} c_2 d a_2 s_2 \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu_1, \left\{ \begin{array}{l} \nu \mapsto \mathbf{I} g \\ \nu_1 \mapsto \mathbf{F} c_2 d a_2 (g : s_2) \end{array} \right\} \rangle
 \end{aligned}$$

Fig. 4. The EVAL and RET instructions.

The NUM instruction replaces a pointer to a constructed value node on the evaluation stack with its constructor number (rule 14). A CASE instruction pops this number and uses it to choose which branch to take (rule 15). Should the constructed value node have any arguments, a SPLIT instruction provides access to them by copying them into local variable slots of the frame node at the current point of reduction (rule 16).

$$\begin{aligned}
 (14) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{NUM} : c) d a_1 (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} k a_2 \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} c d a_1 (k : s) \\ \nu_1 \mapsto \mathbf{V} k a_2 \end{array} \right\} \rangle \\
 (15) \quad & \langle \nu, \left\{ \nu \mapsto \mathbf{F} (\text{CASE} (l_1, l_2, \dots) : \dots : l_k : c_k) d a (k : s) \right\} \rangle \\
 & \Rightarrow \langle \nu, \left\{ \nu \mapsto \mathbf{F} c_k d a s \right\} \rangle \\
 (16) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{SPLIT } n m : c) d [a_1, \dots, a_{(n-1)}] (\nu_1 : s) \\ \nu_1 \mapsto \mathbf{V} k [g_1, \dots, g_m] \end{array} \right\} \rangle \\
 & \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} c d [a_1, \dots, a_{(n-1)}, g_1, \dots, g_m] s \\ \nu_1 \mapsto \mathbf{V} k [g_1, \dots, g_m] \end{array} \right\} \rangle
 \end{aligned}$$

Fig. 5. The NUM, CASE, and SPLIT instructions.

Figure 6 gives five state transition rules for the DO instruction that performs a general tail-call by applying a function — unknown at compile-time — to n arguments. Before the function can be applied, its graph must be reduced to a canonical

application node (rules 17 and 18). Thereafter, there are three possibilities to consider. First, if the canonical application node is missing more than n arguments, the result is a new canonical application node missing n fewer arguments (rule 19). Second, if the canonical application node is missing exactly n arguments, the result is a new frame node (rule 20). Thirdly, if the canonical application node is missing less than n arguments, a new frame node must be constructed with the first few of the n arguments, and then another DO instruction must be used to apply this node to the remainder (rule 21).

$$\begin{aligned}
(17) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } n : c_1) d a_1 (g_n : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{I} g \end{array} \right\} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } n : c_1) d a_1 (g_n : \dots : g_1 : g : s) \\ \nu_1 \mapsto \mathbf{I} g \end{array} \right\} \rangle \\
(18) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } n : c_1) d_1 a_1 (g_n : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{F} c_2 d_2 a_2 s_2 \end{array} \right\} \rangle \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{EVAL} : \text{POP} : \text{DO } n : c_1) d_1 a_1 (\nu_1 : g_n : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{F} c_2 d_2 a_2 s_2 \end{array} \right\} \rangle \\
(19) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } n : c_1) d a_1 (g_n : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \end{array} \right\} \rangle, (n < z) \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} c_1 d a_1 (\nu_2 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \\ \nu_2 \mapsto \mathbf{C} c_2 (z - n) (a_2 \text{ ++ } [g_1, \dots, g_n]) \end{array} \right\} \rangle \\
(20) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } n : c_1) d a_1 (g_n : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \end{array} \right\} \rangle, (n = z) \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} c_1 d a_1 (\nu_2 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \\ \nu_2 \mapsto \mathbf{F} c_2 \emptyset (a_2 \text{ ++ } [g_1, \dots, g_n]) [] \end{array} \right\} \rangle \\
(21) \quad & \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } n : c_1) d a_1 (g_n : \dots : g_1 : \nu_1 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \end{array} \right\} \rangle, (n > z) \\
& \Rightarrow \langle \nu, \left\{ \begin{array}{l} \nu \mapsto \mathbf{F} (\text{DO } (n - z) : c_1) d a_1 (g_n : \dots : g_{(z+1)} : \nu_2 : s) \\ \nu_1 \mapsto \mathbf{C} c_2 z a_2 \\ \nu_2 \mapsto \mathbf{F} c_2 \emptyset (a_2 \text{ ++ } [g_1, \dots, g_z]) [] \end{array} \right\} \rangle \\
(22) \quad & \langle \nu, \left\{ \nu \mapsto \mathbf{F} (\text{POP} : c) d a (g : s) \right\} \rangle \\
& \Rightarrow \langle \nu, \left\{ \nu \mapsto \mathbf{F} c d a s \right\} \rangle
\end{aligned}$$

Fig. 6. The DO instruction.

4 Compilation Rules

This section describes how programs written in the core functional language of Figure 7 can be compiled for our version of the $\langle \nu, G \rangle$ -machine. A core program consists of n functions, f_i^k , where f is the i th function of arity k to be defined. A

$$\begin{array}{l}
 p ::= f_1^{k_1} x_{11} \cdots x_{1k_1} = e_1 \\
 \quad \vdots \\
 \quad f_n^{k_n} x_{n1} \cdots x_{nk_n} = e_n \\
 e ::= x e_1 \cdots e_m \qquad (m \geq 0) \\
 \quad | f_i^k e_1 \cdots e_m \qquad (m \geq 0) \\
 \quad | c_k e_1 \cdots e_m \qquad (m \geq 0) \\
 \quad | \mathbf{case } x \mathbf{ in } c_{k_1} x_1 \cdots x_m : e_1 \parallel \cdots \mathbf{end}
 \end{array}$$

Fig. 7. The core language.

core expression is an argument or function applied to zero or more other expressions, a data value constructor applied to all of its arguments, or a **case**-expression on a variable with simple patterns. Ten years on from the original $\langle \nu, G \rangle$ -machine paper, the trend is for core languages to include types, in the hope that they can be used to produce more efficient code. Their absence here reflects a belief that the right choice of abstract machine alone should be enough to obtain the efficiency of a traditional functional language bytecode interpreter, without recourse to complex and expensive optimisations.

Figure 8 collects together the three compilation schemes: \mathcal{F} generates code for a *function*; \mathcal{R} generates code to *return* the value of an expression on top of the stack; and \mathcal{C} generates code to *construct* the graph for an expression, leaving it on top of the stack. In these schemes, ρ is an environment mapping identifiers to argument/local variable slots, and n is the number of the next free slot. As in the original $\langle \nu, G \rangle$ -machine paper, it is assumed that all functional programs can be transformed into core language programs of a form acceptable to these schemes.

A minor difference between our compilation schemes and those of the original $\langle \nu, G \rangle$ -machine paper is that arguments go on the stack in Java Virtual Machine order (first argument pushed first), rather than functional language order (first argument pushed last). A major difference is that the \mathcal{R} scheme deals with all tail recursion by returning a frame node representing the next function application, instead of performing that application directly. A “UUO handler” loop must be used to drive this node to normal form.

5 The Java Virtual Machine

As a prelude to converting $\langle \nu, G \rangle$ -machine code to Java Virtual Machine code, this section outlines the architecture and instruction set of the Java Virtual Machine (Lindholm & Yellin, 1999).

5.1 Architecture

A Java program is organised into *classes*, which have *methods* for performing computation and describe the structure of *objects*. For every class, the Java compiler produces a file containing Java Virtual Machine code for the methods and a *constant pool* of literals, such as numbers and strings, used by this code. The local state

$\mathcal{F} \llbracket f_i^k x_1 \cdots x_k = e \rrbracket$	$= \mathcal{R} \llbracket e \rrbracket \llbracket x_1 = 1, \cdots x_k = k \rrbracket (k + 1)$
$\mathcal{R} \llbracket x \rrbracket \rho n$	$= \mathcal{C} \llbracket x \rrbracket \rho n; \text{RET}$
$\mathcal{R} \llbracket x e_1 \cdots e_m \rrbracket \rho n$	$= \mathcal{C} \llbracket x \rrbracket \rho n; \mathcal{C} \llbracket e_1 \rrbracket \rho n; \cdots \mathcal{C} \llbracket e_m \rrbracket \rho n;$ DO m ; RET
$\mathcal{R} \llbracket f_i^k e_1 \cdots e_m \rrbracket \rho n, (m \leq k)$	$= \mathcal{C} \llbracket f_i^k e_1 \cdots e_m \rrbracket \rho n; \text{RET}$
$\mathcal{R} \llbracket f_i^k e_1 \cdots e_m \rrbracket \rho n, (m > k)$	$= \mathcal{C} \llbracket f_i^k e_1 \cdots e_k \rrbracket \rho n; \mathcal{C} \llbracket e_{k+1} \rrbracket \rho n; \cdots \mathcal{C} \llbracket e_m \rrbracket \rho n;$ DO $(m - k)$; RET
$\mathcal{R} \llbracket c_k e_1 \cdots e_m \rrbracket \rho n$	$= \mathcal{C} \llbracket c_k e_1 \cdots e_m \rrbracket \rho n; \text{RET}$
$\mathcal{R} \llbracket \text{case } x \text{ in } c_{k_1} x_1 \cdots x_m : e_1 \rrbracket \cdots \text{end} \rrbracket \rho n$	$= \text{ALOAD } \rho(x); \text{EVAL}; \text{ASTORE } \rho(x); \text{ALOAD } \rho(x);$ NUM; CASE (l_1, l_2, \cdots) ; l_1 ; ALOAD $\rho(x)$; SPLIT $n m$; $\mathcal{R} \llbracket e_1 \rrbracket (\rho + [x_1 = n, \cdots x_m = n + m - 1]) (n + m)$...
$\mathcal{C} \llbracket x \rrbracket \rho n$	$= \text{ALOAD } \rho(x);$
$\mathcal{C} \llbracket f_i^k e_1 \cdots e_m \rrbracket \rho n, (m < k)$	$= \text{NEWCAP } f_i^k; \mathcal{C} \llbracket e_1 \rrbracket \rho n; \cdots \mathcal{C} \llbracket e_m \rrbracket \rho n;$ INITCAP $f_i^k m$;
$\mathcal{C} \llbracket f_i^k e_1 \cdots e_m \rrbracket \rho n, (m = k)$	$= \text{NEWFRM } f_i^k; \mathcal{C} \llbracket e_1 \rrbracket \rho n; \cdots \mathcal{C} \llbracket e_m \rrbracket \rho n;$ INITFRM f_i^k ;
$\mathcal{C} \llbracket c_k e_1 \cdots e_m \rrbracket \rho n$	$= \text{NEWVAL } m; \mathcal{C} \llbracket e_1 \rrbracket \rho n; \cdots \mathcal{C} \llbracket e_m \rrbracket \rho n;$ INITVAL $k m$;

Fig. 8. The compilation schemes.

of a method invocation is stored on the Java Virtual Machine stack. It consists of the actual parameters and local variables, and a small operand stack for the intermediate results of expression evaluations. An object is a record whose fields may be either scalar values or references to other objects. Storage for objects is allocated from heap memory and later recovered by garbage collection.

5.2 Instruction Set

Table 2 summarises the Java Virtual Machine instructions that we need in this paper. The names of classes, constructors, methods, and fields are all stored as strings in the constant pool, with constructor names appearing as the string “<init>”. Type descriptors are also stored as strings in the constant pool, with constructor and method types encoded as a sequence of argument types between parentheses, followed by a result type.

6 Conversion to Java Virtual Machine Code

To convert $\langle \nu, G \rangle$ -machine code to Java Virtual Machine code, we use the classes described below.

Instruction	Description
<code>aload <i>i</i></code>	push reference from local variable <i>i</i>
<code>astore <i>i</i></code>	pop reference into local variable <i>i</i>
<code>checkcast <i>c</i></code>	cast object on top of stack to class <i>c</i>
<code>dup</code>	duplicate top stack value
<code>getfield <i>c f t</i></code>	get field <i>f</i> of type <i>t</i> from object of class <i>c</i>
<code>getstatic <i>c f t</i></code>	get field <i>f</i> of type <i>t</i> from class <i>c</i>
<code>invokespecial <i>c m t</i></code>	invoke constructor <i>m</i> of type <i>t</i> for object of class <i>c</i>
<code>invokestatic <i>c m t</i></code>	invoke method <i>m</i> of type <i>t</i> of class <i>c</i>
<code>invokevirtual <i>c m t</i></code>	invoke method <i>m</i> of type <i>t</i> on object of class <i>c</i>
<code>lookupswitch (<i>l₁, l₂, ...</i>)</code>	pop value and transfer control to appropriate label
<code>new <i>c</i></code>	push reference to new heap object of class <i>c</i>
<code>pop</code>	pop top stack value
<code>sipush <i>i</i></code>	push integer <i>i</i>

Table 2. A summary of some Java Virtual Machine instructions.

6.1 The Node Class

Figure 9 shows the `N` class that represents graph nodes. Subclasses of `N` will represent constructed value, frame, canonical application and indirection nodes. The `N` class has three abstract methods, `num`, `ev1` and `do1`, and concrete methods `do2`, `do3` In principle, there are an infinite number of these. In practice, eight have proved to be more than enough. To save space, just three are shown. The methods are for converting the `NUM`, `EVAL` and `DO n` instructions: `num` returns the constructor number of a node; `ev1` returns the result of evaluating it to normal form, or to the next tail recursive call; and `don` returns the result of applying it to *n* arguments.

```
public abstract class N {
    public abstract int num();
    public abstract N ev1();
    public abstract N do1(N g1);

    public N do2(N g1, N g2) {
        return this.do1(g1).do1(g2);
    }

    public N do3(N g1, N g2, N g3) {
        return this.do1(g1).do1(g2).do1(g3);
    }

    public N do4(N g1, N g2, N g3, N g4) {
        return this.do1(g1).do1(g2).do1(g3).do1(g4);
    }
    ...
}
```

Fig. 9. The `N` class.

6.2 Constructed Value Node Classes

Constructed value node classes are part of the run-time system, written in Java. Figure 10 shows the V3 class for constructed value nodes of arity three. There are

```
public final class V3 extends N {
    public N s1, s2, s3; public int k;

    public V3(N s1, N s2, N s3, int k) {
        this.s1 = s1; this.s2 = s2; this.s3 = s3; this.k = k;
    }

    public int num() { return this.k; }

    public N ev1() { return this; }

    public N do1(N g1) { RT.Stop("V3.do1()"); return null; }
}
```

Fig. 10. The V3 class.

three slot fields, `s1`, `s2` and `s3`, used for the arguments, and a field for the constructor number, `k`. The constructor is used to convert the `INITVAL` instruction (rule 4), while, the `num` and `ev1` methods take care of the `NUM` and `EVAL` instructions (rules 14 and 9). A `do1` method must also be provided, but in a correct implementation it should never be invoked.

6.3 Frame, Canonical Application and Indirection Node Classes

Our compiler produces a class for each different function arity in the program, including any functions used from the standard prelude. This reduces the number of classes that need to be produced, but does not limit the arities of the functions that can be written. By way of example, Figure 11 sketches a skeleton `FC3` class for all non-primitive functions f_i^3 in some program. To make matters clearer, the class is written in Java. In reality, however, our compiler produces the equivalent Java Virtual Machine code directly. The class has six fields: an indirection field, `ind`, used when the node is updated; three slot fields, `s1`, `s2`, and `s3` used for the arguments; a missing argument count field, `z`, used to record how many slots are unused; and a function number, `i`, used to identify the function f_i^3 of the frame/canonical application node. There are no fields for local variables or the evaluation stack, for which the Java Virtual Machine's local variables and stack are used.

The `FC3` class has four constructors. The first three are used to convert the `INITCAP` instruction (rule 6), and the last the `INITFRM` instruction (rule 5). A `num` method must be provided, but once again it should never be invoked.

The `ev1` method is used to convert the `EVAL` instruction. It tests for an indirection node, which must be bypassed (rule 11), and for a canonical application node, which is already in normal form (rule 10). The remaining possibility is a frame node that

```

public final class FC3 extends N implements Cloneable {
    private N ind, s1, s2, s3; private int z, i;

    public FC3(int i) {
        this.z = 3; this.i = i;
    }

    public FC3(N s1, int i) {
        this.s1 = s1; this.z = 2; this.i = i;
    }

    public FC3(N s1, N s2, int i) {
        this.s1 = s1; this.s2 = s2; this.z = 1; this.i = i;
    }

    public FC3(N s1, N s2, N s3, int i) {
        this.s1 = s1; this.s2 = s2; this.s3 = s3; this.i = i;
    }

    public int num() { RT.Stop("FC3.num()"); return 0; }

    public N ev1() {
        if (this.ind != null) return this.ind.ev1();
        else if (this.z > 0) return this;
        else {
            /* local 1 = this.s1; local 2 = this.s2; local 3 = this.s3 */
            this.s1 = null; this.s2 = null; this.s3 = null;
            switch (this.i) {
                case 1: /* code for function number 1 of arity 3 */ break;
                case 2: /* code for function number 2 of arity 3 */ break;
                ...
            }
            /* this.ind = local 1; */ return this.ind;
        }
    }

    public N do1(N g1) {
        if (this.ind != null) return this.ind.do1(g1);
        else if (this.z == 0) return RT.UUO(this).do1(g1);
        else {
            try { FC3 f = (FC3) this.clone();
                switch (f.z) {
                    case 1 : f.s3 = g1; f.z = 0; return f;
                    case 2 : f.s2 = g1; f.z = 1; return f;
                    case 3 : f.s1 = g1; f.z = 2; return f;
                }
            } catch (CloneNotSupportedException e) { RT.Stop("FC3.do1()"); }
            return null; /* not reached */
        }
    }
}

```

Fig. 11. A skeleton FC3 class.

must be reduced to normal form (rule 12). Reduction takes place in three stages. First, the pointers are copied from the slot fields to the `ev1` method's local variables, making access to them more efficient, and allowing the node to be *blackholed* by setting all of its slot fields to null. Black-holing avoids space leaks through nodes that will eventually be updated (Jones, 1992). In Figure 11 we show the copying as a comment rather than as three Java variable declarations with initialisation in order to ensure that the first slot field is copied into the first local variable, and so on. Java does not specify the mapping between variable declarations and local variables. Next, the node's function number is used to choose the appropriate code via a `switch` statement. The i th case of this `switch` is obtained by converting the $\langle \nu, G \rangle$ -machine instructions for the i th function of arity three to Java Virtual Machine instructions using the rules given in Figure 12. Each rule produces a list of instructions that are then concatenated. In reality, of course, our compiler uses the efficient versions of the Java Virtual Machine instructions mentioned, such as `aload_0` and `bipush`, whenever possible. Primitive operations are converted in the obvious way so, for example, integer addition uses `iadd`. Finally, the result

<code>ALOAD i</code>	\Rightarrow [<code>aload i</code>]
<code>ASTORE i</code>	\Rightarrow [<code>astore i</code>]
<code>CASE (l_1, l_2, \dots)</code>	\Rightarrow [<code>lookupswitch (l_1, l_2, \dots)</code>]
<code>DO n</code>	\Rightarrow [<code>invokevirtual N don, $(N_1 \dots N_n)N$</code>]
<code>EVAL</code>	\Rightarrow [<code>invokestatic RT UUO $(N)N$</code>]
<code>INITCAP $f_i^k m$</code>	\Rightarrow [<code>sipush i, invokespecial FCk <init> $(N_1 \dots N_m I)V$</code>]
<code>INITFRM f_i^0</code>	\Rightarrow []
<code>INITFRM f_i^k</code>	\Rightarrow [<code>sipush i, invokespecial FCk <init> $(N_1 \dots N_k I)V$</code>]
<code>INITVAL $k m$</code>	\Rightarrow [<code>sipush k, invokespecial Vm <init> $(N_1 \dots N_m I)V$</code>]
<code>NEWCAP f_i^k</code>	\Rightarrow [<code>new FCk, dup</code>]
<code>NEWFRM f_i^0</code>	\Rightarrow [<code>getstatic FCO $c_i N$</code>]
<code>NEWFRM f_i^k</code>	\Rightarrow [<code>new FCk, dup</code>]
<code>NEWVAL m</code>	\Rightarrow [<code>new Vm, dup</code>]
<code>NUM</code>	\Rightarrow [<code>invokevirtual N num $()I$</code>]
<code>POP</code>	\Rightarrow [<code>pop</code>]
<code>RET</code>	\Rightarrow [<code>astore 1</code>]
<code>SPLIT $n 0$</code>	\Rightarrow [<code>pop</code>]
<code>SPLIT $n m$</code>	\Rightarrow [<code>checkcast Vm,</code> <code>dup, getfield Vm s1 N, astore n,</code> <code>dup, getfield Vm s2 N, astore $(n + 1)$,</code> <code>...</code> <code>getfield Vm sm N, astore $(n + m - 1)$]</code>

Fig. 12. Instruction conversion rules.

of running the function's code is used to set the indirection field, and is returned (rule 13). This result is not necessarily in normal form. To get it there, the `ev1` method may need to be repeatedly invoked. The loop

```
public static N UUO(N x) { while (x != (x = x.ev1())); return x; }
```

in the main run-time system class `RT` does this, and is our "UUO handler". As

a detail, notice that the result of a function is stored in a local variable “result register”, rather than being left on the stack. This preserves an invariant of our compiler, which is that at any label in the Java Virtual Machine code, the stack depth is zero. Given this invariant, it is easy for the compiler to calculate the maximum stack usage of a method.

Together, the `do1`, `do2`, ... methods are used to convert the `DO n` instruction. Here, we depart from the state transition rules somewhat in order to save code space. As the `N` class shows, `do1`, `do2`, ... can be defined in terms of `do1`, and so this is all that a node need provide. The `do1` method tests for an indirection node, which must be bypassed (rule 17), and a frame node, which must be reduced to a canonical application node (rule 18). Once a canonical application node has been obtained, the `do1` method takes a copy of it, stores its pointer argument in the first empty argument slot of the node and alters the missing argument count, `z` (rules 20 and 21).

6.4 Constant Applicative Forms

Constant applicative forms (or CAFs) require the usual special treatment: a node must be allocated for each CAF; it must be updated when the CAF is first evaluated; and thereafter it must be used instead of re-evaluating the CAF (Peyton Jones, 1987). By way of example, Figure 13 sketches a skeleton `FC0` class produced for all non-primitive CAFs f_i^0 in some program, including any used from the standard prelude.

For each CAF there is a variable `ci` initialised to reference a frame node with the appropriate function number, i . All uses of the CAF refer to the same frame node via the variable `ci`. By convention, CAF 1 is the Haskell `main` function, and it is the frame node for this CAF that is evaluated when a Haskell program is run with the command “`java FC0`”.

7 Benchmarks

Using the ideas described above, we have constructed a compiler from Haskell to Java Virtual Machine code, based on version 990222 of the Hugs development environment. For benchmarking, we used a SUN UltraSPARC workstation with a 143MHz processor, 192Mbytes of memory and version 2.5.1 of the Solaris operating system.

The first three of our benchmarks are quite small: `calendars` (200 lines) is a program from (Bird & Wadler, 1988) used to format calendars for seven successive years; `clausify` (189 lines) is the program from (Runciman & Wakeling, 1993a) used to put a complex proposition into clausal form; `soda` (128 lines) is the program from (Runciman & Wakeling, 1993b) used to perform a word search in a 20×30 grid. The other five of our benchmarks are among the larger “real” ones in the Nofib suite (Partain, 1992): `bspt` (1,451 lines) exercises a geometric modelling program, making extensive use of arbitrary-precision integer arithmetic; `infer` (597 lines) uses a type checker written in the monadic style to infer the types of a number

```

public final class FCO extends N implements Cloneable {
    private int i; private N ind;
    public static final N c1 = new FCO(1);
    public static final N c2 = new FCO(2);
    ...

    public FCO(int i) { this.i = i; }

    public int num() { RT.Stop("FCO.num()"); return 0; }

    public N ev1() {
        if (this.ind != null) return this.ind.ev1();
        else {
            switch (this.i) {
                case 1: /* code for CAF number 1 */ break;
                case 2: /* code for CAF number 2 */ break;
                ...
            }
            /* this.ind = local 1; */ return this.ind;
        }
    }

    public N do1(N g1) { return RT.UU0(this).do1(g1); }

    public static void main(String[] argv) { RT.UU0(this.c1); }
}

```

Fig. 13. A skeleton FCO class.

of lambda terms; `parser` (1,355 lines) uses parser combinators to parse a large Haskell module; `prolog` (648 lines) interprets a Prolog program to solve the Towers of Hanoi problem with seven discs; and `reptile` (1,449 lines) exercises a graphical design program that performs extensive processing of character strings. All of the program sizes given here include comments and blank lines. Those for the first three programs include functions usually found in the standard prelude; those for the remaining five programs do not.

Five implementations of Haskell were compared:

- Ghc — the Glasgow Haskell compiler (version 4.01);
- Nhc — the York Haskell compiler (prerelease 2);
- Hugs — the Hugs Haskell development environment (version 990222);
- Int — our compiler/Sun interpreter (production version 1.2);
- JIT — our compiler/Sun JIT compiler (production version 1.2).

The Ghc and Nhc compilers were chosen as benchmarks because they represent two extremes: the first compiles to native code, optimising for speed at the expense of space (and the `-O2` flag was used to ensure that it held nothing back); the second compiles to a bytecode that is then interpreted, optimising for space at the expense of speed. Hugs also compiles to a code that is interpreted, and was chosen because it is used by many for program development. For all implementations, the maximum

heap size was fixed at 24Mbyte; for our compiler, the minimum heap size was also fixed at 24Mbyte.

7.1 Code Size

Table 3 compares code sizes in bytes. For both Ghc and Nhc, the sizes are those of dynamically-linked executables stripped of redundant symbol table information; for Hugs, no sizes are given because programs exist only within the development environment; for our compiler, the sizes are the sum of those of the program and run-time system “.class” files, the total number of which are also given.

Program	Ghc	Nhc	Hugs	Int or JIT
<code>calendars</code>	190,656	81,384	—	56,352 (37 classes)
<code>clausify</code>	184,788	58,080	—	56,173 (37 classes)
<code>soda</code>	193,928	58,224	—	60,195 (38 classes)
<code>bspt</code>	480,336	127,880	—	124,803 (39 classes)
<code>infer</code>	318,040	89,652	—	83,784 (38 classes)
<code>parser</code>	382,272	119,120	—	103,699 (41 classes)
<code>prolog</code>	266,068	75,744	—	71,738 (39 classes)
<code>reptile</code>	390,812	128,008	—	106,255 (40 classes)

Table 3. *Code sizes in bytes.*

These sizes all include code used from the standard prelude, but exclude that from the standard C or Java libraries. Those for Ghc include an 80kbyte run-time system; those for Nhc include a 52kbyte run-time system complete with a bytecode interpreter; those for our compiler include a 32kbyte run-time system (32 classes), but exclude either the bytecode interpreter or the “just-in-time” compiler. With this in mind, the most we can say is that the sizes of programs produced by our compiler are of the same order of magnitude as those produced by Nhc.

7.2 Execution Time

Table 4 compares execution times in seconds. For Ghc, Nhc and our compiler, timings were made for ten consecutive runs, the two best and the two worst were removed, and the average of the remainder recorded; for Hugs, the ten consecutive runs were made immediately after starting the interpreter. Previously, when we produced many hundreds of classes for the large benchmark programs, we gave the Java Virtual Machine interpreter or “just-in-time” compiler an initial “warm up” run to load the “.class” files from disk into memory (Wakeling, 1997; Wakeling, 1998b). Now that we produce far fewer classes, however, this is no longer necessary.

Typically, programs produced by our compiler run five times faster with the “just-in-time” compiler than with the interpreter, giving execution times of the same order of magnitude as those of Nhc or Hugs. They are slower with small benchmarks because the Java Virtual Machine must load a large number of standard classes

Program	Ghc	Nhc	Hugs	Int	JIT
<code>calendars</code>	0.1	1.0	7.3	19.8	3.5
<code>clausify</code>	0.2	2.3	11.2	35.4	5.9
<code>soda</code>	0.2	1.5	3.8	8.6	3.8
<code>bspt</code>	2.4	10.9	70.8	235.1	40.2
<code>infer</code>	1.7	18.6	35.5	164.2	28.7
<code>parser</code>	2.3	17.1	46.9	179.7	33.0
<code>prolog</code>	3.5	40.8	112.8	487.7	75.8
<code>reptile</code>	1.3	15.9	11.9	39.6	8.3

Table 4. *Execution times in seconds.*

before it can get going, and also with `bspt` because the Java library for arbitrary-precision integer arithmetic is not as efficient as the GNU one used by Nhc. Overall, Hugs suffers because its mark-sweep garbage collector is not as efficient as those of the other implementations in a 24Mbyte heap.

8 Optimisations

Since our compiler is based on the Hugs interpreter, no significant optimisations, such as strictness analysis, are performed. However, we have tried a few simple optimisations, aimed at reducing code size and increasing execution speed.

8.1 Overriding the `don` Methods

The implementations of the `do2`, `do3`, ... methods in the `N` class are simple, but inefficient. Using `don` to apply a function to n arguments creates n intermediate nodes representing the application of the function to 1, 2, ..., n arguments. Given the high cost of memory allocation in the Java Virtual Machine, we would like to create just one node representing the application of the function to all n arguments. To do this, we *override* the inefficient methods in the `N` class with efficient ones in the `FCk` subclasses. By way of example, Figure 14 shows an efficient `do2` method for the `FC3` class.

Of course, providing the full complement of `do2`, `do3` ... methods in every `FCk` class would make Java Virtual Machine code programs much larger. Profiling our benchmarks, however, reveals that almost all of the benefit can be obtained by providing just `do2`, which accounts for more than 99% of all `do2`, `do3` ... invocations. Table 5 shows the effect of overriding the `do2` method in every `FCk` class (here, a ‘-’ indicates a reduction, and a ‘+’ an increase). The figures reflect the large number of applications of binary functions selected from dictionaries in unoptimised Haskell programs.

8.2 Boxed Values

Functional language implementations often make a few common constant values part of the run-time system to avoid allocating them many times. It is easy for

```

public N do2(N g1, N g2) {
    if (this.ind != null) return this.ind.do1(g1).do1(g2);
    else if (this.z == 0) return RT.UU0(this).do1(g1).do1(g2);
    else {
        try { FC3 f = (FC3) this.clone();
            switch (f.z) {
                case 1 : f.s3 = g1; f.z = 0; return f.do1(g2);
                case 2 : f.s2 = g1; f.s3 = g2; f.z = 0; return f;
                case 3 : f.s1 = g1; f.s2 = g2; f.z = 1; return f;
            }
        } catch (CloneNotSupportedException e) { RT.Stop("FC3.do2()"); }
        return null; /* not reached */
    }
}

```

Fig. 14. The do2 method for the FC3 class.

Program	Code size (bytes)		Int time (secs)		JIT time (secs)	
calendars	56,772	(+0.7%)	19.0	(-4.0%)	3.1	(-11.4%)
clausify	56,593	(+0.8%)	33.3	(-5.9%)	4.9	(-17.0%)
soda	61,051	(+1.4%)	8.2	(-4.7%)	3.4	(-10.5%)
bspt	125,607	(+0.6%)	221.4	(-5.8%)	35.2	(-12.4%)
infer	84,375	(+0.7%)	147.1	(-10.4%)	21.4	(-25.4%)
parser	104,882	(+1.1%)	165.4	(-8.0%)	28.0	(-15.2%)
prolog	72,481	(+1.0%)	457.7	(-6.2%)	62.8	(-17.1%)
reptile	107,201	(+0.9%)	39.5	(-0.3%)	8.2	(-1.2%)

Table 5. The effect of the do2 optimisation.

us to do this too. Instead of generating code to allocate a constructed value node with a small constructor number (in our case, between zero and ten) and without arguments, we generate code to use one built in to the run-time. So, for example, rather than allocate a constructed value node with a constructor number 0 and without arguments, the compiler generates code to use the variable `b0`, initialised as

```
public static final N b0 = new V0(0);
```

in the run-time system class `RT`. Table 6 shows the further effect of this optimisation. The figures show that for short-lived boxed values, the cost of referencing a global field in another class can outweigh the saving from not allocating it.

8.3 Tail Recursion

As we have already explained, tail recursion is dealt with by returning a frame node representing the next function application, instead of performing that application directly. However, it is possible to do better when the next function application has the same arity as the current one. All one need do is put the arguments to the

Program	Code size (bytes)	Int time (secs)	JIT time (secs)
calendars	56,624 (-0.3%)	17.7 (-6.8%)	3.0 (-3.2%)
clausify	56,541 (0.0%)	32.3 (-3.0%)	4.9 (0.0%)
soda	60,975 (-0.1%)	7.6 (-7.3%)	3.4 (0.0%)
bspt	124,348 (-1.0%)	211.3 (-4.6%)	33.6 (-4.8%)
infer	83,846 (-0.6%)	149.1 (+1.4%)	21.7 (+1.4%)
parser	103,898 (-0.9%)	165.6 (+0.1%)	26.7 (-4.6%)
prolog	72,083 (-0.6%)	428.0 (-6.5%)	62.9 (+0.2%)
reptile	106,156 (-1.0%)	40.2 (+1.8%)	9.0 (+9.8%)

Table 6. *The effect of the boxed value optimisation.*

next function application into the appropriate local variables, and then use a `goto` instruction to transfer control to the function's code. Table 7 shows the further effect of this optimisation.

Program	Code size (bytes)	Int time (secs)	JIT time (secs)
calendars	56,509 (-0.2%)	17.0 (-4.0%)	2.9 (-3.3%)
clausify	56,374 (-0.3%)	31.1 (-3.7%)	4.6 (-6.1%)
soda	60,955 (-0.0%)	6.7 (-7.9%)	2.8 (-17.6%)
bspt	123,997 (-0.3%)	205.0 (-3.0%)	37.2 (+10.7%)
infer	83,080 (-0.9%)	144.0 (-3.4%)	18.9 (-12.9%)
parser	103,456 (-0.4%)	186.2 (+12.4%)	26.9 (+0.8%)
prolog	71,891 (-0.3%)	429.0 (+0.2%)	62.5 (-0.6%)
reptile	105,949 (-0.2%)	37.8 (-6.0%)	7.6 (-15.6%)

Table 7. *The effect of the tail recursion optimisation.*

For small benchmarks designed to make extensive use of tail recursion, the optimisation described usually makes programs run 10% faster. For our benchmarks, however, it is uncertain whether execution time will get better or worse, even though around 40% of tail recursive calls are optimised. At the moment, we do not know why this is. Originally, we thought that it was because of a space leak. As soon as the `ev1` method in class `Fck` is invoked, the Java Virtual Machine allocates enough local variable and stack space for any function of arity k . If the method returns, then everything reachable only from this local variable and stack space can be recovered; if it simply branches, then everything reachable from this local variable and (perhaps) stack space is preserved. However, setting unused local variables to `null` before a tail recursive call makes little difference, and we are continuing to investigate. The original reason for generating Java Virtual Code rather than Java was to implement tail-recursion efficiently using the `goto` that Java forbids. But this problem means that the considerable additional effort is not currently worthwhile.

9 Other Java Virtual Machine Implementations

To illustrate the variability among current Java Virtual Machine implementations, we have also run the Java Virtual Machine code produced by the final version of our compiler on a Gateway 2000 Solo notebook computer with a 166MHz Pentium II MMX processor, 48Mbytes of memory and version 4.0 of the Windows NT Workstation operating system (build 1381, SP 3).

Seven implementations of Haskell were compared:

- Ghc — the Glasgow Haskell compiler (version 4.02);
- Nhc — the York Haskell compiler (prerelease 2);
- Hugs — the Hugs Haskell development environment (version 990222);
- Sun — our compiler/Sun interpreter (version 1.2-V);
- Symantec — our compiler/Symantec JIT compiler (version 1.2-V);
- Jview — our compiler/Microsoft JIT compiler (version 5.00.3167);
- Kaffe — our compiler/Transvirtual JIT compiler (version 1.00 beta 3).

Ghc, Nhc and Hugs all run with version 20.1 beta of the Cygwin32 dynamically-linked library (Noer, 1998). The Sun JDK for Windows NT includes both their own interpreter and the Symantec “just-in-time” compiler. For all but Jview, we were again able to set both the minimum and maximum heap size to 24Mbytes. Table 8 compares execution times in seconds, obtained using the same methodology as previously.

Program	Ghc	Nhc	Hugs	Sun	Symantec	Jview	Kaffe
<code>calendars</code>	0.3	0.8	3.3	8.3	3.6	7.8	30.8
<code>clausify</code>	0.3	1.7	5.1	14.7	5.8	13.5	46.4
<code>soda</code>	0.3	0.5	1.8	5.6	2.1	2.1	5.1
<code>bspt</code>	1.8	8.3	30.3	100.7	45.0	537.5	^a
<code>infer</code>	1.6	14.3	15.9	82.0	34.7	511.7	318.0
<code>parser</code>	1.8	13.7	21.8	86.8	42.7	253.0	382.5
<code>prolog</code>	2.5	32.4	50.9	229.4	103.3	^b	^c
<code>reptile</code>	1.2	4.0	5.5	24.4	16.6	33.2	128.2

^a fails because of a spurious verification error

^b fails because of a stack overflow

^c fails because of an unimplemented bignum initialiser

Table 8. *Execution times in seconds.*

10 Other Compilers

Out of interest, we have translated the `calendars`, `clausify` and `soda` benchmarks from Haskell to Standard ML, and rewritten them in Java, so that we can compare three compilers to Java Virtual Machine code, all running on our UltraSPARC:

- Ours — our compiler (final version);
- MLJ — the Persimmon IT Standard ML compiler (version 0.1);
- Javac — the Sun Java compiler (SDK version 1.2).

Table 9 compares code sizes in bytes, and Table 10 compares execution times in seconds with the Sun Java Virtual Machine interpreter and “just-in-time” compiler (both production version 1.2). The figures show that on these small benchmarks our compiler for a lazy functional language is not competitive with one for a strict functional language or an object-oriented one.

Program	Ours		MLJ		Javac	
<code>calendars</code>	56,509	(37 classes)	11,881	(21 classes)	6,145	(4 classes)
<code>clausify</code>	56,374	(37 classes)	12,469	(21 classes)	8,564	(10 classes)
<code>soda</code>	60,955	(38 classes)	15,579	(19 classes)	5,712	(5 classes)

Table 9. *Code sizes in bytes.*

Program	Ours		MLJ		Javac	
	Int	JIT	Int	JIT	Int	JIT
<code>calendars</code>	17.0	2.9	1.4	0.2	0.6	0.1
<code>clausify</code>	31.1	4.6	2.5	0.5	2.4	0.4
<code>soda</code>	6.7	2.8	0.7	0.3	1.6	0.5

Table 10. *Execution times in seconds.*

11 Related Work

In two previous papers, we have described Haskell to Java Virtual Machine code compilers based on the G-machine (Wakeling, 1997) and the $\langle \nu, G \rangle$ -machine (Wakeling, 1998b). The first of these generated one class file for each function, the second generated two, leading to programs that were large and slow. As mentioned earlier, the G-machine based compiler had another drawback — the Java array used for the reduction stack was a source of inefficiency because of the high cost of bounds-checked array access, and because of space leaks caused by the Java Virtual Machine’s garbage collector not knowing that it was being used as a stack. Looking for way to avoid space leaks by splitting the single large array into a linked-list of many small arrays, we soon arrived at an abstract machine remarkably like the $\langle \nu, G \rangle$ -machine. But even armed with the equation $\nu = \text{this}$, it proved hard to get the details right. Our first attempt had clumsy implementations of both the “UO handler” and the DO instruction, and it was only much later that we realised that integer function numbers and a large `switch` statement could eliminate the need for

more than a few class files per functional program, without a significant efficiency penalty.

A number of other researchers have considered compiling lazy functional languages to Java Virtual Machine code. Prototype Haskell compilers based on the Spineless Tagless G-machine have been produced by two students: Tullsen at Yale University (Tullsen, 1997), and Vernet at the Swiss Federal Institute of Technology (Vernet, 1998). From the Spineless Tagless G-machine’s “every node is a closure” design, both created “every closure is a class” implementations. Classes have one method for constructing a closure, with the arguments taken from the Java Virtual Machine’s stack, and another for entering it in order to reduce it to normal form. For higher-order functions, Tullsen uses an `apply` method like our `do` methods, creating a closure for each application to an argument; from his paper, it is unclear what Vernet does. Neither student considers the problem of implementing tail-recursion properly and, unsurprisingly, neither appears to have run more than one or two very small test programs.

Meehan and Joy at Warwick have produced a compiler based on the G-machine for their lazy language, Ginger (Meehan & Joy, 1998). The novelty of their work lies in the implementation of functions. They avoid generating a class file for each by using the Java reflection package to access the methods of a single class, and to apply them to arrays of arguments. In our experience, however, this is a rather poor choice. An early version of our first (ν, G) -machine based compiler used a similar technique, and we found that invoking methods via the reflection package was much slower than doing so directly (our measurements suggested about 1/3rd of the speed). Moreover, according to Sun, the reflection package is intended for use by applications such as debuggers, interpreters and class browsers, so improving its performance is unlikely to be a priority. The need for arrays of arguments in addition to graph nodes, of course, also leads to increased storage allocation. Meehan and Joy’s benchmark figures confirm our fears — with an UltraSPARC and Sun’s JDK 1.1.5, programs produced by their compiler run 5 — 10 times slower than those run with Hugs.

There has also been work on compiling strict functional languages for the Java Virtual Machine. As we have already mentioned, Benton, Kennedy and Russell at Persimmon IT, Inc. have developed MLJ, a compiler from Standard ML to Java Virtual Machine code (Benton *et al.*, 1998). Their compiler performs extensive optimisation of the whole program in order to generate compact code with reasonable performance. Having sight of the whole program is a big advantage. It allows, for example, the removal of polymorphism by specialisation, the avoidance of argument tuple allocation by uncurrying, and the compilation of functions according to how they are used. However, their compiler does not handle general tail recursion properly. With a Java Virtual Machine that performs “just-in-time” compilation, benchmark programs usually run more quickly than with version 1.42 of the Moscow ML interpreter; numerical ones run even more quickly than with version 110 of the SML/NJ compiler.

Per Bothner at Cygnus Solutions has developed a Scheme implementation, complete with the usual read-eval-print loop, that generates Java Virtual Machine code (Bothner, 1998). On the one hand, there are clear similarities here between

his class `Procedure` with its `apply` methods, and our `N` class with its `do` methods. But on the other hand, there are also clear differences between his planned implementation of general tail recursion using a variant of the “UO handler” loop, and our own. For certain procedures, Bothner proposes to use a continuation passing style, and to model traditional frame pointer and program counter registers using Java variables. It is not yet clear how his idea will work out in practice, or if it would be of any use to us. Even without it though, his implementation with Sun’s JDK 1.1.5 can run benchmark programs at more than half the speed of those run with the Scheme48 interpreter.

12 Future Work

An obvious way to continue with this work would be in the direction that Benton, Kennedy and Russell (Benton *et al.*, 1998) have taken with Standard ML, improving performance through extensive optimisation, and providing Haskell with an interface to Java. The first of these requires a re-implementation of our ideas in a more suitable framework than Hugs, and the Glasgow Haskell compiler would seem to be a good choice. The second of these requires us to invent a syntax for Haskell programmers to describe classes, create objects and invoke methods, preferably without forcing them to write Java programs, which is essentially what Benton, Kennedy and Russell do.

13 Conclusions

In this paper we have shown how lazy functional programs can be compiled for the Java Virtual Machine using a mapping between a version of the $\langle \nu, G \rangle$ -machine and the Java Virtual Machine. This is not the obvious thing to do, but it turns out to be both elegant and efficient. The description is entirely straightforward, and both the code size and execution speed of Haskell programs compiled using it are of the same order of magnitude as those obtained with a traditional functional language bytecode interpreter. In future, our work could serve as the basis of an interface between Haskell and Java.

Acknowledgements

As well as inventing the $\langle \nu, G \rangle$ -machine, Lennart Augustsson and Thomas Johnson distribute the Chalmers HBC compiler on which earlier versions of this work were based. Mark Jones distributes the Hugs development environment on which the current version is based. Malcolm Wallace kindly let us have a prerelease version of Nhc compiler. Our thanks to these researchers for making their implementations freely available. Our thanks also to the referees, whose detailed comments on earlier versions of this paper enabled us to improve it considerably.

This work was wholly supported by Canon Research Centre Europe Limited.

References

- Augustsson, L., & Johnsson, T. (1989). Parallel graph reduction with the $\langle \nu, G \rangle$ -machine. *Pages 202–213 of: Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*. ACM Press.
- Benton, N., Kennedy, A., & Russell, G. 1998 (September). Compiling Standard ML to Java bytecodes. *Pages 129–140 of: Proceedings of the 1998 International Conference on Functional Programming*. SIGPLAN Notices 34(1), January 1999.
- Bird, R., & Wadler, P. (1988). *Introduction to functional programming*. Prentice-Hall.
- Bothner, P. 1998 (November). Kawa: Compiling Scheme to Java. *Proceedings of the 1998 Lisp User's Conference ("Lisp in the Mainstream"/"40th Anniversary of Lisp")*.
- Jones, R. (1992). Tail recursion without space leaks. *Journal of Functional Programming*, 2(1), 73–79.
- Lindholm, T., & Yellin, F. (1999). *The Java Virtual Machine specification, second edition*. Addison-Wesley.
- Meehan, G., & Joy, M. 1998 (May). *Compiling lazy functional programs to Java bytecode*. Tech. rept. Department of Computer Science, University of Warwick. submitted for publication.
- Noer, G. J. (1998). Cygwin32: A free Win32 porting layer for UNIX applications. *Proceedings of the Second USENIX Windows NT Symposium*. USENIX.
- O'Connor, M. J., & Tremblay, M. (1997). PicoJava: The Java Virtual Machine in hardware. *IEEE micro*, 17(2), 45–53.
- Partain, W. (1992). The nofib benchmark suite of Haskell programs. *Pages 195–202 of: Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer-Verlag.
- Peyton Jones, S. L. (1987). *The implementation of functional programming languages*. Prentice-Hall.
- Runciman, C., & Wakeling, D. (1993a). Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2), 217–245.
- Runciman, C., & Wakeling, D. (1993b). Profiling parallel functional computations (without parallel machines). *Pages 236–251 of: Proceedings of the 1993 Glasgow Workshop on Functional Programming*. Springer-Verlag.
- Steele, G. L. (1978). *Rabbit: A compiler for scheme*. Tech. rept. AI-TR-474. MIT Laboratory for Computer Science.
- Tullsen, M. 1997 (September). *690 project: Compiling Haskell to Java*.
- Vernet, A. 1998 (February). *The Haskell project*.
- Wakeling, D. (1997). A Haskell to Java Virtual Machine code compiler. *Pages 39–52 of: Proceedings of the 1997 Workshop on the Implementation of Functional Languages*. Springer-Verlag. LNCS 1467.
- Wakeling, D. (1998a). The dynamic compilation of lazy functional programs. *Journal of Functional Programming*, 8(1), 1–21.
- Wakeling, D. (1998b). Mobile Haskell: Compiling lazy functional programs for the Java Virtual Machine. *Pages 335–352 of: Proceedings of the 1998 Conference on Programming Languages, Implementations, Logics and Programs (PLILP'98)*. Springer Verlag. LNCS 1490.