The University of New South Wales

School of Computer Science and Engineering

Multi-paradigm Just-In-Time Compilation

Don Stewart

BSc (Computer Science) Honours Thesis

November 2002

Supervisor: Dr. Manuel Chakravarty
Assessor: A/Prof. Jingling Xue

Abstract

This thesis describes a standalone backend for the Glasgow Haskell compiler, targeting the Java virtual machine. The backend generates Java source files, which are then compiled to Java bytecode by a conventional Java compiler. The interface between the front end and backend is via the Spineless Tagless G machine intermediate language.

A key concern is to map the features of the Spineless Tagless G machine efficiently to the Java virtual machine, and to cover in depth the variety of approaches have been made in this respect.

Contents

Chapter 1

Introduction

This thesis focuses on the compilation of the non-strict purely functional language Haskell [13] to the Java virtual machine (*JVM*) [8, 26]. Recognition of the similarity of the Haskell execution model and that of the JVM is important for a clear design and implementation of a Haskell to Java compiler, allowing us to concentrate on the difficult language features. The key features of Haskell for a compiler writer are its non-strict semantics and its higher order, curried functions.

Implementations of functional languages, such as Haskell, have historically differed from those of conventional imperative languages. The compilation of Haskell is usually described by an abstract machine. The abstract machine encapsulates the translation from high to low level code and the runtime system by which the code is executed. Haskell code is then executed via graph reduction: the program is represented as a graph and evaluation takes place by applying rewrite rules to this graph. It has become clear that optimised implementations of graph reduction systems look much the same as a standard imperative language systems, with the addition of a few advanced language features.

For example, the first class citizens of a Haskell system are its heap allocated objects, associating data with functions. A set of primitive data types and operations are required for reasonable execution performance; as is a stack to implement argument passing and flow control between functions. A garbage collector must run to free unused objects. These are common requirements of modern imperative languages. The more advanced features of Haskell require special design consideration, in particular its lazy evaluation paradigm and higher order functions.

Much of the terminology of functional language implementations is specialised and historical in origin. However, the objects described are conventional. Heap allocated data associated with method code is a *closure*, in the functional world. Both data and functions are *values*, stemming from their high level language equivalence. Primitive types are *unboxed* values, they are not "boxed up" onto the heap. Flow control via inspect and branch statements are uniformly implemented as *continuations*, and so on.

The mapping of functional code to standard hardware is often described using an *abstract machine*. One of the most efficient abstract machines is the

Spineless Tagless G machine (*STG machine*) [9], used by the Glasgow Haskell Compiler (*GHC*) [6]. GHC is used as the optimising frontend in this project.

The Java virtual machine (*JVM*) has been a desirable target for a compiler backend for a number of years. Code compiled for the JVM is architecture and operating system independent and this is beneficial for any language, particularly when compared with the problem of porting a compiler to a new architecture.

Java is an object oriented, imperative language compiled to bytecode for execution on a virtual machine. Java is not purely object oriented, however, in that it has a small number of unboxed primitive data types. It has no pointers, only restricted references. Objects and primitive data types are first-class; methods (functions bound to objects) are not. The JVM itself is a stack-based machine. All calculation, at least at the abstract level, takes place on the stack. There are no programmer accessible registers.

The Java virtual machine was designed and optimised specifically to support features of the Java language though, as hinted in the JVM specification, "any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine." [8] There are difficulties in generating efficient code for languages with different evaluation paradigms, although the situation is improving.

The problem of executing functional code on the JVM is then the problem of generating an efficient mapping of the STG abstract machine to the Java virtual machine. As described above, many of the fundamental features of the two systems are similar. There have been some successes in redirecting functional language compilers to the JVM, particularly for the Scheme language [3,19] and Standard ML [1]. There have also been a number of experimental compilers for Haskell [5, 21, 22, 24, 25] and subsets of Haskell [2, 16, 17, 23]. The most efficient Haskell implementations have been based on the STG machine, and these designs generally extended an existing optimising compiler. This is the approach taken by Serpette and Serrano [19] for their Scheme backend, as well as in the Haskell projects of Tullsen [21], Choi, Lim and Han [5], and Vernet [22]. This project follows the same path.

There are four main functional language features that cause problems at the implementation level and that arise throughout this thesis: lazy evaluation, tail calls, flow control and higher order, curried functions.

- Lazy evaluation poses the most problems for a JVM implementation. There must be some technique to capture and pass a suspended chunk of code. On execution this expression needs to be updated with its evaluated form. All this is all reasonably difficult to do well on the JVM.
- Function calls in Haskell are simply jumps to code addresses (tail calls). Returns jump back to an address held on a stack. This is much the same as a conventional function call, except that an STG machine implementation often does not want to use the native stack, nor to be forced to perform register saves on every "call". The other problem is that loops are usually implemented via recursion. Having to perform real function calls for tail-calling recursion puts us at risk of stack overflow and of unnecessary stack

manipulation. The solution is to use some form of `goto` instruction which, of course, is limited on the JVM.

- Flow control via branching is implemented using *continuations*. These are switch blocks to select alternative code paths on scrutiny of an expression's result. They are problematic in that Java compilers often poorly compile the code blocks used for continuations. The local environment of the continuation must also be saved and passed to the selected alternative somehow. This is an issue as we tend not to use the native stack for such purposes.

- Finally, Haskell makes much use of higher order, curried functions. Functions may be called with less than the required number of arguments and may return functions as their result. Functions may be curried: they must be able to accept one argument at a time, returning a new function of lower arity. There is no obvious way to perform this efficiently on the JVM.

In summary, most of the Haskell runtime system maps to conventional hardware cleanly, and the JVM is pretty much just virtual, conventional hardware. It is the advanced language features, however, that pose difficulty in a compiler implementation.

This thesis integrates the ideas developed in existing papers on the subject, and presents a comprehensive approach to the design of a mapping of functional code to Java, and its implementation on the Java virtual machine. Particular attention is paid to the implementation of:

- Closures
- Function and data values
- Suspended computations
- Continuations
- Tail calls
- Updates

The intermediate language, Spineless Tagless G code, produced by GHC is focused on. This is discussed in the context of a new external Java backend for GHC.

Chapter 2

Background

This chapter presents some of the background of the functional programming landscape this project inhabits. Only a brief overview of the rich literature is presented. An extensive discussion of this material is available in [9, 11]. The organisation of this material is inspired by [23] and [9]. An outline of functional programming and its compilation, along with a sketch of the semantics of the Spineless Tagless G machine is the general focus of this section.

2.1 Functional programming

Functional programming languages treat the function as the fundamental unit of the program. A program is just a set of such functions. An important property is the absence of side effects in function evaluation. A function will produce the same result given the same set of arguments. Additionally, iterative loops are formally unnecessary, instead implemented using recursion.

Functional programs are a set of definitions, a set of declarations, of function behaviour. A distinguished function is used as an entry point to program evaluation, equivalent to `main` in a C program.

The evaluation of a functional program can be represented via a graph. Nodes of the graph correspond to values in the program, and the graph is rewritten and reduced as execution proceeds. Eventually the graph reaches and irreducible state, the normal form, signifying completion. This is the historical framework from which modern implementations of functional languages are derived. Contemporary functional language systems are now more like classical imperative systems, with the addition of a few advanced features, such as lazy evaluation.

In general, functional languages hide from the programmer low level details like order of execution and memory allocation and reclamation. These details being determined by the evaluation process instead. As a result, errors related to these features usually do not occur in functional programs: they are dealt with once and for all in the compiler implementation.

Functional programs often provide other advanced features, including non-strict evaluation, static type checking, full polymorphic type systems and higher order functions, all adding to the expressivity of the language. These are often the features most difficult to implement well on conventional hardware.

2.2 Graph reduction

As described earlier, execution of a functional program involves evaluating a distinguished entry function, in the presence of other functions. This model of execution can be described by graph reduction. The code in the body of a function corresponds to a sequence of reduction rules, specifying how the graph is to be modified. For example, a `let` expression causes new nodes to be added to the graph, defining a subexpression. Expression evaluation causes the graph to be reduced as a function node and its free variables are evaluated and replaced with their normal form. It should be clear how these graph reduction features correspond to conventional behaviour, such as heap allocation and function body execution.

An overview of graph reduction primitive operations:

`let`	→	allocate an acyclic subexpression
`letrec`	→	allocate a cyclic subexpression
`case`	→	evaluate expression, scrutinise the result node, taking a branch
function application	→	replace with instance of body
constructor	→	heap allocate ("box") arguments when saturated
primitive operators	→	perform operation immediately

Normal Form

When an expression is irreducible, it has reached normal form. The most useful variety of normal form is weak head normal form (WHNF). An expression is in weak head normal form if no reducible expressions (redexes) exist at the top level of that expression. This is useful for data structures such as lists. We can then access the head of a list without evaluating all elements of the list.

Free variables

A function binding has both arguments (parameters) and free variables. A free variable is a value used during the execution of the body of the function that is not a formal parameter. STG code identifies and makes explicit the free variables of a function, enabling efficient code generation. An example:

```
f a b = let
            g = a + b
        in  g * g
```

In the above example, function `f` has two arguments, `a` and `b`, and a free variable, `g`.

Strictness

Strictness is a property of a language describing the order in which expressions are evaluated. Languages can be divided into either strict or non-strict. Strict languages require all arguments to a function to be evaluated fully before that function can produce a result. Machine instructions have this property, for example. Non-strict languages allow arguments to be passed unevaluated, and

the function may still produce a result. It is interesting to note that the if-then-else expression common to many languages is essentially a non-strict function of three arguments: either the second or third arguments will be evaluated fully depending on the value of the first argument.

Strictness corresponds to *call-by-value*. Arguments are evaluated fully, to normal form, before function application. It is the resulting value that is passed to the function. As a result arguments may be evaluated unnecessarily if they do not contribute to the result of the function. Call-by-value is common in classical imperative languages.

As an aside, *call-by-reference*, where a pointer to an argument is passed to a function, is another common form of argument passing used in imperative languages. It could be thought of as a way to pass certain kinds of arguments unevaluated. It has no clear correlation in functional programming and its non-strictness is really limited to function pointers. It is a *way* to implement non-strictness.

Non-strictness is usually linked with *lazy evaluation*. In a lazy language, a function body can begin execution before the the arguments have been evaluated. An argument is only evaluated when the function requires the argument.

If arguments to functions are passed unevaluated two possibilities arise: the argument could be evaluated repeatedly, every time it is passed to a function; or the argument could be overwritten with its normal form after the first evaluation. This kind of update is fine as we know that the expression will produce the same result every time it is evaluated. The first method of argument passing is *call-by-name*, and the second is *call-by-need*. Clearly, call-by-need is to be preferred.

Unfortunately, call-by-need non-strictness is hard to implement efficiently on the JVM, although performance is good on a normal hardware [9]. Standard Java virtual machines have no mechanism to physically overwrite an object with its evaluated form, which is usually the most efficient way to perform an update.

2.3 Abstract machines

How do we compile a high-level language to executable code such that we maintain the semantics of the language? The problem is that high level functional code is seems to be quite different to conventional machine code, or even to C. It can be hard to see or reason about the translation to low level code. A standard technique has been to define an *abstract machine* capable of executing a small assembly-style language. The high level language can be defined in terms of the assembly language. The semantics of the assembly language can be defined in terms of its operation on the abstract machine. Finally, the abstract machine can be implemented as a set of data structures and runtime routines, instantiating a graph reduction system on real hardware. Peyton Jones' "Stock Hardware" paper [9] is the key reference for this kind of compilation technique, in the context of the Haskell language, and there is a depth of other literature on this subject. An innovation of the Peyton Jones paper is to define the assembly language as a small functional language: STG code.

STG code is executed by the Spineless Tagless G machine, but this is not the only abstract machine on the market. In recent years a number of abstract machines have proved useful in implementing Haskell compilers for the Java

virtual machine. The main alternative to the STG machine, though, has been some variety of the Chalmers G machine.

The G machine

The G machine was developed by Augustsson and Johnsson in the 1980s. An implementation of this system is described in [11]. The G machine is important in that it introduced the idea of compiling code that would instantiate a graph reduction system, rather than having an interpreter do the job. The G machine executes an intermediate language looking somewhat like an assembly for graph reduction. It is relatively easy to compile functional code to these instructions, and then to generate real machine code.

An early Haskell to Java compiler, that of Wakeling [24], used the G machine. This implementation is often cited, although its performance has been bettered. The work does, however, anticipate many of the problems with the JVM that have been confirmed in later compilers.

The Spineless Tagless G machine

A new step in the compilation of non-strict functional code comes with the invention of the Spineless Tagless G machine [9, 15]. As explained earlier, the STG machine executes a small *functional* language. This language, STG code, has both the normal denotational semantics of a small λ-calculus language, but also an operational semantics in terms of the STG abstract machine.

The full operational semantics of STG code on an STG machine are formally defined in [9], a quick outline is presented here. The mapping of the STG machine itself onto conventional hardware is described informally via translation to C in the same paper. A key benefit of the STG machine model is that it maps neatly onto the features provided by conventional hardware. This neat mapping from abstract graph reduction machine to concrete machine makes implementing a functional language backend for a new architecture clearer and easier.

The abstract STG machine has a state of five components [9]:

- the code
- the heap
- the global environment
- an argument stack
- a return stack
- an update stack

The heap contains *closures*. Closures are much the same as Java objects. They represent bindings of variables to both data and function values. Delayed computation is also implemented with closures. The STG machine is elegant in that both data and function values have this uniform representation as a closure object.

The global environment is the outer scope of the symbol table. It holds the addresses of statically allocated closures, corresponding to the global bindings of the program.

The argument stack holds the values of the arguments of functions. Values are either primitive unboxed data types, or heap addresses.

The return stack holds the addresses of pattern matching code (a continuation) to jump to after evaluating a `case` expression.

The update stack is used when evaluating the code of an updatable closure. It holds update frames saving the environment of closures potentially to be updated. Updates occur when evaluation of a closure completes. In general, the evaluated closure is overwritten with its result, preventing redundant evaluation. There are exceptions to this, such as the case of a closure entered only once. There is no need to update such a closure with its value, as it will not be needed again.

In a concrete implementation the different stacks are merged for efficiency reasons, just as a conventional "native" stack has multiple uses.

Finally, the code aspect of the global state specifies four execution states. The machine can either:

- evaluate an expression
- enter the code associated with a closure
- return a constructor applied to its values
- return a primitive unboxed type

A return passes a value to some pattern matching code to be scrutinised. A branch will be taken depending on this value. This is a *continuation*; implemented as a switch block to inspect the type or primitive value of the returned data.

The initial state

The machine begins execution at the global function `main` with empty stacks. The global state is a list of top level function addresses. The closures for these objects will have been pre-allocated on the heap.

Application

To apply a function to its arguments:

```
g = f a b c
```

the arguments are pushed onto the arg stack, and then a jump (tail call) is made to the entry code for the closure representing the function, `f`. It is possible that the function may be partially applied; that is, without all of its arguments present. Currying is used for this. A function must be able to accept arguments one at a time, collecting them until fully saturated.

If the function is a primitive operation, not represented with a closure it must be applied with all arguments present (*saturated*). They are dealt with specially.

On entering the closure, after argument satisfaction checks, and stack and heap overflow checks, the machine evaluates the expression represented by the closure.

It should be noted that making a tail call to a function body through a code pointer is impossible on the JVM architecture. Emulating tail calls with method calls as efficiently as possible is important to a fast implementation, and is covered in depth in chapter 5.

`let(rec)`

`let` and `letrec` expressions cause closures to be allocated on the heap. `letrec` allows the definitions to be (possibly mutually) recursive. After allocation, evaluation continues in the modified environment. The `let` expression binds a variable to an expression. A simple pseudo-STG code example: the `let` expression below allocates closures for variables `x` and `y`, for use in the expression `x + y`. When evaluated `a` has the value `10`.

```
a = let
        x = 1 + 2
        y = 3 + 4
    in x + y
```

The recursive case is distinguished from the normal `let` to allow the code generator to optimise recursive bindings:

```
loop n = letrec
            ns = cons n ns
       in ns
```

The translation of `let` and `letrec` constructs to JVM constructs is generally straightforward, as they correspond neatly to dynamically allocating Java objects.

`case`

The `case` expression is where expression evaluation and pattern matching occur and where branches are taken. Consider the pseudocode expression:

```
toupper c = case (islower c) in
                True  -> c - 'a' + 'A'
                False -> c

islower c = ('a' <= c && c <= 'z')
```

To execute this code, first first the value of `c` is saved on a stack for use in the case branches. The address of the pattern matching code for `True` and `False` (of the continuation) is pushed onto the return stack. Then a jump is made to the `islower` closure with an argument of `c` on the stack. This function is evaluated. The resulting value is returned to the continuation to inspect. One of the alternative branches will be taken depending on this scrutinised value.

The continuation part of the `case` expression embodies the imperative language if-else block in the STG language. This is the only place in the language where branches are taken.

The `case` expression requires close attention in the compiler. It exercises the implementation of both argument passing and environment saving. It requires a good return convention for both boxed and unboxed values, and somehow we have to jump back to a previously-saved continuation. The pattern matching code needs to be able to compare the types of boxed arguments, as well as to inspect their primitive values. All these aspects are covered individually in chapters 4 and 5.

Updating

Updating is the process by which we avoid unnecessary recomputation in a lazy language. It is essential to an efficient implementation. It also produces a whole set of design problems.

The idea is that on entering an updatable closure, the current argument and return stacks are saved. When evaluation of the closure's code is complete the update is triggered, and the closure is overwritten with the evaluated form, be it a constructed value, literal or function value.

Partial application is a problem in this situation. In this case a function returns another function of smaller arity. This requires us to compile code for each application of the function for different numbers of arguments.

Updating causes all sorts of inefficiencies on the Java virtual machine, as there is no way for us to physically overwrite a closure as we would usually like to do. Updating is discussed in depth in the design chapter.

Summary

While functional languages, such as Haskell, may have different syntax and semantics to those of imperative languages, it is clear that their execution can be described in terms of a conventional machine. Some of the advanced features of Haskell need special consideration in the compiler design. The features of particular interest are the non-strict evaluation model, with the updates that requires, as well as how tail calls, higher order, curried functions and pattern matching will be implemented. None of this is any worse than implementing a garbage collector or some other advanced feature in an imperative language, and we gain the well known clarity and simplicity associated with functional languages. However, first it is necessary to describe the particular technology involved in this project.

Chapter 3

Overview

The motivation for compiling Haskell to Java is the benefit to be gained in compiling functional code to a fast virtual machine with a standard bytecode representation and full hardware and operating system independence. There has been a growing view that the JVM is too tied to the Java language to be a truly multi-language machine, but an explicit description of the limits of the JVM will inform future implementations of more language-independent virtual machines and bytecode designs.

This section describes how the various pieces of software used in this project fit together. In particular, we look at the path from Haskell code, which is transformed by GHC to an intermediate language, STG code, which in turn is compiled to code that will execute on the Java virtual machine. An overview of the JVM and GHC is included, along with a discussion of the STG intermediate language.

The full compilation route from Haskell to hardware is described in Figure 3.1:

- Haskell is parsed and typechecked by our GHC frontend
- It is then desugared into a kernel language: Core
- Core is optimised, annotated and simplified, to become Spineless Tagless G code (STG code)
- This STG code is dumped in a human-readable form to `stdout`
- The external backend, the `stg2jvm` compiler, reads and parses this STG code, before transforming it to an abstract Java representation
- Java is then written out in multiple source files
- At this point a conventional Java compiler can compile the Java, and link in our Java runtime support routines and primitive operations, as well as previously built versions of Haskell libraries
- This produces a series of bytecoded class files, ready for execution on any Java virtual machine

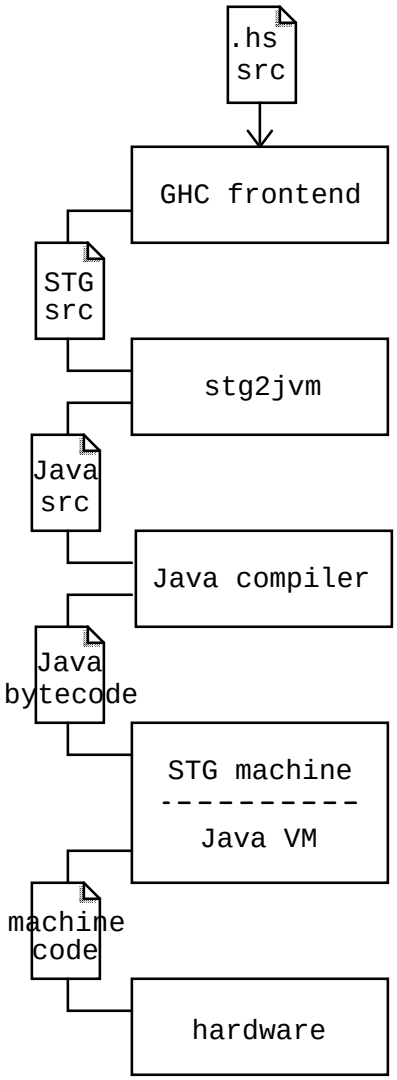

Figure 3.1: Compiling Haskell to the JVM

- The JVM loads the necessary classes, and the code is executed by the hardware

We concentrate on the intermediate backend, that is, the mapping of the STG language to Java code, and the mapping of the STG machine to the Java virtual machine.

3.1 Java and the Java virtual machine

Java programs are divided into classes. Classes describe heap allocated objects and associated with each class is a list of methods. A method is simply a function associated with a class. Methods are compiled to machine code held in the *method pool*, which corresponds to the text segment in a conventional object file. Constants and literals referenced by methods are similarly stored in the *constant pool*, corresponding to the data segment.

An object, in the Java sense, is a heap-allocated instantiation of a class. Pointers are restricted to being references to heap allocated objects: there are no code pointers available to the programmer, for example. Pointer arithmetic and casting of references to other primitive data types is illegal. A garbage collector runs to free unused objects. In addition to the heap-based objects, Java also provides the usual number of unboxed primitive types: ints, chars (16 bit unicoded), doubles and others. An interesting feature is that strings are immutable, and are not simply character arrays.

The JVM is a stack machine. Each invoked method has a stack frame, similar to the native C stack. Along with local variables and return addresses, the JVM stores expression operands on the stack, as computation is performed. There are no explicit operand registers.

Java source code is usually compiled to bytecode format, this in turn can be interpreted, compiled to machine code statically, or compiled just-in-time. Java code is usually verified at runtime by the JVM to ensure the it is well behaved.

One of the clear advantages of targeting the JVM is the machine-independent bytecode object file format. Additionally, there are many JVM implementations, most of which are freely available.

Limitations

A consideration when compiling for the JVM is what form of "machine code" we produce. Compilers for Scheme and Standard ML have written bytecode class files directly [1, 3, 19]. Compilers for Haskell and Haskell subsets and supersets, have produced Java source files [2, 5, 17, 21, 23–25] and there has been at least one Haskell compiler producing an assembly, textual form of bytecode [22].

The key reason to compile to bytecode, or Java assembly, is that Java bytecode has a `goto` instruction allowing a direct jump with a 16 or 32 bit offset inside a method body. This is not available at the Java language level. The restriction that the jump must be within the local method body has, however, been enough to convince people it is not worthwhile compiling to bytecode, as the `goto` cannot be used as a general tail call instruction.

There is one benefit though: the `goto` can be used as the jump for proper recursive tail calls. This is used by Serpette and Serrano [19] in their Scheme

compiler backend. This is an optimising compiler, and utilising the `goto` for this purpose is worthwhile. However, there are general solutions to the the tail call problem and the benefits to be gained by compiling to bytecode are otherwise few. For this reason, this project compiles to Java source, rather than bytecode. It should also be noted that a local `goto` can be achieved in high-level Java using a labelled loop block, with judicious use of `continue` and `break`.

Another possibility at the bytecode level is creating loops that modify that native stack. Such a trick allows us to emulate a recursive function call, passing arguments on the native stack, without performing any unnecessary register saves or stack frame allocations. It has been noted that while such code would be valid Java bytecode, some JVM implementations cannot executed it. Non-standard, specification breaking and poorly implemented JVMs have often been revealed by the various functional language compiler projects. Poor JVM implementations are a continuing problem.

The JVM implements a number of load and runtime checks. Many of these checks are unnecessary when using a statically typechecked language like Haskell. The JVM performs both bounds checking on array access, and null pointer dereferencing tests. These are both unnecessary for standard Haskell code. If the JVM still performs such tests, performance will suffer. This issue has been noted by most papers in this area. A virtual machine with a configurable runtime system would be a good thing for multiple language support.

Disabling the bytecode verifier is also desirable in terms of performance. As we compile via Java source, we can be sure that our code will be verifier compliant and will not do anything unusual (although it may be then be optimised is unhelpful ways). Serpette and Serrano report that their Scheme compiler, which uses idiosyncratic bytecode constructs for performance reasons, often breaks the verifier. Modifying their compiler to produce verifier-compliant bytecode results in a 10% to 55% slowdown. If we were generating Haskell-optimised bytecode directly it could be assumed this would break the conventional verifier too.

3.2 The Glasgow Haskell Compiler

The Glasgow Haskell compiler is an optimising compiler for the Haskell language. The compiler itself is written mostly in Haskell. In this project, GHC acts as our optimising front end. By utilising an existing compiler as our frontend we get all the optimisations we want, as well as a correct parser and typechecker for full Haskell. The optimised code that we ultimately translate to Java is key to a successful implementation. Choi, Lim and Han [5] suggest that code optimised by GHC is around 5 times faster than unoptimised code when compiled for their Java STG machine implementation. Our project explicitly deals only with the optimised code produced by GHC, so hopefully the worst case of the `stg2jvm` compiler will be the best case of previous implementations.

Figure 3.2 illustrates the GHC frontend, and how this project, the `stg2jvm` backend, slots in. It also illustrates the other possible compiler targets available: to C and to assembly.

GHC parses full Haskell, then typechecks and desugars this into a Core language. Core is a small functional language to which all Haskell can be translated. This Core code is then optimised, before producing the STG code that

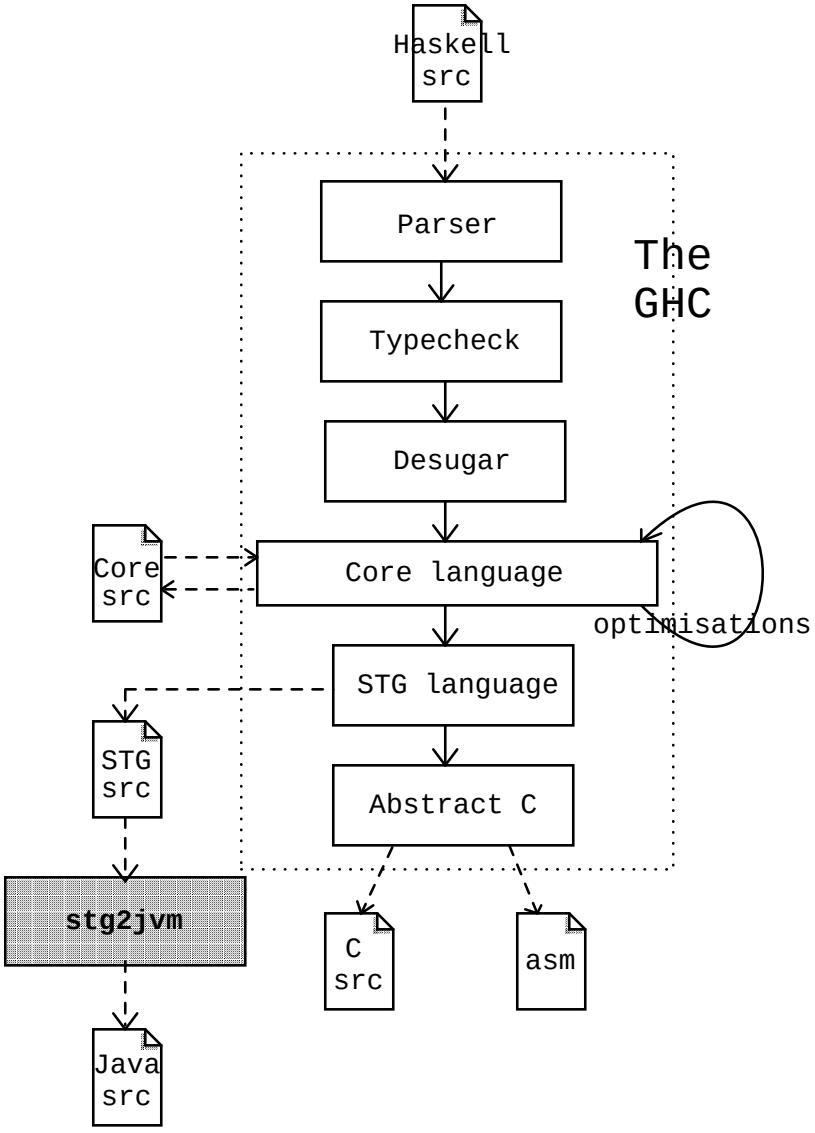

Figure 3.2: An overview of the Glasgow Haskell Compiler, along with the external `stg2jvm` backend

the external backend transforms to Java.

Normally GHC takes Haskell source files as input, but it can also read and write Core language files. The input Core potentially could be generated by an external frontend for another similar language or by an external optimising pass. After repeatedly optimising the Core language it is reduced and annotated as STG code. GHC then translates the STG code into Abstract C, implementing the graph reduction abstract machine and its "machine" code in a pseudo-C. GHC's code generator then outputs either C or native assembly.

Core

As Figure 3.2 suggests, GHC provides two external intermediate languages: Core and STG code. These are both small functional languages. These intermediate languages allow us to write an external backend for GHC, rather than having to link our code into GHC, which can be quite time consuming.

Core is a well documented [20] language resembling an enriched λ-calculus. The key benefit of Core is that all higher level Haskell constructs are defined by translation to Core. It is effectively the kernel of the Haskell language and this allows a compiler writer to concentrate on optimisation and code generation for this small language, rather than for all of Haskell. Core is the semi-official intermediate language of GHC. It is less tied to any particular abstract machine design than STG code might be.

Core explicitly deals with unboxed values and primitive operations. In this way we can design optimisations on unboxed values, rather than resorting to ad hoc code generation. It differs from the (lower level) STG code in that it:

- provides a lambda abstraction expression
- maintains the Haskell module namespace separation, with Core programs consisting of a set of modules, currently not explicit in STG code
- provides `data` and `newtype` statements
- is explicit about type coercion
- identifiers may be exported with the `%external` keyword.
- has nice syntax for type abstraction
- has its grammar standardised
- can be fed to, parsed and typechecked by GHC. This is currently not possible with STG code

Core and STG code are much the same in other respects. They both use encoded identifier names, have a similar lexical and grammatical structure, as is described in detail in the next chapter, and are semantically similar.

We now present a detailed discussion of the STG intermediate language, which is derived from Core, and which is the input language for our new Java backend.

Chapter 4

Spineless Tagless G code

The Spineless Tagless G machine is an abstract machine useful as a target for a functional language compiler. The STG machine provides the link between high level functional code and that code's execution on real hardware. The evaluation system described by the abstract machine is implemented as a set of runtime routines. The runtime system has been outlined briefly sketched earlier, and a detailed examination of its implementation follows in the next chapter. For now we concentrate on the language executed by the abstract machine: STG code. STG code is an intermediate language produced by GHC, and executed by the STG machine. STG code is sometimes called the Shared Term Graph language to emphasise its independence from any particular graph machine implementation.

At the top level an STG program is just a set of function bindings, similar to a Haskell program. It differs in that STG code is a minimal language with little syntactic sugar. Below is the classic higher order function `map`, in Haskell and in a pseudo-STG code (which we may call STG-lite). We will focus on this example to explore the STG language in the following sections. In Haskell:

```
map f [] = []
map f (y:ys) = (f y) : (map f ys)
```

becomes the STG-lite binding:

```
map = [] \n [f xs] ->
    case xs of
        Nil []      -> Nil []
        Cons [y ys] -> let fy = [f y] \u [] -> f [y]
                           mfy = [f ys] \u [] -> map [f ys]
                       in Cons [fy mfy]
```

This function applies its first (higher order) argument to each element of its second argument (a list). STG-lite code is necessary to make things a bit clearer, as real STG code is a machine generated jumble, rather like reading compiler generated assembly code. A new parseable grammar for the real STG code GHC currently generates is given further on, and differences between it and STG-lite are discussed.

4.1 A simple functional language

Firstly, `map = ...` defines a top-level, or global, binding for a statically allocated closure. An STG program consists of a set of such bindings, including the entry point `main`. These bindings are global in scope, and as such can be called without a qualified name. Global bindings define either values or `constructors`. A constructor is a function for creating algebraic data types. It is similar, but not identical, to a Java constructor.

The capitalisation of the first character of the binding identifier is used to distinguish the kind of binding, just as in Core and in Haskell code. Values (either data or function values) must have identifiers beginning with a lower case letter (or `'_'`). Constructors begin with an upper case letter. Qualified names also begin with an uppercase letter, and are of the form `Module.field`.The module name is capitalised. The field can then be distinguished as either a constructor or value definition in the usual way. This convention is discussed in the Haskell 98 report [13].

Lambda forms

The right hand side of a global definition is a *lambda form*. The example above:

```
map = [] \n [f xs] -> case ...
```

defines a function value. The first three pieces of the lambda form are the free variable list, an update flag, and the formal argument list, respectively. In this example, `map` has no free variables, but has two formal arguments, `f` and `xs`.

The update field for `map` is a `\n`, indicating that the binding need not be updated. This implies that after evaluation it is not necessary to produce an update closure to replace the evaluated `map` closure, and thus it is possible to avoid generating update code. This is a good idea for a function like `map` that is called recursively, and which should be compiled to something approximating a loop.

The alternative is `\u`, an updatable expression, which should be somehow overwritten with its value on evaluation. Such a closure requires an update frame be allocated before entering the closure, to signal that an update must take place after evaluation.

Let(rec)

Lambda forms in STG-lite bind a variable to an expression. Expressions are either `let(rec)` statements, `case` blocks, function applications, constructors, or some variety of literal.

`let` statements, as we have seen, perform a local binding operation. They allocate a closure and bind it to a variable. Here is a real world STG `let` fragment translated to an STG-lite `let` binding:

```
f = [c] \u [] ->
    let
        cs = [] \u [] GHCziBase.unpackCString# "astringliteral"
    in
        elem c cs
```

The function `f` has a free variable in scope `c` (presumably a char). The `let` expression binds the identifier `cs` to a function value creating a string literal. The expression `elem c cs` is then declared. The `GHCziBase.unpackCString#` identifier is a compiler base library function for making a Haskell `[Char]` from a C-style string.

In our initial `map` example, one of the alternatives in the `case` expression branches to code containing a non recursive `let`:

```
Cons [y ys] -> let fy = [f y] \u [] -> f [y]
                   mfy = [f ys] \u [] -> map [f ys]
               in Cons [fy mfy]
```

Here, if the `case` evaluation returns a `Cons` closure the `let` branch is taken. The `let` causes two variables to be bound to local functions. The first, `fy`, to a function applying the argument `f` to one element (`y`) of the list we are mapping `f` over. The second binding, `mfy`, describes a function that calls `map` over the rest of the list. These two bindings are then passed in unevaluated form to the `Cons` call.

Case

`map` employs a `case` construct to perform the pattern matching used in the original Haskell code to distinguish the empty list.

```
case xs of
    Nil []      -> Nil []
    Cons [y ys] -> let ... in Cons ...
```

As stated earlier, everything happens around the `case` expression. In the above example `xs` is evaluated, returning a boxed value, which is taken apart by the pattern matching code. Depending on which constructor was used one of the `case` branches is taken and another expression results.

In an implementation of `case`, an integer tag can be used to represent the particular constructors of a data type. This tag can be loaded into a register (or global variable in Java) on return of the `case`'s evaluated expression, and then used as an offset to a jump table pointing to the case alternatives.

A more Java-like way would be to use `instanceof` in an if-else block to select among branches. This has, however, been found to be more inefficient than the integer tag method, due to slow implementations of `instanceof` [17, 21] in the JVM. Of course, if there is only one pattern to match the alternative can be executed immediately.

Other cases

Evaluation of a `case` may also return primitive, unboxed values. Pattern matching must then be done on this raw value, and the STG language distinguishes such values in the grammar:

```
case e of
    MkInt x# -> e1
    MkInt y# -> e2
```

Here, `MkInt` is a constructor to box a primitive integer, recognisable by the `#` char appended to the variable name. Clearly, the inspection of unboxed values can be implemented in Java using switch blocks, which operate over primitives, and most of the STG primitive types map well to Java primitive types. Vectored returns, for small integer primitives, could be implemented this way.

In this project, as we compile to Java source, we rely on the Java compiler to produce efficient bytecode. There is some evidence available to suggest that Java machines tend to perform poorly when executing the kind of `tableswitch` bytecode instructions we hope will be generated [19] to emulate true vectored returns. This is problematic unless we can be sure the Java virtual machine implementation is up to scratch.

Unboxed data types are built-in to the STG compiler, and a variety of fundamental *primops* do things with such values. Case expressions are the only place where unboxed objects can be dealt with directly. They can be passed on to the case's alternatives and manipulated using primops. Let expressions only ever bind boxed objects; they are just for heap allocations. A full discussion of the issues relating to primitive operations follows in the next chapter.

Application and construction

Constructors and application are both expressions, appearing in the grammar for STG-lite as `var atoms` and `constr atoms`. As pointed out earlier, these similar operations are distinguished by the capitalisation of the first letter of the identifier. Here are some real, qualified, function calls:

```
SystemziIO.putStrLn [s]
SystemziTime.getClockTime [t]
```

and some constructors:

```
GHCziBase.True
GHCziBase.I# [4]
Network.PortNumber [lit]
```

`I#` is a constructor for boxing primitive integer types (data value of type `Int#`).

Constructors, of course, are used to construct data types. It is required that constructors are saturated when called, and that arguments to functions and constructors are simple variables.

It is important to note, too, that function application is suspended in general, and is only forced in the evaluation step of a `case` expression. However, primitive operations are not suspended.

4.2 A not so simple functional language

A summary of STG-lite (derived from [9], and modified to be closer to the STG code currently produced), is presented in Table 4.1. Full STG code is less human-readable, littered as it is with primitive operations, as well as unique and encoded names. It is generated by GHC using the "`-ddump-stg -dppr-debug`" flags. The syntax can be slightly odd and has developed over time. It can still be parsed, though, and this grammar is described in Tables 4.3 and 4.2.

program	→	*binds*
binds	→	var_1 `=` lf_1 `;...;` var_n `=` lf_n
lambda-form (lf)	→	$vars_f$ `\`π $vars_a$ `->` *expr*
update flag (π)	→	*u* | *n*
expr	→	`let` *binds* `in` *expr*
	|	`letrec` *binds* `in` *expr*
	|	`case` *expr* `of` *alts*
	|	*var atoms*
	|	*constr atoms*
	|	*prim atoms*
	|	*literal*
alternatives	→	$aalt_1$ `;...;` $aalt_n$ `;` *default*
	|	$palt_1$ `;...;` $palt_n$ `;` *default*
algebraic alt	→	*constr vars* `->` *expr*
primitive alt	→	*literal* `->` *expr*
default alt	→	*var* `->` *expr*
	|	`default ->` *expr*
literal	→	`1` | `2` | `...`
	|	`...`
primops	→	`+#` | `-#` | `*#` | `/#` | `...`
	|	`...`
vars	→	`[` var_1 `...` var_n `]`
atoms	→	`[` $atom_1$ `...` $atom_n$ `]`
atom	→	*var* | *literal*

Table 4.1: A grammar for STG-lite

Although a number of projects have used STG code in their compilers, as far as I am aware, this is the only presentation of the actual grammar.

STG code is annotated specifically for compilation to GHC's STG machine implementation, but it is not necessarily limited to this machine. There is nothing in the language that ties it to the STG abstract machine. STG code uses encoded identifiers, and an explanation of this encoding follows.

Bindings

An STG program consists of a set of top-level bindings, just like STG-lite and Core, with the addition that recursive bindings are explicitly identified and wrapped in `{- StgRec (begin) -}` and `end` flags. It is useful for us to then distinguish constructor bindings from variable bindings at the top-level, unlike in STG-lite where the differences are implied.

Binding information

The right hand side of a binding is further annotated in real STG code. Along with the free variable list and the argument list, further update flags are distinguished. The update flags are now `\u`, `\r` and `\s`. The semantic differences between these flags can be a bit subtle, and are discussed in depth in appendix 5 of [9]. Briefly, `\u` still means the closure should be updated, as discussed previously. The no-update flag is now divided into either reentrant (`\r`) or

single-entry (\s) flags. These both indicate closures that do not need to be updated on evaluation and a variety of extra optimisations are available to the single entry closures, mostly to do with special garbage collector treatment in GHC. Constructors and partial applications are always flagged reentrant. An example reentrant function binding:

```
s = [] \r [] showsPrec t;
```

Two additional fields are added to the lambda form information. A binder info flag (bi) and cost centre information. The binder info flag is usually empty, but if the compiler can prove that a function is always saturated with its arguments, then the sat-only flag is set. Argument satisfaction checks on entering the closure can be omitted, so a sat-only function can be jumped to directly. This can be implemented in Java by providing a special sat_entry method for fast closure entry.

The cost centre information is used when generating profiled code. In GHC this specifies that additional information, a cost centre stack, is added to closures. There are a number of different CC flags and they are used collect profiling data for the various kinds of closures: thunks, functions, locally defined, and even static closures. On entering a closure the CC data is updated. A full discussion is not relevant to this project, and the profiling flags are not sued in the stg2jvm backend.

Lambda forms

The simple representation of all top-level bindings with a lambda form has expanded in STG code. The -> tag is removed, and instead right hand sides come in the following forms. Firstly, constant applicative forms (CAFs):

```
Main.zdmain = [] \u [] GHCziTopHandler.runIO main1;
```

CAFs are bindings with no arguments. The binding above is the entry point function for STG programs, and its result is the program's value. CAFs allow a variety of optimisations, for example, they can be statically allocated.

The next variety of binding is the constructor. A constructor binding has no free variables and has a capitalised identifier (ignoring encoded names for a minute). An exclamation mark is also added to the data type member identifier. For example:

```
Tokens.zdfEqPovToken = NO_CCS GHCziBase.ZCDEq! [zsze zeze];
```

or this constructor for boxing a primitive char:

```
a = NO_CCS GHCziBase.Czh! ['z'];
```

Constructors are all reentrant bindings, non-updatable, and the update flag is omitted from the definition.

Finally, the general binding case, in which arguments are passed to the closure, as in the following fragment:

```
safe = sat-only [] \r [x d ds]
    case ds :: Just GHCziBase.ZMZN of wild {
        -- lvs: [ds x d]; rhs lvs: [x d]; srt: (0,8)
        GHCziBase.ZC q l -> let {
                                sat_s1c6 = [x q d l] \u []
                                ...
```

The case expression above is discussed in the next section.

Case expressions

The above example illustrates a reentrant binding of a more complex expression. The `case` expression in full STG code is extended from that of STG-lite as follows:

- The expression to be evaluated is tagged with its type. In the above code this is `(Just GHCziBase.ZMZN)`, which decodes to `(Just GHC.Base.[])`. This is a list.
- the value resulting from the case scrutinee (the expression being evaluated) is bound to a variable declared after the '`of`' keyword
- Algebraic alternatives are all labelled with a data constructor of that algebraic type
- The number of variables must match the constructors arity
- Primitive alternatives, over unboxed types, must be distinct primitive values of the same type
- Braces and semi colons are added to the case block to make it easier to parse
- The alternatives in the `case` are annotated with *live variable* lists, indicating which variables in the local environment can be referenced directly on the stack. The grammar currently does not parse this information.
- Along with live variable information, sprinkled throughout the code are static reference table (*srt*) notes, such as (`srt: (0,8)`), indicating whether the current function references any statically allocated CAFs. This information is ignored in this project and is not in the current grammar.

Some examples:

```
case f s lit :: Just GHCziBase.Bool of wild2 {
    GHCziBase.True  -> SystemziIO.putStrLn s;
    GHCziBase.False -> usage1;
};
```

In the above case the scrutinee is bound to `wild2` and results in a boxed boolean value. This value is inspected and either a print statement results, or a usage function is called. The next example illustrates the use of unboxed values.

```
case bzh :: GHCziPrim.Intzh of wild2 {
       4       -> GHCziBase.True  [];
       DEFAULT -> GHCziBase.False []
};
```

The expression bound to the variable `bzh` (decoded to `b#`) is evaluated, producing a primitive integer value. The case body then tests to see if the value is equal to `4`. The entire case expression evaluates to a boolean result. The default case is equivalent to the wildcard pattern in Haskell. A good compiler implementation might return the scrutinee value in a register, with no boxing necessary. The code is then much as efficient as an imperative boolean test. As an aside, for some odd reason in the grammar, a default expression is not terminated with a semicolon.

Let expressions

The `let` expression in STG code differs from that of STG-lite in the following ways:

- The bindings are wrapped in braces
- Recursive bindings are flagged in the same way as global bindings
- An new form of `let` is added: `let-no-escape`.

The following code illustrates the use of the `let` expression:

```
zsze = [] \r [a b]
 let {
   sat_s733 = [a b] \u [] zeze a b;
 } in GHCziBase.not sat_s733;
```

`zsze` decodes to `/=`, so this is a binding for the "not-equals" function. The `let` statement allocates a closure for the expression `(a == b)`. This code is then passed to the library function `GHC.Base.not`, hence defining `/=`.

The next example illustrates a list indexing operation. The `let` expression boxes up the primitive integer `1`, which is then used as an index into the `argv` list. Note that this is unoptimised code! In a good implementation a range of small primitive integers are pre-allocated, and constructor calls to box such integers will instead return references to a share static object.

```
let {
    sat_s3Gol = NO_CCS GHCziBase.Izh! [1];
} in GHCziList.znzn argv sat_s3Gol;
```

`let-no-escape` statement is a special form in which the compiler guarantees that a variable can be used without a full closure creation for the expression it is bound to. A variable, `x`, can be let-no-escaped if all variables needed in the execution of the `x` are live when `x` itself is live. Liveness is an operational property a variable has if it can be directly accessed on the stack. A let-no-escaped variable is then simply represented as a code pointer and a stack pointer.

The grammar

The grammar in tables 4.2 and 4.3) for the STG intermediate language was generated directly from the parser this project uses for the language. The full language is essentially the STG-lite language with the addition of the features mentioned previously. The lexical structure is that of the Haskell, except that identifiers have been Z encoded. This encoding is explained in the next section.

qtcon	→	*modid* . *tcon*
qvar	→	*varid*
	|	*modid* . *varid*
qcon	→	*modid* . *conid*
var	→	*varid*
tcon	→	*conid*
varid	→	*VARID*
conid	→	*CONID*
modid	→	*CONID*
lit	→	*INTEGER*
	|	*FLOAT*
	|	*CHAR*
	|	*STRING*

Table 4.2: STG literals and identifiers

```
program  →  binds

binds    →  binds bind | ϵ

bind     →  binder
         |  {- StgRec (begin) -}
            binders
            {- StgRec (end) -}

binders  →  binders binder | ϵ

binder   →  qvar = rhs ;
         |  qcon = rhs ;

rhs      →  cc bi [ vars ] flag [ args ] expr
         |  cc qcon ! [ args ]

cc       →  NO_CCS | CCCS | CCC_OVERHEAD
         |  CCS_DONT_CARE | CCS_SUBSUMED
         |  ϵ

bi       →  sat-only | ϵ

flag     →  \u | \r | \s

expr     →  let { binds } in expr
         |  let-no-escape { bind } in expr
         |  case expr :: tdef of var { alts def }
         |  qvar [ args ]
         |  qcon [ args ]
         |  qvar args
         |  qcon args
         |  lit lit

alts     →  alts alt | ϵ

alt      →  qcon args -> expr ;
         |  lit -> expr ;

def      →  DEFAULT -> expr | ϵ

args     →  args arg | ϵ

arg      →  qvar
         |  qcon
         |  lit

vars     →  vars var | ϵ

tdef     →  tcon qtcon
         |  qtcon
```

Table 4.3: Real-world STG grammar

4.3 The Z encoding

GHC encodes identifiers for use internally with the 'Z encoding'. The encoding is used in GHC's `.hi` files, assembly code and the Core and STG interfaces, and internally. Non-alphanumeric characters are encoded so that all identifiers become valid C identifiers, as well as valid Haskell identifiers.

Normal Haskell identifiers have few restrictions. Identifiers can consist of a wide range of non-alpha characters, as well as normal C identifier chars. This makes parsing identifiers difficult. The Z encoding to make things easier for the parser, but is pretty human-unreadable in large volumes.

The characters 'z' and 'Z' are used to introduce escape sequences. The two cases are used to maintain the distinction between variables and constructors. Table 4.4 describes the mapping:

Character	Code
Tuples:	
(##)	Z1H
()	Z0T
(,,,)	Z3T
Constructors:	
(	ZL
)	ZR
[	ZM
]	ZN
:	ZC
Z	ZZ
Variables:	
z	zz
&	za
|	zb
^	zc
$	zd
=	ze
>	zg
#	zh
.	zi
<	zl
-	zm
!	zn
+	zp
'	zq
\	zr
/	zs
*	zt
_	zu
%	zv
c	znnnU
	where nnn *is a decimal*

Table 4.4: The Z-encoding

Chapter 5

Spineless Tagless Java

Compiling functional languages to Java presents a variety of problems, as described previously. The key issues are the implementation of tail calls, lazy evaluation, continuations and updates. This chapter presents an investigation of the design of a Java mapping of the STG language, as well as comparisons to similar projects. The design and implementation features to focus on are:

- closures
- lazy evaluation
- continuations
- tail calls
- updates
- the runtime system

To begin with we look at the major representation decision: how to implement closures.

5.1 Closures

The STG machine represents function and data values as heap allocated *closures*. Delayed computations, *thunks*, are also represented by a closure object. It is a tagless machine for this reason: no tags are required to distinguish thunks from values.

A closure, at its most basic, is a structure consisting of a code pointer and a list of free variables. For the function:

```
f = [a b] \u [] add a b;
```

the code pointer of the closure points to code to execute the function body (the `add` application). The free variables, `a` and `b`, are part of the closure object.

To execute the function the closure is entered. This means we jump to the function code pointed to by the closure's code pointer. On a Java machine this implies calling a distinguished method of the object. It may be desirable to provide a variety of entry points to the closure code, such as the case in which

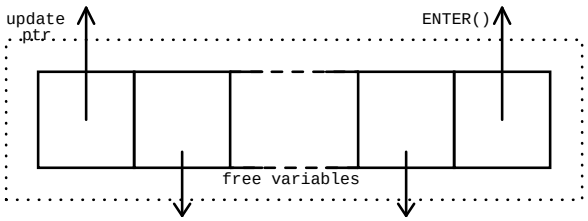

Figure 5.1: A Java Closure implementation

we know that a function is applied with all its arguments available. The closure structure is, then, much like an object in the Java sense of the word.

A data value is just a special variety of function value. It can be represented as a closure whose entry code immediately returns the data it contains. This representation can be optimised to provide direct entry to the closure's value, avoiding unnecessary argument checking and the like.

Non-strict languages usually pass arguments to functions in unevaluated form. As we have seen, expressions are only ever evaluated (or *forced*) inside `case` expressions. Elsewhere they remain suspended. A thunk, considered abstractly, is a function value representing the suspended expression. When the value of the expression is required the thunk is *forced*, evaluating it. The thunk is then updated with the value of the expression, saving unnecessary recomputation. Thunks, clearly, can be represented as closures too: they are an updatable, zero arity function value.

Java closures

The first step is to recognise that the closure structure required by the STG machine has an obvious representation as a heap-allocated Java object. They are equivalent in most aspects.

All Haskell to Java compilers seem to have made this representation decision. The code pointer in a conventional closure, however, must be emulated with a method. Figure 5.1 presents a closure in the JVM environment (the update pointer is necessary for our tail call implementation, as is discussed later in the chapter).

There is an obvious inefficiency in this Java representation: we cannot make a direct jump to the code body of the closure, but must make a method call. In general this will require register saves and stack manipulation. Hooks to modify such default behaviour would be a nice feature, if extending the JVM for true multi-language support (similar to some of `gcc`'s flags).

All closures are implemented as subclasses of the `Closure` superclass. Each binding in an STG program is translated to a Java class extending this superclass. The superclass specifies the standard layout of the `Closure` object and individual closures can override the free variables and entry methods of the superclass.

A problem with this design is that even small programs can generate quite large numbers of classes. Any reasonable program will have hundreds, even

thousands, of bindings, many of them very similar. This is noted by Wakeling in his G machine compiler [24]. The SML compiler, MLJ [1], solves this problem ingeniously. Closures are shared. The idea is to use a single closure for all bindings with the same kind of free variables, but with different function types. So a single closure object will have multiple entry methods, corresponding to the different functions sharing that closure.

An alternative design would be to compile entire STG modules to individual classes and to compile global functions to members of this module class. However, we would still have to implement higher order function values as instances of a closure object, requiring their own Java classes.

Below is some example code generated for the `Closure` superclass. The `enter()` method has a dummy return value that must be overridden by any instance binding. All closures need to import the `runtime` package, which provides the various STG runtime support routines and data structures.

```
package runtime;

class Closure {
    public Closure ind = this;
    public Closure enter() { return this; }
}
```

A particular binding instance might look like the following Java code. This particular example shows an implementation of a boxed primitive integer value. A data constructor to box a primitive can use the `init()` method as the constructor for that class, which is a nice correlation. Recursive bindings prevent us from uniformly using automatic Java constructors to initialise closures.

```
class BoxedInt extends Closure {
    private int value;

    public BoxedInt() {}

    public init(int i) { this.value = i; }

    public Closure enter() {
        Runtime.ireg = value;
        return this;
    }
}
```

All closures require an `enter` method. The `ind` reference is used for updates. We now present a detailed look at how different binding varieties map to closures.

Function and data values

Functions are simply represented in the closure model. They are compiled to a method of a closure class. Function evaluation corresponds to calling an instance method of an object. The standard entry code arranges for argument satisfaction checks, before jumping to code implementing the function body. The function body implements the operational semantics of STG code, discussed

previously. For example, a `let` expression will cause heap allocation; a `case` expression will force expression evaluation. A reference to the return value is passed back to the calling method. Free variables required by the binding appear as instance fields of the closure. Their representation is discussed below. Data values (normal forms) compile to closures whose entry code simply returns the data they represent.

5.2 Thunks

Implementing full laziness requires some technique to capture and suspend a computation, as expression bindings must be passed around unevaluated. The object that captures the suspended computation is the thunk. To evaluate a thunk is to *force* it. The thunk must then be updated with its evaluated form.

Two main thunk representations are useful: cell mode and closure mode [9]. Tullsen [21] provides a nice summary of the two representations. Cell mode is equivalent to the following Haskell algebraic type:

```
data Thunk a = Evaluated a | Unevaluated (() -> a)
```

Forcing a thunk in cell mode causes `Unevaluated` to become `Evaluated`.

The alternative is closure mode, where the thunk is a parameterless function:

```
data Thunk a = () -> a
```

Forcing a closure-mode thunk returns the value of the function. Such a thunk has state as it must become and remain evaluated after the initial force. This is, of course, a little tricky on the JVM.

In closure mode a conventional implementation would physically overwrite the closure with the new value. Such an update cannot be performed on the JVM. We have to allocate a new closure representing the normal form of the thunk. Access to this update node is via an indirection from the original closure.

In cell mode, the original cell is updated with an indirection to a new cell representing the updated value. A closure mode that requires all updates to be accessed via an indirection is close to being cell mode, and it loses the efficiency gained by the overwriting convention.

Tullsen provides a benchmark of forcing time using both a cell mode implementation and a JVM-style closure mode implementation, and this reveals similar performance for the two models [21].

This project represents thunks as instances of the closure class in an identical manner to function objects. Forcing a suspended expression is just to evaluate a zero arity function, resulting in a value. The update, if it needs to be performed, requires allocating a new closure for the updated value and adjusting an indirection reference of the thunk object to point to its updated form. Updates are discussed further on.

5.3 Currying

We treat partial applications in much the same way. The possibility of partially applied functions in the language requires some technique to allow us to call a

function with less than the full number of arguments. Unless we know that a function is saturated when called, we must instead collect arguments one at a time until the function can be fully evaluated. This is currying.

The implementation of curried functions follows Tullsen [21]. A curried function collects its arguments by providing entry code to check the number of arguments on the stack, saving them until fully saturated, at which point the function evaluation can proceed. So:

```
f x y
```

becomes

```
(apply (apply f x) y)
```

This is inefficient for the majority of cases where known STG functions are called fully saturated. For these cases we provide a direct entry point, jumped to from the closure entry code. The function code can be directly executed with all arguments on the stack. It is only unknown functions that must be called in a curried fashion.

Globals

The top-level function bindings of an STG program must be globally visible, and their closures may be statically allocated. This is achieved by making references to the closure from fields of the universally imported `G` class (the class implementing the global environment of the STG machine for us). This follows Vernet [22] and others (although Vernet goes further, generating a distinct class of globals for each module. This means, that with the JVM's dynamic loading, on the class of bindings used by the current code are loaded, rather than the entire program at once).

Free variables

There are two main options for how to store the free variables of a binding in a closure. Either the variables are copied to the closure in a flat form, else a reference to a block or list of variables is stored in the closure. This later design trades space for time, as the variables may then be shared amongst a set of closures. The flat representation uses more space, but accessing the variables will be faster as they are available locally and directly. It is also a simpler design and is used in all known Java implementations.

Following Vernet [22] each closure requires an `init()` method to set the values of the free variables in the current environment. Recursive bindings prevent us from using the native constructor for the closure: if two bindings both have each other as free variables, they must both be allocated before the free variables can be set, hence the `init()` method.

5.4 Primops

Hundreds of primitive operations are defined for the STG machine, corresponding to low level operations available on the underlying hardware or to operations

controlling the STG machine itself. Most can be implemented in a few instructions, though some must be implemented as functions. Only a small number of primitive operations are implemented in this project.

Primops are not really first class values on the STG machine, as they cannot be delayed or passed around. Primitive data values are first class, as has been explained. Where primops appear they are generally executed inline and, if possible, primitive values are returned in global "registers" on the JVM, rather than boxing them. As shown previously, the STG language clearly indicates where this is possible.

Complex primitive operations can be declared as static methods of the global class `G`, rather like other globally scoped functions. Following [16] we can implement a nice optimisation. As they are guaranteed not to tail call, there is no risk a primop will cause a stack overflow, and so arguments to non-inlined primops can be passed on the native stack. Meehan and Joy [16] found 30-60% improvements in their G machine implementation with this technique, instead of the alternative of providing STG binding wrappers and hence heap allocated closures for primops.

Another useful trick, as used in the GHC, is to preallocate boxed closures for many primitive literals. They can be declared static and global in scope. Constructors attempting to box common values can instead return a reference to the globally shared pre-allocated constant. Serpette and Serrano [19] find that, after preallocating closures for the ascii characters, and integers in the range `[-100 .. 2048]`, only around 4% of arithmetic operations required values outside of this range, saving many dynamic heap allocations.

5.5 Constructors

The implementation of algebraic data types follows easily in this implementation, and has been discussed earlier in the STG continuation section. Rather than using Java's object oriented features, and `instanceof` to distinguish type members, an integer tag is associated with each type member. This allows efficient case analysis in pattern matching code. Each algebraic type is implemented as a unique Java class, and the tag distinguishes its members.

5.6 Continuations

A `case` forces evaluation of its scrutinee expression, whose result is inspected to select among alternative branches. For the following STG code:

```
case xs of
    Nil  []     -> e1
    Cons [y ys] -> e2
```

The closure for `xs` must be entered and evaluated. The result is returned via a reference on the return stack. In the algebraic alternative situation the integer tag of the type member is inspected to determine which branch is taken. This seems to be the most efficient way to perform this kind of pattern matching on the JVM. Pattern matching via inheritance with `instanceof`, though logical, has proven to have poor performance in existing JVM implementations

(although one might expect it to perform as well as the explicit integer tagging technique). The Mondrian [17], Bigloo [19] and SML [1] compilers all cite poor performance of the `instanceof` pattern matching technique.

Another pattern matching technique for algebraic alternatives involves the raising of `ClassCast` exceptions, and was considered by Tullsen [21]. If the scrutinee value is bound to the variable `x`, we can attempt to cast this value to each alternative type to establish which pattern it matches:

```
try {
    c1 = (NilClass)x;
    ... code of e1 ..
} catch (ClassCastException ex1) {
try {
    c2 = (ConsClass)x;
    ... code for e2 ...
} catch (ClassCastException ex2) {
...
}
```

Although this is elegant in the Java circumstance (though perhaps less so than the `instanceof` technique), once again raising exceptions produces performance penalties. Tullsen found this to be two orders of magnitude slower than switching on integer tags. An advantage of using integer tags is that it should be easy for the compiler to produce good `tableswitch` bytecode.

Unfortunately, [19] cites problems with some Java compilers producing non-constant complexity `tableswitch` operations, along with the anomalies in the `instanceof` operation. The existence of better implemented JVMs should make explicit tagging redundant. The problem seems to be that the Java machines have been tested and optimised to generate the kind of code required for the Java language, and little testing is performed on the relatively rare constructs created for functional languages. This is a pity.

So although the compilation by Java compilers of integer tag switches in sometimes poor it still seems to be the best alternative. It also allies closely with pattern matching on primitive unboxed types, which usually can be implemented directly with Java switches.

Saving the environment

A `case` block forces evaluation of its scrutinee. It is not known how complex the scrutinee expression is, and it is possible that it will in turn evaluate a `case` expression. Some mechanism to save the local environment of the `case` must be implemented in order to maintain the values of free variables and to pass them on to the alternative branches of the `case` block.

The STG machine requires that before entering the closure of the `case` scrutinee we push a continuation on the return stack. In this way, when the scrutinee expression is evaluated, control can be returned to the closure on top of the return stack. We can do this simply by pushing a reference to the current closure onto the return stack, and then entering the scrutinee.

The scrutinee can then return control to the closure pointed to on the return stack. A flag can be set in this closure so that the `enter()` method jumps

straight to the pattern matching code for the `case` expression we were evaluating.

5.7 Tail calls

The implementation of tail calls is crucial to the performance of a functional language on the JVM. Ideally, closures are entered by jumping directly to the code pointed at in the closure.

As we cannot jump to code directly, as we cannot perform true tail calls, we must emulate tail calls with method calls. The problem now is preventing the native stack from overflowing because of the pervasive use of recursive loops. Having to adjust the stack on a recursive call also causes performance to suffer. All this would be resolved if the JVM provided a true tail call instruction allowing unrestricted jumps. A good description of the general problems of compiling to a target that doesn't explicitly support tail calls is [12]. The only tail calling tricks that seem possible on the JVM are:

- Embed the entire program inside a single Java method. The problem with this is the limits imposed by the JVM specification on both method body sizes and how far you can jump. There is a 16 bit offset `goto` and a 32 bit offset `goto_w`. These only operate within the calling method. The real problem is that there is a hard limit on the size of method bodies, limiting them to 64K - 1 bytes.

- A tiny-interpreter: treat parameter-less Java methods as extended basic blocks, using a one-level trampoline to transfer control forwards and backwards. Jump to a closure's code, and have that code return the address of the next closure to jump to. Unfortunately, in Java we can't return a pointer to the code to jump to directly, so there is one extra level of indirection here.

- A general form of trampolining can be used. We call functions as normal. When it appears the stack is about to overflow an exception can be raised returning control back to the top level, and cleaning up the stack. This device is used in the Mondrian compiler [17], and is quite elegant.

The tiny interpreter is used in the GHC for unoptimised target architectures [9] and in the JVM implementation of [5]. The top level of the runtime system is a loop of the form:

```
while (c != null) { c = c.enter(); }
```

This loop passes control between closures. This is the technique used in this project, though the more general trampoline technique seems worthy of investigation. Each closure must return a reference to the closure it wants to jump to. So a `case` expression will return a reference to the scrutinee it wants to evaluate, which will then return the continuation reference stored on the return stack. A `null` reference is used to signal completion.

Non-escaping local functions that are not used as first-class values and that are invoked tail recursively (all of which happens reasonably often) could be

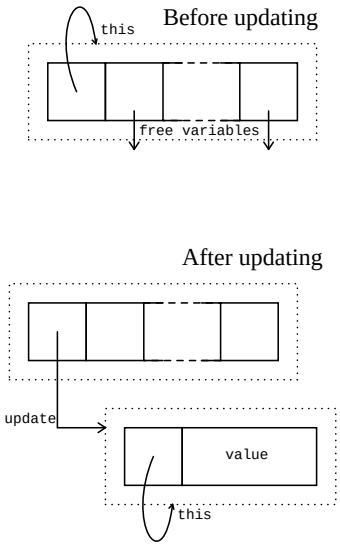

Figure 5.2: Performing an update

compiled as loops, following Serrano and Serpette [19]. This is a handy optimisation for a language like Scheme which makes less use of higher order values than Haskell.

5.8 Updates

We now turn to updates. For lazy evaluation to be feasible we must update expressions with their normal form to prevent redundant evaluation. This is achieved, ideally, by physically overwriting a closure with its normal form. Unfortunately, the JVM provides no mechanism to directly overwrite a heap object. Updates, then, involve generating a new node to represent the normal form of the expression. Entry to the closure is now modified to handle this indirection possibility. Each closure has a reference, `ind`, that normally points to itself (a `this` pointer). After updating the closure adjust `ind` to point to the new node.

The tiny interpreter now looks like:

```
while (c != null) { c = c.ind.enter(); }
```

This allows one level of indirection. Further updates must be flattened so that the original closure always points to the most recent update. This is described in figure 5.2.

As was seen earlier, STG code provides update flags indicating closure in which updates are unnecessary and this is helpful in such a costly update implementation. Use of these flags has not been discussed by any of the other STG machine implementations for Java, so their exploitation may be new on the JVM, where they may be quite helpful.

This general update technique has been used in a number of other projects: those of Wakeling [24] and Choi, Lim and Han [5]. We retain the spirit of the

GHC self-updatable closure model by arranging the code of the closure itself to perform updates and arrange the indirections, maintaining the uniform closure entry technique.

That it is necessary to allocate a new closure on the heap for every update explains most of the poor performance of the Java implementations. Creating a JVM that can do destructive updates, as well as providing a tail call instruction would be an improvement.

5.9 Runtime system

An STG machine implementation requires an explicit stack for passing arguments, return values, addresses of continuations and triggering updates. A variety of mappings of the argument stack to JVM features have been proposed:

- The argument stack can be mapped to the JVM's stack (which may implemented in terms of the native stack). This is the approach taken in the Bigloo Scheme compiler [19], and the MLJ compiler [1]. The SML compiler in [1] uses the Java native stack occasionally, along with local variables wherever possible. This compiler is impressively fast, although the compiler is a large piece of software now.

- As the arguments to functions are of varying types, we could implement the stack as an array of type `Object`. (`Object` is essentially the polymorphic type of Java). This is the approach taken by the Mondrian compiler [17], and most of the Haskell to Java implementations.

  A problem with this approach is that unboxed types would have to be boxed to be passed, with a large performance penalty, as the STG code produced by GHC makes extensive use of unboxed types. Tullsen in [21] compares a boxed and unboxed integer add function, with the arguments passed on the conventional Java stack. The boxed add is roughly 5 times slower than the unboxed add. Boxed integer addition using an `Object` stack is around twice as slow as with the native Java stack [21].

- GHC's STG machine implementation merges the separate stacks, and used to divide them between pointer and non-pointer stacks. This is an interesting possibility for a JVM mapping. An `Object[]` could be used to hold closure references, while keeping a variety of primitive arrays for integers, characters and the like, to avoid unnecessary boxing. This is a new variety proposed by this project, although it is roughly equivalent to the use of global variables as "registers", along with a polymorphic array for the large amount of pointer passing.

The variety of approaches to stack implementation reveal that the ideal solution is far from clear. The fastest systems have at least used the native stack occasionally. As we use a tiny interpreter to jump forward and back between closures, the closures require a uniform entry method, so passing arguments on the native stack seems difficult under this model without compiling to bytecode and manipulating the stack by hand. Such tricks are possible, as mentioned in [19], although current Java machines tend to not appreciate such code, often crashing.

Using the native stack results in faster code, but it makes calling functions harder. Curried functions that take their arguments one at a time are difficult under the native stack passing technique, as separate entry points must be compiled for all combinations of arguments a function could be called with. This is to say, it is harder on the compiler to use the native stack uniformly, though not impossible, and a high performance implementation may attempt this. The results for the Scheme and Standard ML compilers suggest that for full performance this is the way to go.

Summary

It is clear, then, that the closure structure in functional languages maps neatly to a Java object. Arguments are passed to the closure's code via explicit Java stacks, not the native stack, and we provide alternative entry points (as instance methods of closures) to enable more efficient entry to functions applied saturated, bypassing argument satisfaction checks.

The perennial problems of tail calls and updates are still difficult, though with reasonable care performance may not be too bad. Explicit use of unboxed values provides a large performance gain, as does compiling tail recursive functions to standard loops,

We now tie these design considerations together in a concrete implementation of a Haskell to Java compiler. This compiler benefits from the separate design innovations of previous experimental work, along with some new, or at least previously unmentioned, features.

Chapter 6

Compiler Implementation

The `stg2jvm` compiler is an external Java backend for GHC. It parses the STG code produced by GHC, constructing an internal representation of the STG program which is used to create an abstract Java tree. This is then written out as a set of Java class files implementing an STG machine. These can then be compiled with a normal Java compiler, and then executed on a JVM. The compiler itself is written in Haskell, and uses a variety of Haskell-based compiler construction tools.

Build system

`stg2jvm` is portable to any architecture to which GHC has been ported. It has been been successfully compiled and tested on the following platforms:

i386	OpenBSD 3.1
i386	GNU/Linux 2.4
i386	Solaris 2.6

It can be compiled with any GHC version newer than 5.00.2. Earlier versions of GHC require hand building, rather than with the current build system, which depends on GHC's new `--make` code to resolve dependencies for us. It builds with both GNU and BSD `make`.

GHC frontend

The ability to use GHC as an optimising front end for the compiler is extremely useful, and has been a requirement of this project from the start. GHC produces highly optimised STG code, far better than any we could reasonably expect to produce. GHC parses full Haskell, plus extensions. It typechecks the source, before translating the Haskell to the Core intermediate language.

Core code is repeatedly rewritten, as GHC performs a variety of optimisations. The Core is then annotated with free variable information, update flags and other information described in previous chapters, becoming STG code. STG code is then printed to standard output ascii form.

STG code is a rather ad hoc representation of the internal syntax tree, and a clean, standardised STG code is desirable. Core itself has recently been standardised and it would be nice for STG code to follow this lead.

`stg2jvm` reads in the STG code piped from GHC. The entire compiler is implemented in Haskell, along with a significant amount of scaffolding code in `perl`, `shell` and `make` scripts.

The entire compilation path from Haskell to Java source can be traversed by invoking the `hs2java` script on a Haskell source file. This is a perl script arranging the execution of all the various components of the system, from GHC to Java source.

Driver

The driver for the entire compiler is a simple Haskell `do` block arranging for the parsing of command line switches and then executing the various phases of the compiler in order. Haskell allows compiler execution paths to be described quite elegantly in this manner.

Test suite

A test suite is available for the compiler. The entire set of tests for the lexer, parser, pretty printer and code generator can executed by typing `make check` in the source directory. These include parser tests of extremely large STG files extracted from the infamous `nofib` Haskell testsuite.

6.1 Lexer

The lexer for the raw STG code is implemented using the Compiler Tool Kit [27] lexing combinators, documented in [4]. This is a Haskell library enabling a Haskell lexer to be written in a clean regular expression code style. CTK also provides a parser and pretty printing library along with other tools useful for a compiler writer.

The lexical structure of STG code is much the same as that of the Haskell identifiers, literals and keywords found in chapter 2 of the Haskell 98 Report [13]. The variations on these, such as the recursive binder flags or the `DEFAULT` keyword, can be blamed on the ad hoc nature of GHC's current STG code pretty printer.

A textual representation of the lexer results can be produced by invoking the compiler with the `-l` or `--dump-lex` flags.

6.2 Parser

A parser for the STG code grammar described in chapter 4 was implemented using the Happy Haskell parser generator [14]. `Happy` allows us to write a monadic parser to capture error conditions. It takes a source file in much the same format as Yacc. The grammar for STG code was reverse engineered from GHC's source.

The parser results can be obtained with the `-p` or `--dump-parse` flags. As an aside, the parser grammar illustrated in this document was obtained directly from the parser source using a 100 line `sed` script, `happy2tex`, available with the compiler.

6.3 Pretty printer

A pretty printer for STG code, and for the Java source code generator has been implemented, using the CTK library's implementation of John Hughes' pretty printing combinators [7].

The compiler can read in STG code, parse it, and then print it out in a visually appealing form with the `-P` or `--pretty-print` flags. The Java source code printer was implemented in a similar way.

6.4 Code generation

Code generation takes place in two passes. The first pass walks the STG syntax tree constructing an abstract tree of Java classes. This abstract Java is then printed out as Java source files using the pretty printer combinators.

Code generation is the default operation of the compiler, and Java source files will be generated if no other flags are specified.

6.5 Runtime system

The Java source must then be compiled and linked in with the runtime support routines implemented in Java, along with the various superclass definitions.

The top-level of the Java source system is a list of `public`, `static` globally visible Java closure references. This is implemented as the public class `G`, imported by all generated Java source files.

The runtime support package currently implements the various registers and stacks something like this:

```
class Runtime {
    public int sp;
    public int slimit;
    // our stacks
    public Object[] A;
    public int[]   aI;
    public char[]  aC;
    public float[] aF;
}
```

We provide separate stacks for the various different primitive types, as well as the pointer (reference) type.

The global environment described earlier is a set of Closure references, with special identifiers beginning with `$`. The Java main method controls execution via the tiny-interpreter discussed in the previous chapter.

```
class Global {
    public static Closure $a = new Closure();
    public static Closure $b = new Closure();
    public static Closure $main = new Closure();
}
```

Chapter 7

Conclusion

7.1 Results

At the time of writing no performance results are available as the code generator is only partially completed. It is *expected* that it will produce code that when executed is around twice as fast as the Hugs Haskell interpreter. It may indeed be faster, as this appears to be the only project attempting to fully utilise optimised STG code, along with an efficient runtime system developed from the fastest Scheme and Haskell Java compilers available.

Previous Haskell compilers have been anywhere from 5 times slower than Hugs, to around twice as fast. By using optimised STG code, and as much of the native features of the JVM as possible, we can reasonably expect such performance. This expectation awaits rigorous testing.

7.2 Getting there from here

Correct compilation of switch blocks to jump tables, and a fast implementation of the native `instanceof` operator, both seem like useful advances. Java compilers currently tend to produce poor code when given unusual constructs, such as those produced by functional language translation. This has been noticed in many papers on this subject. Fixing the broken Java compilers and JVMs would be a starting point.

Worth investigating is whether compiling to Java bytecode, rather than source, would provide any benefit. Low level manipulation of the native stack, along with explicit jumps seem useful, although there has been no real consensus on the bytecode target issue. The outstanding cases, however, have been the Scheme and Standard ML compilers that follow this route, whose performance has been excellent. Haskell's far greater use of higher order functions, and its lazy semantics are much of the problem here.

Using the native stack for all argument passing, returns and updating also seems like a reasonable optimisation, as the native stack can be assumed to be a reasonably fast data structure. Significant changes are requried to the compiler design to enable this, and it is unknown how much of a performance increase can be expected.

It seems reasonable to conclude, however, that limits to the efficiency of the Java mapping of Haskell code are close. Real performance gains can now only be achieved by employing a modified virtual machine that allows both destructive updates and true tail calls. Some ability to reduce the number of indirections by providing function pointers inside the virtual machine would also be helpful. Further work is proposed to investigate such a modified virtual machine.

Bibliography

[1] Benton, N., Kennedy, A., and Russell, G., *Compiling Standard ML to Java Bytecodes*, (1998), Proc. ICFP '98, 129-140

[2] André Rauber du Bois and Antônio Carlos da Rocha Costa, *Functional Beans*, (2000), Functional and Logic Programming Ninth International Worskshop, WFLP'2000, `http://www.dsic.upv.es/~wflp2000/`

[3] Bothner P., *Kawa Internals: Compiling Scheme to Java*, (1998), `http://www.gnu.org/software/kawa/internals.html`

[4] Chakravarty M., *Lazy Lexing is Fast*, (1999), in A. Middeldorp and T. Sato eds., Fourth Fuji International Symposium on Functional and Logic Programming, Springer-Verlag, LNCS 1722, 68-84

[5] Kwanghoon Choi, Hyun-il Lim, Taisook Han, *Compiling Lazy Functional Programs Based on the Spineless Tagless G-Machine for the Java Virtual Machine*, FLOPS 2001, 92-107

[6] The GHC Team, *The Glasgow Haskell Compiler User's Guide*, v5.02 (2002), `http://www.haskell.org/ghc/docs/latest/html/users_guide/users-guide.html`

[7] Hughes J., *The Design of a Pretty-printing Library*, (1995) Advanced Functional Programming, LNCS 925 Springer Verlag

[8] Lindholm T., Yellin F., *The Java Virtual Machine Specification*, 2nd ed (1999), `http://java.sun.com/docs/books/vmspec/`

[9] Peyton Jones S.L., *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, v2.5 (1992), in Journal of Functional Programming, 2(2):127-202

[10] Peyton Jones S.L., Santos A., *A transformation-based optimiser for Haskell*, in Science of Computer Programming, 32(1-3):3-47, 1998.

[11] Peyton Jones S.L., Lester D., *Implementing Functional Languages: A Tutorial*, (1992), Prentice Hall

[12] Peyton Jones S.L., Nordin T., Oliva D., *C--: A Portable Assembly Language*, (1998), LNCS 1467

[13] *Report on the Programming Language Haskell 98*, eds. Simon Peyton Jones, John Hughes, (1999) `http://www.haskell.org/onlinereport/`

[14] Marlow S., Gill A., *Happy user guide*, (2002), `http://www.haskell.org/happy`

[15] Marlow S., Peyton Jones S.L., *The New GHC/Hugs Runtime System*, (1998), unpublished, `www.haskell.org/~simonmar/abstracts/rts.html`

[16] Meehan G., Joy M., *Compiling Lazy Functional Programs to Java Bytecode*, (1999), Software Practice and Experience, 29(7):617-645

[17] Perry N., Meijer E., *Implementing Functional Language on Object-Oriented Virtual Machines*, (2002), `http://www.research.microsoft.com/~emeijer/Papers/ImplementingFL.pdf`

[18] Schinz M., Odersky M., *Tail call elimination on the Java Virtual Machine*, Electronic Notes in Theoretical Computer Science 59(1), (2001)

[19] Serpette B., Serrano M., *Compiling Scheme to JVM Bytecode: a performance study*, (2002), Proc. ICFP '02, 259-270

[20] Tolmach A., *An External Representation for the GHC Core Language (DRAFT for GHC5.02)*, (2002), `http://www.haskell.org/ghc/docs/papers/core.ps.gz`

[21] Tullsen M., *Compiling Haskell to Java*, (1996), Technical Report YALEU/DCS/RR-1204, Yale University, `http://www.cs.yale.edu/homes/tullsen/haskell-to-java.ps`

[22] Vernet A., *The Jaskell Project*, (1998) `http://www.scdi.org/~avernet/projects/jaskell/`

[23] de Vries M., *Octopus: A Parallel Functional Programming Language for Loosely-Coupled Architectures*, (2000), `http://vo.com/~martijn/report.pdf`

[24] Wakeling D., *A Haskell to Java Virtual Machine Code Compiler*, Implementation of Functional Languages 1997: 39-52

[25] Wakeling D., *Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine*, PLILP/ALP 1998, 335-352

[26] Arnold K., Gosling J., *The Java Programming Language*, Addison Wesley, (1996)

[27] `http://www.cse.unsw.edu.au/~chak/haskell/ctk/`