SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(7), 617–645 (1999)

Compiling Lazy Functional Programs to Java Bytecode

GARY MEEHAN*AND MIKE JOY
Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
(email: {gary.meehan, m.s.joy}@dcs.warwick.ac.uk)

SUMMARY

The Java Virtual Machine (JVM) was designed as the target for Java compilers, but there is no reason why it cannot be used as the target for other languages. We describe the implementation of a compiler which translates a lazy, weakly-typed functional program into Java class files. We compare the performance of our compiler to the only other known compiler from a lazy functional language to the JVM. The results are broadly similar, suggesting that to get a significant performance speed-up using this compilation paradigm will come only from increasing the performance of the JVM, rather than enhancing the compiler itself.

KEY WORDS: lazy functional languages; compiling; Java Virtual Machine

INTRODUCTION

The Java Virtual Machine (JVM) [1] provides a machine-independent execution environment which executes Java *bytecode*, which is essentially a machine code for object-oriented programs. It was designed as the target of Java compilers, but there is no reason why compilers of other languages cannot target it. We are interested in using it as the target code of a functional language, in particular a pure, lazy one. This approach has several advantages:

(a) Java bytecode will run on any machine for which an interpreter is available.
(b) Java programs can be run in web-browsers as applets or in embedded systems.
(c) Java has a built-in garbage-collector, hence any language which targets Java bytecode has no need to handle garbage collection itself.

Our aim is to create a compiler which will translate a functional program, with each function being translated into a static method of the generated class file. If the functional program we are compiling is designed to be executed (as opposed to being a set of library functions, for example) then we also generate a `main` method which will, when the class file is executed, perform any initialisation necessary and evaluate the program which we have compiled. Our source language is Ginger [2], a simple, pure, lazy, weakly-typed functional language. We base our evaluation methods on those of the G-machine [3,4].

We assume that the reader is familiar with programming in Java, and has some understanding of how Java classes are structured, and also how to program using a lazy functional language, but we assume no prior knowledge of the JVM itself or of implementing functional languages. The following two sections give brief guides to the JVM and the

*Correspondence to: G. Meehan, Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK.

CCC 0038–0644/99/070617–29$17.50

Received 19 May 1998
Revised 12 January 1999
Accepted 18 January 1999

evaluation of functional programs using graph reduction. We then discuss the design of the run-time environment of our compiler and how we compile Ginger programs. We give some results of running programs compiled with our compiler, comparing them with Nottingham and Yale Haskell interpreter Hugs [5], the Glasgow Haskell Compiler [6] and another lazy functional program compiler, developed by David Wakeling at Exeter, which targets the JVM [8,7], and finally conclude.

THE JAVA VIRTUAL MACHINE

The JVM [1] provides instructions for implementing object-oriented programming languages; that is, for creating objects, invoking their methods, manipulating their fields, as well as the usual basic operations, such as adding integers.

The format of the bytecode resembles that of Java programs [9]. For each new class, there is a header declaring the class, its superclass and its package and the declaration of the fields and methods. Each method provides a separate environment consisting of a stack, which is used as a working space, and a set of local variables. In the case of an instance method, register 0 holds a reference to the object that the method was invoked on (i.e. the `this` reference), and variables $1, \ldots, n$ hold the n parameters of the method. Static methods are slightly different, in that since there is no object to invoke the method on, the n parameters of the method are stored in variables $0, \ldots, n-1$. Each class also has a *constant pool*, where all the symbolic data used by the class – fields, methods, class, interfaces, etc. – is stored. For example, consider the following class definition:

```
public class ExampleClass {
  public int value;
  private static ExampleClass one = new ExampleClass(1);

  public ExampleClass(int v) {
    value = v;
  }

  public static ExampleClass getOne() {
    return one;
  }

  public void add(ExampleClass e) {
    value += e.value;
  }
}
```

This class can be compiled into a class file of bytecodes using a Java compiler, `javac`, say. The class file can then be examined using a disassembler, such as `javap` [10]. First, we have the header of the file which declares the class's super-class and its members:

```
Compiled from ExampleClass.java
public synchronized class ExampleClass extends java.lang.Object
    /* ACC_SUPER bit set */
{
    public int value;
```



```
    private static ExampleClass one;
    public ExampleClass(int);
    public static ExampleClass getOne();
    public void add(ExampleClass);
    static static {};
}
```

We then have the bytecode for the constructor function:

```
Method ExampleClass(int)
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 aload_0
   5 iload_1
   6 putfield #6 <Field int value>
   9 return
```

A reference to the object that is created by the constructor is held in register 0. This is placed on the stack by the `aload_0` instruction. The `invokespecial #3` pops the top object off the stack (i.e. `this`) and invokes on it the zero-argument constructor of its superclass, i.e. `Object`, which is item number 3 in the constant pool. When this has completed, `this` is again loaded onto the stack and then the integer 1 is loaded onto the stack by the `iload_1` instruction. The `putfield #6` instruction pops the top two values of the stack and stores the value that was held at the top of the stack (the integer 1) in the `value` field (item number 6 in the constant pool) of the object that was the second topmost item in the stack (`this`). The work is now done, the stack is empty, and the constructor function returns to the method that called it, with a `void` result, using the `return` instruction.

We then have the two method declarations. The method `getOne` merely loads the static field `one` onto the stack and returns it using the `areturn` instruction:

```
Method ExampleClass getOne()
   0 getstatic #5 <Field ExampleClass one>
   3 areturn
```

The `add` method is a little more complicated:

```
Method void add(ExampleClass)
   0 aload_0
   1 dup
   2 getfield #6 <Field int value>
   5 aload_1
   6 getfield #6 <Field int value>
   9 iadd
  10 putfield #6 <Field int value>
  13 return
```

This first loads the `this` reference onto the stack and duplicates it using the `dup` instruction, leaving the copy on the top of the stack. The `getfield #6` instruction pops the top of the stack and gets the value of its `value` field which it puts on top of the stack. We then load the argument of the method (named `e` in the original code) which is in register 1, and get the

value of its `value` field. The top two items on the stack are now the two integers equal to the value fields of the object the method was invoked on, and the argument of the method. These integers are popped off the stack and added together using the `iadd` instruction, which places the result on the stack. This value is then stored in the `value` field of the object referenced by `this` – remember the `dup` instruction – and the method returns to its caller, returning a `void` value.

The final method is the class's static initialiser which is executed when the class is loaded. This has the job of creating a new `ExampleClass` whose value field is set to 1 and storing it in the static field `one`:

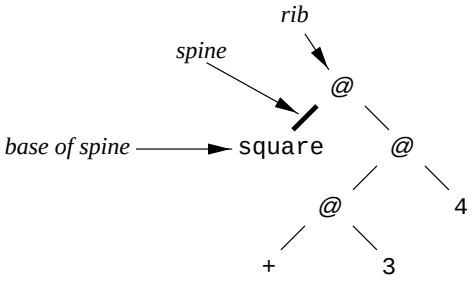
```
Method static {}
   0 new #1 <Class ExampleClass>
   3 dup
   4 iconst_1
   5 invokespecial #4 <Method ExampleClass(int)>
   8 putstatic #5 <Field ExampleClass one>
  11 return
```

GRAPH REDUCTION

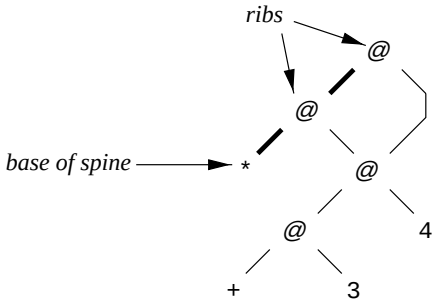
Graph reduction provides a way to implement functional languages [3,4,11]. The evaluation of an expression involves the application of reduction rules, i.e. the definition of functions, until no more reduction rules are applicable, at which point the expression is said to be in *normal form*. For instance, suppose we have the definition:

```
square x = x * x;
```

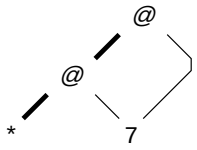
and we want to evaluate the expression `square (3 + 4)`. This is stored as the graph:



Note that we represent infix operators like + (which come between their arguments) as prefix ones (which come before their arguments). Since values in pure functional languages are immutable, we can share sub-expressions saving both time and memory. We can evaluate the expression by *reducing* the graph [4,12]. For example, if we reduce `square (3 + 4)` we get the graph:



The first time `3 + 4` is evaluated the graph reduces to:



Hence we only need to evaluate `3 + 4` *once* as the result is shared between the nodes that point to it. Reduction of the multiplication yields the answer `49` which is the normal form of `square (3 + 4)`.

Usually when we are evaluating an expression, there is more than one subexpression that can be reduced (each such subexpression is called a `redex`). Implementations of lazy functional languages choose to evaluate the outermost redex first; that is, apply the reduction rule of the function at the base of the spine of the graph. So we evaluate the function itself before its arguments. This is known as *lazy evaluation* or *outermost graph reduction*. For instance, we reduce `square (3 + 4)` to `(3 + 4) * (3 + 4)` rather than `square 7` (the latter reduction would be done by strict languages, and is known as *strict evaluation* or *innermost graph reduction*) . Of course, eventually we may need to evaluate the arguments of a function, but such an operation is only usually done explicitly by primitives of the language, such as `*` in our `square` example.

Lazy and strict evaluation will always reduce an expression to the same normal form, however evaluating an expression using strict evaluation may not terminate in some cases where lazy evaluation does. For example, consider the small Ginger program:

```
from x = x : from (x + 1);

main = hd (from 0);
```

Here `:` (cons) is the list constructor and `hd` selects the head of the list, i.e. the expression to the left of the `:`. Note that the expression *h* `:` *t* is in normal form no matter what *h* and *t* are (i.e. cons does not evaluate its arguments). The expression `from x` yields the list `[x, x + 1, x + 2,...]`, and hence evaluating the expression `hd (from 0)` should give the answer `0`. With lazy evaluation, we have the reduction sequence:

```
main ⟹ hd (from 0)
     ⟹ hd (0 : from (0 + 1))
     ⟹ 0
```


⟨*program*⟩ ::= (`package` ⟨*package*⟩`;`)?
(`import` ⟨*class*⟩`;`)∗
⟨*definition*⟩∗

⟨*definition*⟩ ::= ⟨*identifier*⟩ ⟨*identifier*⟩∗ `=` ⟨*expr*⟩`;`

⟨*expr*⟩ ::= ⟨*integer*⟩ | ⟨*boolean*⟩ | ⟨*float*⟩ | ⟨*character*⟩ | ⟨*string*⟩
| ⟨*identifier*⟩
| `[]` (empty list)
| (⟨*expr*⟩`:`⟨*expr*⟩) (list constructor)
| (⟨*expr*⟩ ⟨*expr*⟩) (application)
| `(`⟨*expr*⟩`,` . . . `,`⟨*expr*⟩`)`
| `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩ `endif`
| `let` ⟨*identifier*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩ `endlet`
| `letrec`
⟨*identifier*⟩ `=` ⟨*expr*⟩
. . .
⟨*identifier*⟩ `=` ⟨*expr*⟩
`in` ⟨*expr*⟩ `endletrec`

Figure 1. The EBNF of the Ginger language after lambda-lifting and dependency analysis

Like `*`, `hd` evaluates its argument (to normal form). The result of the evaluation should be a cons, otherwise we have an error, at which point `hd` selects the leftmost argument of the cons. Now suppose we evaluate the expression strictly:

```
main ⟹ hd (from 0)
     ⟹ hd (0 : from (0 + 1))
     ⟹ hd (0 : from 1)
     ⟹ hd (0 : 1 : from (1 + 1))
     ⟹ hd (0 : 1 : from 2)
     ⟹ hd (0 : 1 : 2 : from (2 + 1))
     ...
```

The recursion would never terminate as the innermost expression can always be reduced.

THE GINGER LANGUAGE

The Ginger language [2] was developed at the University of Warwick as a means of investigating parallelism in functional languages (we don't consider the parallel features of

the language in this paper). It is a simple, weakly-typed, pure lazy functional language with no pattern-matching or user-defined types.

After parsing, the Ginger compiler transforms the source tree into a set of supercombinators (functions in which no free variables occur, see Section 13.2.1 of Peyton Jones (1987) [4]) by lifting any lambda abstractions into separate function definitions [13]. Then dependency analysis [4] is performed, which transforms local variable definitions into blocks of simple, non-recursive `let` expressions and blocks of minimally mutually-recursive `letrec` blocks. The source at this stage is structured as in Figure 1.

We have an optional `package` declaration in which the class we eventually create will be placed. Then come a number of `import` declarations which deal with the importing of Ginger functions (stored in Java classes) from other sources followed by the definitions of the supercombinators, both those defined by the user and those created by lambda-lifting.

REPRESENTATION OF GRAPH NODES

Since we are creating a Java class file, it makes sense to represent graph nodes by Java objects. The five simple types – integers, floats, booleans, characters and strings – are represented using the Java classes `Long`, `Double`, `Character`, `Boolean` and `String` found in the `java.lang` package. The first four are the object equivalent of the primitive Java types `long`, `double`, `char` and `boolean` respectively. Note that we use the largest size possible for integers and floats, and we do *not* represent strings as list of characters.

Lists are represented using the `List` class, which is just an empty class which its two subclasses `Cons` (list constructor) and `EmptyList` subclass. The `Cons` has the skeleton following definition:

```
public final class Cons extends List {
  public Object head;
  public Object tail;

  // ...
}
```

The representation of functions utilises the `java.lang.reflect` package, which provides classes that 'reflect' the members of the class, in particular there is a `Method` class the instances of which reflect a particular method of a particular class. This class has an instance method `invoke`:

```
public Object invoke(Object obj, Object[] args)
```

This invokes the method reflected by the instance on the object `obj` with the arguments in the array `args`. If the method being reflected is static then `obj` is ignored and can be `null`.

Our compiler will translate all the supercombinators in a file into static methods of the generated class file (see below). This gives us a maximum of 255 arguments and local variables per function. We use static methods as these are more like ordinary functions than instance methods, who expect at least one argument which is an instance of the class the

method is defined in. When we import a class file we thus store each of its methods (functions of the original program) as `Method` objects which are held inside `Func` objects:

```
public class Func {
  public final Method method;
  public final int arity;
  public final boolean isCAF;

  public Func(Method m) {
    method = m;
    arity = m.getParameterTypes().length;
    isCAF = arity == 0;
  }

  // ...
}
```

We could represent functions directly as `Method` objects, but we commonly needed to determine the arity of a function and whether it is a *Constant Applicative Form* (CAF), that is a function of arity 0. The only way to do this provided by the `Method` class is to use the `getParameterTypes` method, which returns an array representing the types of the method's parameters and determining its length. By storing the `Method` object inside a `Func` class, we only have to do this operation once, when the instance is created. Note that this doesn't add too much as an overhead, as we only create *one* instance of each function which is then shared, rather than creating a new function each time we come across one (see below).

Our representation of applications is guided by the type of the `Method.invoke` method which will be called every time we apply a function. Since this method expects its arguments to be packed into an array, it makes sense to store the arguments of an application in an array, rather than in some intermediate data structure from which we make an array. In particular, we use multiple-argument applications. The `App` class has the following definition:

```
public final class App {
  public Object functor;
  public Object[] args;

  public boolean in_nf = false;
  public boolean total_app = false;

  // ...
}
```

This represents the application `functor args[0] ... args[args.length - 1]`. The field `in_nf` is set when the `App` is in normal form, that is when the `functor` is a function and there are not enough arguments present, or the `App` is acting as an indirection to a non-application (see below). If `functor` is a function and it is applied to exactly the right number of arguments, then we set the `total_app` field. The use of the `in_nf` and `total_app` fields prevents unnecessary work being done.

Updating

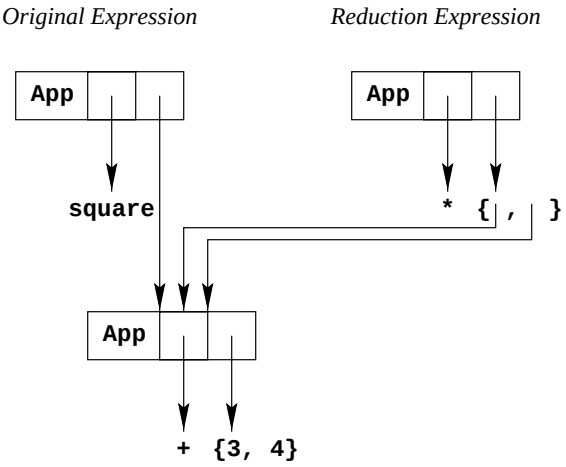
As we saw in the section on graph reduction, after applying a function to its arguments, we need to update the original application with the result of the application. This is to prevent the unnecessary re-evaluation. For instance, the application `square ((+) 3 4)` becomes `(*) ((+) 3 4) ((+) 3 4)` (with the instances of `(+) 3 4` being shared) and `(+) 3 4` becomes 7.

In the case where we update one application with another, we simply copy the result field-by-field onto the original application. However, if the result *isn't* an application, as in the second case, then things aren't so simple as we can't copy an object of one type onto an object of a different type. Instead, we turn the `App` into an indirection by setting its `args` field to the `null` reference, and setting its `functor` field to the result in question. We can view any `App` with a `null args` field as serving as an indirection or a wrapper to its actual argument. Note that, once the result of an application becomes a non-application, it is in normal form, and thus no further evaluation is necessary and so we do not get chains of indirections. Updating is done in the method `App.update`:

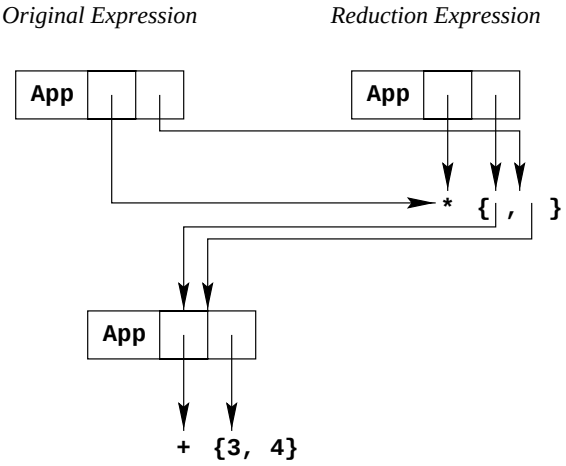
```
private void update(Object o) {
  if (o instanceof App) {
    // copy o onto this App
    App a = (App) o;
    functor = a.functor; args = a.args;
    in_nf = a.in_nf; total_app = a.total_app;
  }

  else if (o instanceof Func) {
    functor = o; args = empty;
    if (((Func) o).isCAF) {
      total_app = true; in_nf = false;
    }
    else {
      total_app = false; in_nf = true;
    }
  }

  else { // we have an indirection
    functor = o; args = null;
    in_nf = true; total_app = false;
  }
}
```

Here `empty` is a field of `App` which is set to be an empty array of `Object`s.

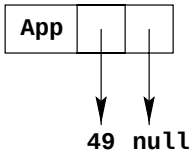
Going back to our original example, `square ((+) 3 4)` reduces to `(*) ((+) 3 4) ((+) 3 4)`. Before updating we have:



Updating the original expression involves reassigning its `functor` and `args` fields (ignoring the other two `boolean` fields for the moment):



After the reduction of `(*) ((+) 3 4) ((+) 3 4)` our `App` becomes an indirection to `49`:



Applications are not the only graph nodes that can be updated, there is one other case, namely CAFs, that is functions taking zero arguments. These can be treated as applications of the CAF in question to zero arguments, and we do just this by storing all CAFs as `App`s whose `functor` is the actual CAF in question and whose `args` is the empty array (note that if we wish to invoke a method that takes no arguments then we need to pass an empty array to `Method.invoke`).

Evaluation

The evaluation of graph nodes is controlled by the method `eval` in the class `Node` which contains various static methods used for the evaluation and printing of graph nodes. The method only has to do something when it is called to evaluate an `App`, otherwise it simply returns its argument.

The instance method `App.eval` repeatedly evaluates and updates itself until it is in normal form. Once it becomes so, it returns its `functor` if it is an indirection, or itself otherwise:

```
public Object eval() {
  while (! in_nf) {
    if (total_app)
      // ...
    else if (functor instanceof Func)
      // ...
    else
      // ...
  }
  // if we have an indirection, return the functor,
  // else return the whole App
  return (args == null) ? functor : this;
}
```

The `App` being evaluated forms the spine of the graph, with the `functor` being the base of the spine and the `args` forming the ribs. If the `App` is not in normal form, then we have one of three cases, corresponding to the three branches of the `if-then-else` ladder inside the `while` loop.

If we have a function applied to the exact number of arguments (that is, when `total_app` is true), we simply need to apply the function to its arguments and update the `App`:

```
if (total_app)
  update(((Func) functor).apply(args));
```

The method `Func.apply` is where all the work is done. In this method, we unpack any indirections from `App`s and then invoke the function on these unpacked arguments:

```
public Object apply(Object[] as) {
  for (int i = 0; i < as.length; i++)
    if (as[i] instanceof App) {
      App a = (App) as[i];
      if (a.args == null) // N.B. we keep CAFS inside Apps
        as[i] = a.functor;
    }

  return method.invoke(null, as);
}
```

If we have a function applied to too many arguments, i.e. the `functor` is a function and both `total_app` and `in_nf` are false. In this case we split the `args` array into two, apply the `functor` to the number of arguments it needs and update the `functor` and `args` fields appropriately:


```
else if (functor instanceof Func) {
  // we must have more arguments than the function takes
  Func f = (Func) functor;

  // split this array into two parts.
  int unused = args.length - f.arity;

  Object[] first = new Object[f.arity];
  Object[] rest = new Object[unused];
  split(args, f.arity, first, rest);

  // the functor becomes the result of apply f to the first arity arguments
  functor = f.apply(first);

  // and the args become the rest of the args
  args = rest;

  setType();
}
```

The method `App.setType` examines the `functor` and `args` fields of the `App` and sets `total_app` and `in_nf` appropriately.

The last case is when the functor is another `App`, in which case we need to unwind the `App` at the functor onto this one. If the `functor` is a function applied to the correct number of arguments, then we do the application before unwinding:

```
else { // functor instanceof App
  // need to unwind
  App a = (App) functor;

  if (a.total_app)
    // the functor contains a function applied to the correct
    // number of arguments, so we apply it and continue unwinding
    a.update(((Func) a.functor).apply(a.args));
  else {
    functor = a.functor;
    args = cat(a.args, args);
  }

  setType();
}
```

The method `App.cat` joins together two arrays in a manner similar to list concatenation in functional languages.

Printing

The evaluation of graph nodes is Initially triggered by the `Node.print` method, which evaluates a node and prints it on standard output:


```
public final static void print(Object node) {
  node = eval(node);

  if (node instanceof Cons) {
    System.out.print("[");
    boolean not_at_end = true;
    do {
      Cons c = (Cons) node;
      c.head = eval(c.head); // evaluate and update head
      c.tail = eval(c.tail); // and tail
      printNoEval(c.head);
      if (c.tail instanceof Cons) {
        System.out.print(", ");
        node = c.tail;
      }
      else
        not_at_end = false;
    } while (not_at_end);
    System.out.print("]");
  }
  else
    System.out.print(node);
}
```

The method `Node.printNoEval` is similar to `print`, except that it presumes that its argument is in normal form and thus doesn't bother evaluating it before printing.

If the node *isn't* a `Cons`, then we simply print out the node using the `toString` method (which is implicitly called by the `System.out.print` method). Otherwise, if it is a `Cons` then we iteratively print out each element of the list. It is done this way, rather than farming it out to some method of `Cons`, which would be the standard object-oriented way, as we don't wish to keep an unnecessary reference to the head of the list (note that we repeatedly overwrite the head in `Node.print`), which could lead us to keeping the whole of the list that is being printed in memory when in reality it could be garbage and the memory it occupies could be freed. We also reassign `c.head` and `c.tail` after evaluating them; this enables us to bypass any indirection introduced by any evaluation, which needed to be done since the `eval` method will return the evaluated object unpacked from the application which is serving as an indirection.

Primitives

Like user-defined functions we store primitives, such as arithmetic operators, comparison operators, list constructors and deconstructors, in Java class files. The primitives are spread over a number of classes, which are all subclasses of the `Node` class which contains the top-level evaluation and print routines. The strict primitives are kept in different classes to the lazy ones, thus giving us a quick and dirty way of determining if a primitive is strict. We also ensure that the result of all strict primitives is in normal form.

We allow overloading on primitives, e.g. we use `-` for integer and real subtraction, but this is done in an *ad hoc* way inside the primitive itself, making use of the Java `instanceof`

operator rather than in any systematic kind of way such as the use of type classes in Haskell. For example, subtraction is performed using the `_minus` method in the class `StrictPrimitives`:

```
public class StrictPrimitives extends Node {
  public final static Class TYPE = (new StrictPrimitives()).getClass();

  public static Object _minus(Object lhs, Object rhs) {
    lhs = eval(lhs);
    rhs = eval(rhs);
    if (lhs instanceof Long && rhs instanceof Long)
      return new Long(((Long) lhs).longValue() -
                      ((Long) rhs).longValue());
    else
      return new Double(((Number) lhs).doubleValue() -
                        ((Number) rhs).doubleValue());
  }
  public final static Object _minus = Function.make(TYPE, "_minus", 2);
  // ...
}
```

The `StrictPrimitives` has a `TYPE` field, which stores the `Class` representation of itself which is used to construct the `Object` equivalent of each method.

The `_minus` method first evaluates its two arguments and reassigns them (remember that the `eval` method unpacks any indirections). It then determines whether it is doing integer or real subtraction using the `instanceof` operator (integers are cast to reals if necessary), does the necessary subtraction, and returns a new object containing the result of the subtraction.

The structure of the primitive classes and the classes created by the compiler are identical, and hence we can treat primitives exactly the same way as we do user-defined functions, except in the case of a tail recursion (see below).

COMPILATION

In this section we deal with the creation of Java class files from our Ginger source which has been lambda-lifted and had dependency analysis performed on it (see Figure 1). Rather than creating the class files directly, or using the Java language itself as a source (which would complicate matters *viz* local variables), we target the Jasmin assembly language [1]. This language is very similar to the byte code used by the Java Virtual Machine, but is easier to program in as it deals with such things as the Java constant pool (where all constants and object references as such) and calculating offsets for jumps automatically. Our Ginger program, in the file *prog*`.g`, is compiled into an intermediate Jasmin file *prog*`.j` (which may be discarded after use), which is assembled into a Java class file *prog*`.class` by Jasmin.

Each supercombinator definition of our Ginger program is compiled into a static method of the class file we are creating. Functions are not compiled by creating code to create a new instance each time one occurs, but rather a single instance is created and is stored as a static field of the class we are creating. These fields will be set up in a static initialiser of the class we are creating. Therefore, whenever we want a function we just access the relevant field. This method also applies when we want to access a function defined in another class. The job of the various `import` declarations is thus just to tell us in which class to find each function that we use. There is a limit on the number of fields, i.e. functions, in a class (65,535), but if

this limit is reached then the program can be split up into smaller segments (a program which hit this limit must have been fairly big and unwieldy anyhow).

Our compilation schemes are based on those presented in Peyton Jones (1987,92) [4]. We view our source as a triple $\langle cl, fs, ss\rangle$, where *cl* is the class we are to create, *fs* is the set of *all* functions defined or imported, and *ss* is the set of supercombinator definitions. Note that, although the JVM has shorter, more optimal versions of some instructions (the instruction `iconst_0` is a more efficient way of loading the integer zero onto the stack than `ldc2_w 0`, for example), for clarity we use the most general instruction in our description of the compilation schemes, though we do use the most efficient instruction in our actual implementation. Our primary compilation scheme, $\mathcal{P}$, starts off as:

$$\begin{aligned}&\mathcal{P}\langle cl, fs, \{s_1, \ldots, s_n\}\rangle = \\ &\quad \texttt{.class public } cl \\ &\quad \texttt{.super } \textit{Object}\end{aligned}$$

This declares our class and its superclass. Note that Jasmin requires the full name of all classes and members, but for brevity we have omitted the package name where this is obvious, indicating the omission by using italics for object names rather than teletype. We then proceed by declaring the fields corresponding to each function defined in the file:

$$\begin{aligned}&\texttt{.field public static } \phi(s_1)\ \texttt{L}\textit{Object}\texttt{;} \\ &\vdots \\ &\texttt{.field public static } \phi(s_n)\ \texttt{L}\textit{Object}\texttt{;}\end{aligned}$$

The function ϕ returns the name of the field that holds the supercombinator, which is just its name of said supercombinator. Note that when an object name, *obj* say, is used as a type, it is written as `L`*obj*`;`. Functions imported will be declared and defined in the class that they are imported from. $\mathcal{P}$ progresses by setting each of these fields to its appropriate value inside a static initialiser which is a method called `clinit` that takes no arguments, and returns `void` (indicated by the `V`):

$$\begin{aligned}&\texttt{.method <clinit>()V} \\ &\texttt{new } cl \\ &\texttt{dup} \\ &\texttt{invokespecial } cl\texttt{/<init>()V} \\ &\texttt{invokevirtual } cl\texttt{/getClass()L}\textit{Class}\texttt{;} \\ &\texttt{astore\_0} \\ &\mathcal{D}\ s_1 \\ &\vdots \\ &\mathcal{D}\ s_n \\ &\texttt{return} \\ &\texttt{.end method}\end{aligned}$$

The method first gets the `Class` object reflecting the class we are creating and stores it in register 0. This `Class` object is used when creating the `Func` object representing each supercombinator (or `App`s in the case of CAFs). $\mathcal{D}$ creates the code necessary to create a new

instance of its argument, and store it in the relevant field. If we have a supercombinator, *s* say, then we have:

$$\begin{aligned}\mathcal{D}\,s \;=\;& \texttt{aload\_0}\\ & \texttt{ldc } n(s)\\ & \texttt{ldc } a(s)\\ & \texttt{invokestatic } \textit{Function}\texttt{/make(L}\textit{Class}\texttt{;L}\textit{String}\texttt{;I)L}\textit{Object}\texttt{;}\\ & \texttt{putstatic } \phi(s)\ \texttt{L}\textit{Object}\texttt{;}\end{aligned}$$

This loads the `Class` object reflecting *cl* onto the stack, then the name of the supercombinator (using the function *n*) and its arity (using *a*). This information is then used by the method `Function.make` to create a `Func` object (non-CAFs) or an `App` (CAFs), which is then stored in the appropriate field.

Moving back to the $\mathcal{P}$ scheme, after declaring and defining our constants, we now need to create the code for each of the supercombinators. This is done using the $\mathcal{F}$ scheme:

$$\begin{aligned}&\mathcal{P}\langle cl, fs, \{c_1, \ldots, c_m\}, \{s_1, \ldots, s_n\}\rangle =\\ &\quad\vdots\\ &\quad\mathcal{F}\ fs\ s_1\\ &\quad\vdots\\ &\quad\mathcal{F}\ fs\ s_n\end{aligned}$$

The $\mathcal{F}$ scheme is defined below. Finally, we need to determine if we need to create a `main` method which will make the class file executable, using the `java` interpreter, say. This is so if we have defined a function called `main`. The `main` method will print the result of evaluating the Ginger function `main` which we rename `_main`, so as to separate the reduction rule from the code which does the evaluation and printing. The code for this is:

```
.method public static main([LString;)V
getstatic φ(_main) LObject;
invokestatic Node/print(LObject;)V
return
.end method
```

Here the `[String` denotes an array of strings (the closing brace is not used) which in the case of the arguments to the `main` method represent the command-line arguments passed by the Java interpreter. If we don't create a `main` method then we can view the generated class as a library of functions.

Compiling supercombinators

We now need to give the definition of $\mathcal{F}$ which creates a method from a supercombinator. This method takes *n* arguments of type `Object`, where *n* is the arity of the supercombinator in question, and returns an object of type `Object`. The return object will be the object left on the top of the stack by the code generated by the $\mathcal{R}$ scheme, which compiles the expression

on the right-hand side of a supercombinator definition.

$$\mathcal{F}[\,f\ x_1\ \ldots\ x_n = E\,]\ fs =$$

```
    .method public static f(LObject; ... LObject;)LObject;
                                  n times
    R E [x1 = 0, ..., xn = n − 1] fs n
    areturn
    .end method
```

The scheme $\mathcal{R}$ takes as arguments the expression to compile, an environment detailing which register each variable is in, the set of functions defined and imported, and the next free variable register (used to store local variables).

The $\mathcal{R}$ compilation scheme

The purpose of the $\mathcal{R}$ scheme is to take an expression which forms the right-hand side of a definition (the return expression) and compile it to code that will, when executed, create the graph of the expression and leave a reference to it on top of the stack.

If we have an integer, i say, then we have to create a new `Long` object to store it in:

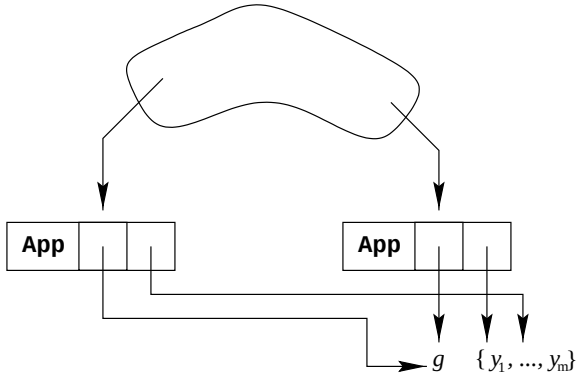
$$\mathcal{R}\ i\ \rho\ fs\ v\ =$$

```
new Long
dup
ldc2_w i
invokespecial Long/<init>(J)V
```

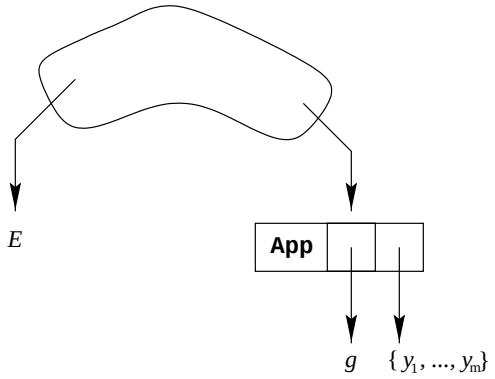
The instruction `ldc2_w` is used to load a long or a double onto the stack (each stack cell is 32 bits in size, but long integers and doubles take up 64 bits so they have to be spread across two cells). A similar method is used to define the other types of constants.

If the right-hand side of a super-combinator consists of a single variable then, unless we evaluate it before returning, we risk doing extra work because some loss of sharing occurs. Suppose we have an expression $f\ x_1\ \ldots\ x_n$ which reduces to an expression $g\ y_1\ \ldots\ y_m$, and further suppose that this latter expression is reducible. These are represented as two `App`s and recall that updating the original expression involves copying the fields of the `App` representing the value of the reduction onto the fields of the `App` representing the original expression.

If the `App` $g\ y_1\ \ldots\ y_m$ was created during the evaluation of the reduction rule for f, then the `App` object $g\ y_1\ \ldots\ y_m$ is never used again (i.e. it becomes garbage), though its fields become the fields of the original expression. However, if the `App` *wasn't* created by reduction rule, then it must be referenced by some other part of the graph, and hence we have two copies of the same application:



Suppose that $g\ y_1\ \ldots\ y_m$ evaluates to some expression E, say. Then if we continue our evaluation we have:



Note that we have only reduced one of the `App`s: by copying we have lost not only the sharing of nodes but the sharing of work.

This situation occurs whenever we have a function whose return value is a single variable or function (or, more specifically, a CAF). We can prevent the replication of work by making sure that whenever we have such a function we first evaluate it (to normal form) before returning it. Thus if such a function still returns an `App` it will be in normal form, and although we may still make an unnecessary copy of it, we cannot waste time by duplicating evaluation as there is no evaluation to do. The $\mathcal{R}$ scheme for variables and CAFs is thus:

$$\mathcal{R}\ id\ \rho\ fs\ v \quad = \quad \mathcal{C}\ id\ \rho\ fs\ v$$
`invokestatic` *Node*`/eval(L`*Object*`;)L`*Object*`;`

If *id* is a function of arity greater than zero, we have:

$$\mathcal{R}\ id\ \rho\ fs\ v = \texttt{getstatic}\ \phi(id)\ \texttt{L}\mathit{Object};$$

Before compiling applications we first unwind them, using the left-associativity of function application, so they are of the form $f\ e_1\ \ldots\ e_n$, where f is an identifier (anything else would be a type error). If f refers to a variable, then we just compile the arguments and functor of the application, packing the arguments into an array, and construct the `App` object:

$\mathcal{R}\ (f\ e_1\ \ldots\ e_n)\ \rho\ fs\ v =$
`new` *App*
`dup`
`ldc` n
`anewarray` *Object* } Create an array of n objects
`dup`
`ldc 0`
$\mathcal{C}\ e_1\ \rho\ fs\ v$
`aastore` } Set the 0^{th} element of the argument array
$\vdots$
`dup`
`ldc` $(n-1)$
$\mathcal{C}\ e_n\ \rho\ fs\ v$
`aastore` } Set the $(n-1)^{\text{th}}$ element of the argument array
$\mathcal{C}\ f\ \rho\ fs\ v$
`invokespecial` *App*`/<init>([L`*Object*`;L`*Object*`;)V`

Here $\mathcal{C}$ is the generic scheme used to compile expressions (see below).

If f is a function, then we can provide a bit more information to the constructor. If there aren't enough arguments present then we can tell the `App` constructor to set the `in_nf` field. If there are exactly enough arguments present, then we tell the constructor to set the `total_app` field. This is done by using a three-constructor of `App` which as well as taking the arguments and `functor` of the application takes an additional boolean which if set to `true` sets the `in_nf` and the `total_app` to `false`, and *vice versa*. For these two cases we have:

$\mathcal{R}\ (f\ e_1\ \ldots\ e_n)\ \rho\ fs\ v =$
`new` *App*
`dup`
nf
$\vdots$
compile arguments and functor as before
$\vdots$
`invokespecial` *App*`/<init>(Z[L`*Object*`;L`*Object*`;)V`
where
nf = `iconst_1`, **if** $n <$ arity of f
= `iconst_0`, **if** $n =$ arity of f

The JVM uses integers to represent booleans (the type of which is denoted by a `Z`). The most efficient way to load these onto the stack are using the `iconst_1` for `true` and `iconst_0` for `false`.

If there are too many arguments present, then we split the application into two: if f is of arity m and is applied to n arguments where $n > m$, then we create the application of f applied to the first m arguments applied to the other $n - m$ arguments.

$\mathcal{R}\ (f\ e_1\ \ldots\ e_m\ e_{m+1}\ \ldots\ e_n)\ \rho\ fs\ v =$
    `new` *App*
    `dup`
    `ldc` $(n - m)$
    `anewarray` *Object*
    `dup`
    `ldc 0`
    $\mathcal{C}\ e_m\ \rho\ fs\ v$
    `aastore`
    $\vdots$
    `dup`
    `ldc` $(n - m - 1)$
    $\mathcal{C}\ e_n\ \rho\ fs\ v$
    `aastore`
    $\mathcal{C}\ (f\ e_1\ \ldots\ e_m)\ \rho\ fs\ v$
    `invokespecial` *App*`/<init>([L`*Object*`;L`*Object*`;)V`

Compiling `if` statements requires us to evaluate the antecedent and jump accordingly.

$\mathcal{R}$ (`if` a `then` t `else` f `endif`) $\rho\ fs\ v =$
    $\mathcal{E}\ a\ \rho\ fs\ v$
    `checkcast` *Boolean*
    `invokevirtual` *Boolean*`/booleanValue()Z`
    `ifeq` *FALSE*
    $\mathcal{R}\ t\ \rho\ fs\ v$
    `goto` *ENDIF*
    FALSE`:`
    $\mathcal{R}\ f\ \rho\ fs\ v$
    ENDIF`:`

where *FALSE* and *ENDIF* are unique labels. The `checkcast` instruction makes sure that we have a `Boolean` object after evaluating the antecedent. The scheme $\mathcal{E}$ is used to compile

an expression whose result is known to be needed. Note that the $\mathcal{R}$ scheme is used to compile the two branches of the conditional.

Compiling a simple `let` requires us to compile the definition, store it in the next free local variable, updating the environment accordingly, and compiling the body with respect to this new environment.

$$\begin{aligned}&\mathcal{R}\ (\texttt{let}\ x\ \texttt{=}\ d\ \texttt{in}\ b\ \texttt{endlet})\ \rho\ fs\ v = \\ &\quad \mathcal{C}\ d\ \rho\ fs\ v \\ &\quad \texttt{astore}\ v \\ &\quad \mathcal{R}\ b\ \rho[v = n]\ fs\ (v + 1)\end{aligned}$$

Compiling a `letrec` is more complex, as each definition in the block will refer to at least one other one, and hence if we are not careful we could end up loading objects from registers that haven't yet been filled. We thus need to first of all put place-holders in each of the registers that are to be defined by the `letrec` and update them when each appropriate definition is compiled. These place-holders are `App`s whose `functor` and `arg` fields are null, and they are updated using the same `update` method used to update `App`s that have been evaluated (see above).

$$\begin{aligned}&\mathcal{R}\ (\texttt{letrec}\ ds\ \texttt{in}\ b\ \texttt{endletrec})\ \rho\ fs\ v = \\ &\quad A\ ds\ v \\ &\quad CL\ ds\ \rho'\ fs\ v' \\ &\quad \mathcal{R}\ b\ \rho'\ fs\ v' \\ &\quad \textbf{where}\ (\rho', v') = X\ ds\ \rho\ v\end{aligned}$$

Here A allocates the place-holders, CL compiles the definitions and updates the registers and X updates the environment and the next free variable. A is defined as:

$$\begin{aligned}&A\ (x_1 = e_1, \ldots, x_k = e_k)\ v = \\ &\quad \texttt{new}\ App \\ &\quad \texttt{dup} \\ &\quad \texttt{invokespecial}\ App\texttt{/<init>()V} \\ &\quad \texttt{astore}\ v \\ &\quad \vdots \\ &\quad \texttt{new}\ App \\ &\quad \texttt{dup} \\ &\quad \texttt{invokespecial}\ App\texttt{/<init>()V} \\ &\quad \texttt{astore}\ (v + k)\end{aligned}$$

CL is defined as:

$$
\begin{aligned}
&CL\ (x_1 = e_1, \ldots, x_k = e_k)\ \rho\ fs\ v = \\
&\quad \texttt{aload}\ (v - k) \\
&\quad \mathcal{C}\ e_1\ \rho\ fs\ v \\
&\quad \texttt{dup} \\
&\quad \texttt{invokevirtual}\ \textit{App}\texttt{/update(L}\textit{Object}\texttt{;)V} \\
&\quad \vdots \\
&\quad \texttt{aload}\ v \\
&\quad \mathcal{C}\ e_k\ \rho\ fs\ v \\
&\quad \texttt{dup} \\
&\quad \texttt{invokevirtual}\ \textit{App}\texttt{/update(L}\textit{Object}\texttt{;)V}
\end{aligned}
$$

Finally, X is defined as

$$X\ [x_1 = e_1, \ldots, x_k = e_k]\ \rho\ v = (\rho[x_1 = v, \ldots, x_k = v + k - 1], v + k)$$

The $\mathcal{C}$ compilation scheme

The $\mathcal{C}$ scheme, for the most part, is similar to the $\mathcal{R}$ scheme. The main difference is in the handling of variables. Since we don't have to worry about any loss of sharing, the definition of $\mathcal{C}$ when compiling a single variable is:

$$
\begin{aligned}
\mathcal{C}\ id\ \rho\ fs\ v &= \texttt{getstatic}\ \phi(id)\ \texttt{L}\textit{Object}\texttt{;}, && \textbf{if}\ id \in fs \\
&= \texttt{aload}\ \rho(id), && \textbf{otherwise}
\end{aligned}
$$

We also use the $\mathcal{C}$ scheme to compile the branches of conditionals and the body of local variable declarations, rather than the $\mathcal{R}$ scheme as in the definition of the $\mathcal{R}$ scheme.

The $\mathcal{E}$ compilation scheme

The $\mathcal{E}$ scheme is used when we know that an expression is to be evaluated and is *not* a tail call (this is handled by $\mathcal{R}$). Later on, we shall see how we can optimise the code produced by this scheme, but for now we shall just add a call to `Node.eval`:

$$
\begin{aligned}
\mathcal{E}\ e\ \rho\ fs\ v = \quad & \mathcal{C}\ e\ \rho\ fs\ v \\
& \texttt{invokestatic}\ \textit{Node}\texttt{/eval(L}\textit{Object}\texttt{;)L}\textit{Object}\texttt{;}
\end{aligned}
$$

We can also use the $\mathcal{E}$ scheme to compile the branches of conditionals and the body of local variable declarations.

Tail recursion

A tail recursion occurs when the result of one function is the result of applying another function to the correct number of arguments. Any tail-recursive calls in our program will be compiled using the $\mathcal{R}$ scheme. It is a property of graph reduction that tail-recursive calls can

Table I. Initial running times of programs produced by the Giner compiler

Program	Time (s)
`take 500 primes`	65.4
`nfib 30`	357.4
`soda`	11.5
`cal`	34.1
`edigits 250`	82.8
`queens 8`	107.2

be run in constant space, no matter how deep the recursion is [14], and our implementation preserves this property. This is because if we have an expression $f\ x_1 \ldots\ x_n$ that reduced to $g\ y_1\ \ldots\ y_m$, then the call to g is not executed inside the code of f, but the application is passed back to the `eval` method which then calls g. Thus, if we have a chain of tail-recursive calls $g_1, \ldots, g_n$ with g_i calling g_{i+1}, then instead of recursing down the chain of g_is and back again, and having a recursion that is $O(n)$ levels deep, we instead 'bounce' between `eval` and each g_i, and the recursion only $O(1)$ levels deep. Hence our implementation executes tail-recursive functions in constant space.

Initial results

Table I shows the running times of some programs compiled using our Ginger compiler. Because the running times of Java programs can vary significantly, we have averaged our times over three runs. The programs were run on a Sun Enterprise 3000 with two 168 MHz processors and 512 MB of memory running Solaris 2.6 and using the Sun JDK 1.1.5. Descriptions of the programs are as follows:

(a) `take 500 primes` outputs the first 500 prime numbers using the sieve of Eratosthenes method.
(b) `nfib 30` calculates the number of reductions used to calculate the 30th Fibonacci number using the naïve doubly-recursive method.
(c) `soda` performs a serial word-search on a 10×15 grid.
(d) `cal` outputs calendars for the years 1990–99.
(e) `edigits 250` evaluates the first 250 digits of e (2.7182 . . .).
(f) `queens 8` prints all 92 solutions to the eight queens problem.

OPTIMISATIONS

In this section we detail two optimisations that can increase the performance of the code produced by our compiler: the direct invocation of some function applications, and the use of a single instance to represent each constant declared by a program.

Direct function invocations

Implementations of lazy functional languages use application nodes to store 'suspended' function calls, that is function calls whose result may or may not be needed. However, in some cases it can be predicted that the result of the function call will be needed, and we can avoid

having to build the application representing the function call and instead invoke the function directly.

The first place we can use this optimisation is in the $\mathcal{E}$ scheme, as we know that we always need the result of an expression compiled using this scheme. If we have a function, f, of arity n applied to the correct number of arguments, we have:

$$
\begin{aligned}
&\mathcal{E}\ (f\ e_1\ \ldots\ e_n)\ \rho\ fs\ v = \\
&\quad \mathcal{C}\ e_1\ \rho\ fs\ v \\
&\quad \vdots \\
&\quad \mathcal{C}\ e_n\ \rho\ fs\ v \\
&\quad \texttt{invokestatic}\ f(\underbrace{\texttt{L}\mathit{Object};\ldots\texttt{L}\mathit{Object};}_{n\ \text{times}})\texttt{L}\mathit{Object}; \\
&\quad \texttt{invokestatic}\ \mathit{Node}/\texttt{eval(L}\mathit{Object};\texttt{)L}\mathit{Object};
\end{aligned}
$$

If f is known to be strict then we can compile each of the arguments using the $\mathcal{E}$ scheme rather than the $\mathcal{C}$ one.

We can also use this technique with the $\mathcal{R}$ scheme, with one major caveat. It is safe to evaluate all tail calls, which will be compiled using the $\mathcal{R}$ scheme, since their result will eventually be needed. However if the method we invoke is itself tail-recursive then we could have a long chain of recursions and evaluation will no longer occur in constant space, and we could be in danger of overflowing the stack on which the JVM stores the return address for each method call. Although it is possible to optimise tail calls by replacing a method call with a jump, many JVM implementations do not do so. We cannot provide this optimisation manually either, as the JVM does not allow jumps between methods.

We thus only directly invoke tail calls involving functions that are not tail-recursive, which for simplicity's sake we assume that is just our set of primitives, none of which are tail-recursive. If we have a primitive p applied to the correct number of arguments, we have:

$$
\begin{aligned}
&\mathcal{R}\ (p\ e_1\ \ldots\ e_n)\ \rho\ fs\ v = \\
&\quad \mathcal{C}\ e_1\ \rho\ fs\ v \\
&\quad \vdots \\
&\quad \mathcal{C}\ e_n\ \rho\ fs\ v \\
&\quad \texttt{invokestatic}\ p(\underbrace{\texttt{L}\mathit{Object};\ldots\texttt{L}\mathit{Object};}_{n\ \text{times}})\texttt{L}\mathit{Object};
\end{aligned}
$$

If p is known to be strict, then we can compile each of the arguments using the $\mathcal{E}$ scheme rather than the $\mathcal{C}$ one. In all other cases, $\mathcal{R}$ remains as before. As can be seen from Table II, direct invocation is a very worthwhile optimisation.

Single-instance constants

Since constants are immutable in a pure functional language, we can represent each constant used by a pure functional program by a single instance, saving both the time and space needed to create a new instance of a constant each time one is encountered. We shall store each constant as a static field of the class that our functional program is compiled to (cf. how we store functions), and thus each time we need an instance of a constant we just need

Table II. Running times (s) of programs produced by the Ginger compiler with and without direct function invocation

Program	Non-optimised	Direct invocation	Decrease (per cent)
`take 500 primes`	65.4	38.2	42
`nfib 30`	357.4	140.2	61
`soda`	11.5	8.4	27
`cal`	34.1	25.6	25
`edigits 250`	82.8	49.1	41
`queens 8`	107.2	73.0	32

to access the relevant field. This is similar to a technique used by most Java implementations to implement strings (which are immutable in Java).

It is required that the $\mathcal{P}$ scheme is modified to handle constants. Suppose that $c_1, \ldots, c_n$ form the set of constants that are program uses. Then $\mathcal{P}$ becomes:

$\mathcal{P}\langle cl, fs, \{c_1, \ldots, c_m\}, \{s_1, \ldots, s_n\}\rangle =$

```
.class public cl
.super Object
.field public static φ(s1) LObject;
⋮
.field public static φ(sn) LObject;
.field public static φ(c1) LObject;
⋮
.field public static φ(cm) LObject;

.method <clinit>()V
new cl
dup
invokespecial cl/<init>()V
invokevirtual cl/getClass()LClass;
astore_0
D s1
⋮
D sn
D c1
⋮
D cm
return
.end method
```


Table III. Running times (s) of programs produced by the Ginger compiler with and without single-instance constants

Program	Non-optimised	Single-instance constants	Decrease (per cent)
`take 500 primes`	65.4	47.1	28
`nfib 30`	357.4	326.0	9
`soda`	11.5	11.4	1
`cal`	34.1	33.4	2
`edigits 250`	82.8	67.2	19
`queens 8`	107.2	103.1	4

where the other names are as in the original definition of $\mathcal{P}$. The function ϕ has been extended to return the field name of a constant as well as that of a supercombinator. The scheme $\mathcal{D}$ is also extended to take constants as arguments. For example, if we have an integer, i, we have:

$$\begin{aligned}\mathcal{D}\,i \;=\;& \texttt{new } \textit{Long}\\ & \texttt{dup}\\ & \texttt{ldc2\_w } i\\ & \texttt{invokespecial } \textit{Long}\texttt{/<init>(J)V}\\ & \texttt{putstatic } \phi(i)\ \texttt{L}\textit{Object}\texttt{;}\end{aligned}$$

A similar method is used to define the other types of constants. As we can see from Table III, this results in a modest, but significant speed-up in the running times of our programs.

RESULTS AND OTHER WORK

The only other work we are aware of in this area is that of Wakeling, one based on the G-Machine [8], which translates the G-code produced by HBC (the Haskell compiler developed at Chalmers) into Java bytecode; and one based on the $\langle \nu, \mathrm{G} \rangle$ machine [7] which compiles a core language into a set of $\langle \nu, \mathrm{G} \rangle$ instructions which are then transformed in Java bytecode, again using HBC. Both versions use a separate class, and hence a separate file, for each *function*, rather than each *program*, as with our compiler. There is also a compiler for Standard ML from Persimmon which compiles stand-alone SML programs to Java bytecode [15] ,but as this is for a strict language we do not include it in our comparisons.

Table IV gives the running times for several programs using our compiler with both optimisations switched on, and using both Sun's JVM and the Kaffe Open VM [16] to run the generated class files; Wakeling's compiler (the $\langle \nu, \mathrm{G} \rangle$-Machine version [7]) using Sun's JDK; the Haskell interpreter Hugs (version 1.4); and the Glasgow Haskell Compiler, GHC (version 2.10). The Haskell and Ginger sources were made as close as possible, but all the Haskell programs compiled using Wakeling's compiler have been explicitly mono-typed where appropriate. This is because if the overloading used in the programs is not resolved at compile-time, then the running times can slow down by as much as a factor of 10 in extreme cases, because of the need to pass around dictionaries to resolve the overloading at run-time.

Both the JVM-targeting compilers perform poorly when compared to Hugs, with our compiler (using the JDK to execute the class files) being some 4–11 times slower than Hugs except in the case of `nfib`, which is a somewhat artificial benchmark anyway, when

Table IV. Running times (rounded up to the nearest tenth of a second)

	Gingerc				
Program	Sun JDK	Kaffe	Wakeling's	Hugs	GHC
`take 500 primes`	30.5	266.2	50.5	6.3	0.8
`nfib 30`	118.0	638.4	56.6	114.6	6.7
`soda`	8.7	43.9	4.2	0.9	0.1
`cal`	25.0	140.3	19.4	5.0	0.3
`edigits 250`	46.2	174.4	10.0	4.0	0.5
`queens 8`	72.6	369.6	46.9	16.2	0.9

performance matched that of Hugs. Why then is our compiler so much slower than Hugs, when both are either interpreters or produce code that is interpreted? First, there are the deficiencies in our source language, Ginger, when compared to the much more complex Haskell. In particular, the lack of static type-checking and user-defined (algebraic) types, and all but the crudest from of strictness analysis will have an appreciable effect on run-time performance, but Wakeling's compiler has these and it is not much faster than ours. There is also the cost incurred by not being able to resolve overloading until runtime; if we could, at least partially, then we could replace function calls to methods implementing basic primitives such as arithmetic and comparison operators to uses of basic JVM instructions. There is also the cost of using a high-level language (Java) as opposed to the lower-level C used to implement Hugs. For instance, Java does bounds checking on array accesses, and that casts are legal whereas C does not, and both these operations occur frequently in our compiler.

Wakeling [8] ascribed the poor performance of his compiler when compared to Hugs to the poor memory handling in the JVM, hypothesising that memory-allocation in Java is an order of magnitude more expensive than in Hugs. Functional programs certainly will create and destroy objects on a more frequent basis than an imperative object-oriented one – both `primes` and `edigits` creates something in the order of 500,000 `App` nodes and 130,000 `Cons` nodes, for example. Unfortunately, using the `-profile` of the `java` interpreter to look at the cost of these allocations is not useful as we can only look at the time taken by the code in the constructor (around 2.5 seconds for all 500,000 `App` objects used by `primes`) and not the time taken to allocate the memory, which is done before the constructor is invoked.

It is the allocation of objects which we suspect to be the reason why running our programs using the Kaffe VM is some 3–9 times slower than running them using the Sun JVM, despite the fact that in some cases Kaffe can be around 2–3 times faster than the Sun JVM. Consider the following segment of code, which we compile using both the Kaffe and the Sun Java compiler:

```
Long l;
for (int i = 0; i < 1000000; i++)
  l = new Long(i);
```

The Kaffe VM takes 44 seconds to run the code produced by both compilers, while the Sun JVM takes only 3 seconds (both Virtual Machines take only half a second to run the same loop with an empty body).

Although the large amount of mutual recursion in our programs confuses Java's profiler, making a detailed run-time analysis difficult, we can examine the overhead imposed by the

Table V. Running times (s) of programs produced by Ginger compiler with initial heap sizes of 1 and 4 megabytes

Program	1 MB	4 MB	Decrease (per cent)
`take 500 primes`	30.5	28.0	8
`nfib 30`	118.0	119.2	−1
`soda`	8.7	7.8	10
`cal`	25.0	20.7	17
`edigits 250`	46.2	23.2	50
`queens 8`	72.6	64.2	12

garbage collector, using the `-profile` or the `-verbosegc` options of the Java interpreter. This shows that using the default initial heap size of 1 megabyte garbage collection (as in Table IV) takes anything from 10 per cent of the total running time to 50 per cent in the extreme cases like `edigits`. If we increase the initial heap size to 4 MB then we get the running times in Table V. The speed-up in running time is due to garbage collection being run less frequently, but freeing more memory when it does.

CONCLUSION AND FURTHER WORK

We have succeeded in producing a compiler for a functional language which creates Java class files as its object code, with a performance comparable to that of an approach which used a fully-fledged compiler, with various optimisations not present in our compiler, as its front end. However the performance of our compiler is poor when compared to a conventional lazy functional language interpreter (Hugs). This leads us to suspect that we may have reached a point where we cannot achieve any significant speed-ups no matter how we optimise our run-time architecture of the graph reduction on the JVM, and that any major leaps in performance can only come from optimising the JVM itself.

Further work may involve basing our evaluation mechanism on a different abstract machine, such as the Three Instruction Machine (TIM) [17], the Spineless, Tagless G-Machine [18] or the lazy abstract machine derived from Launchbury's semantics for a lazy functional language [19] by Sestoft [20].

REFERENCES

1. J. Meyer and T. Downing, *Java Virtual Machine*, O'Reilly, 1997.
2. M. Joy, 'Ginger – a simple functional language', Technical Report CS-RR-235, Department of Computer Science, University of Warwick, Coventry, UK, 1992.
3. T. Johnsson, 'Efficient compilation of lazy evaluation', *Proceedings of the 11th ACM Symposium of Principles of Programming Languages*, 1984, pp. 58–69.
4. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.
5. M. Jones, 'Hugs documentation', (http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs14/docs/index.html).
6. The Aqua Team, 'Glasgow Haskell Compiler User's Guide, Version 0.26', Department of Computer Science, Glasgow University, March 1996.
7. D. Wakeling, 'Mobile Haskell: Compiling lazy functional language for the Java Virtual Machine', Unpublished paper.
8. D. Wakeling, 'VSD: A Haskell to Java Virtual Machine code compiler', *Proceedings of the 9th International Workshop on the Implementation of Functional Languages*, 1997.
9. K. Arnold and J. Gosling, *The Java Programming Language, 2nd ed.*, Addison-Wesley, 1998.

10. Sun Microsystems, Inc., 'Javap – The Java Class File Disassembler', (http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/javap.html).
11. S. L. Peyton Jones and D. Lester, *Implementing Functional Language – A Tutorial*, Prentice Hall, 1992.
12. R. Bird, *Introduction to Functional Programming using Haskell, 2nd ed.*, Prentice Hall, 1998.
13. T. Johnsson, 'Lambda-lifting – transforming programs to recursive equations', *Conference on Functional Programming and Computer Architecture*, Nancy, 1985. (LNCS 201, Springer-Verlag, 190–203.)
14. D. Turner, 'A new implementation technique for applicative languages', *Software—Practice and Experience*, 31–49 (1979).
15. Persimmon IT, 'The Persimmon MLJ Compiler', (http://research.persimmon.co.uk/mlj/).
16. Transvirtual Technologies, 'What is Kaffe Open VM?', (http://www.transvirtual.com/kaffe.html).
17. J. Fairbairn and S. Wray, 'TIM: A simple, lazy abstract machine to execute supercombinators', *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, Springer-Verlag, 1987, pp. 34–45.
18. S. L. Peyton Jones, 'Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine', *Journal of Functional Programming*, 127–202 (July 1992).
19. J. Launchbury, 'A natural semantics for lazy evaluation', *Principles of Programming Languages*, Charleston, 1993.
20. P. Sestoff, 'Deriving a lazy abstract machine', *Journal of Functional Programming*, 231–264 (1997).

