

Efficient Translation of Haskell to Java

Master's Thesis Proposal

Brian Alliet
bja8464@cs.rit.edu

March 15, 2007

Abstract

The Java Virtual Machine [LY99] is an abstract machine designed to support object oriented languages. The strong security guarantees enforced by the JVM and its portability make it a desirable target for distributing mobile code. Haskell [JHA⁺99] is non-strict, higher-order functional language. Mapping non-strict higher order languages to any stock machine poses many unique challenges [Jon92]. Mapping these languages to strongly typed secure virtual machines such as the JVM leads to further challenges.

First we discuss the the general issues involved in implemented non-struct functional language on Von Neumann style architectures. Next we discuss previous attempts at implementing these languages on general purpose secure virtual machines [Tul96, CiLH01, Ste02]. Finally we present our plan for efficiently mapping non-strict, higher-order functional languages (specifically Haskell) to the Java Virtual Machine.

1 Introduction

There are several compelling reasons for implemented functional languages on the Java Virtual Machine [LY99]. JVM bytecode is widely portable and can therefor run unmodified on a wide variety of different machines. This makes it an ideal delivery mechanism for mobile code. This is perhaps even more important for applications written in more obscure languages such as Haskell where a native compiler probably isn't available on the target system. Code compiled to run within the JVM also allows for easy (and

efficient) integration with existing Java libraries which are growing in number every day. The JVM is also an attractive implementation target as you get a good deal of runtime support (garbage collection, dynamic loading, verification, etc) “for free”.

Several aspects of the Haskell [JHA⁺99] programming language are pervasive in the its implementations. Haskell is non-strict, meaning arguments to functions are not evaluated when a function is applied. The most common way of implementing non-strict semantics is with lazy evaluation. Under lazy evaluation no expression is evaluated until is required. Haskell is higher-order, meaning functions are first-class values. They can be used as arguments, returned, and stored in data structures. In fact, functions with multiple arguments are most commonly defined as functions of one argument which return another functions which expects the remaining arguments (a technique known as “currying”). Haskell is tail recursive, a property that is essential for any functional language, as loops are implemented using recursion. Functions in tail call position are executed without increasing the stack size. Finally, non-primitive types are defined as algebraic data types, that is, types composed of one or more “data constructors” (for example: `cons` and `nil`). Flow control is based on pattern matching on these data constructors (do this if it is a `cons`, do that if it is a `nil`).

Similarly, several aspects of the Java [G⁺] programming language are pervasive in the implementation of its virtual machine (the JVM). Java is strict, meaning arguments are evaluated prior to applying a function. Java function (most commonly referred to as methods) are not first class. They cannot be used as values. Java classes, which contain methods, are first class. For security reasons, the JVM does not support tail recursion. A function whose last action is calling another functions will still retain its stack frame until the callee returns. Non-primitive types in Java are defined as “classes” (a collection of values and operation on them). Flow control is based on virtual method dispatch.

Haskell and Java seem to differ in every aspect of the language. The key challenge in this project is reconciling these differences in order to efficiently implement the Haskell programming language on top of the Java Virtual Machine.

2 The Spineless Tagless G-Machine

It's just a simple functional language ¹

The Spineless Tagless G-Machine is an abstract machine designed to support non-strict, higher-order functional languages [Jon92] (and by no coincidence, Haskell). The STG machine runs code written in the STG language, a minimalist functional language with precisely defined *denotational* and *operational* semantics. STG is the intermediate language used by the Glasgow Haskell Compiler [JHH⁺93], the most widely used implementation of Haskell.

The STG language is basically the typed lambda calculus with the addition of algebraic data types and some restrictions to make it more suitable for code generation. These restrictions include [Jon92]:

- Function arguments are atomic (literals or variables, not arbitrary expressions). This means the code for a function call simply needs to copy the arguments to registers, no need to evaluate or create thunks for them.
- The right hand side of a let binding is either a data constructor application, a lambda abstraction, or a thunk (unevaluated expression). Allocation is performed only by let expressions.
- Constructor and primitive applications are always fully saturated (applied to all their arguments).

2.1 Heap Objects

In STG heap objects consist only of constructor applications, lambda abstractions, and thunks, precisely what can be on the right hand side of a let binding. Every heap object is of a standard form. The first word is always a code pointer ². This is a pointer to the code to run when the heap object is "entered". Following the code pointer is the "payload", the arguments to the data constructor or the free variables in the expression.

¹"It's just a simple functional language" is an unregistered trademark of Peyton Jones Enterprises, plc.

²Actually it is a pointer to an "info table" which contains the entry point as well as additional information about the object needed by the garbage collector, but all we really care about for now is the code pointer.

A heap object is entered when it needs to be evaluated (when case analysis is performed on it). The entry point for a data constructor application simply returns to the caller as these objects are already evaluated. The entry point for a lambda expression pops its arguments off the stack and evaluates its body ³. The entry point for a thunk simply evaluates itself (the expression contained within the thunk) then *updates* the thunk with an indirection to the result of evaluating the thunk (so further attempts to evaluate the thunk simply return the already computed value).

2.2 Evaluating STG Expressions

Each STG expression has a straightforward translation to native code.

- Application expressions push their arguments onto the stack then enter the function.
- Let expressions allocate heap space for all of their bindings, fill in their code pointers and free variables, then enter the body of the let expression.
- Case expressions push a “return address”, that is, an address to jump to once evaluation is complete onto the stack then evaluate the expression they are scrutinizing. The return address is a pointer to code that dispatches to the appropriate case alternative depending on what data constructor the expression evaluates to and evaluates the expression for the alternative.

This is a vastly simplified view of the STG machine (see [Jon92] for a complete description) but the most important concepts to implementing it on the JVM are there.

3 JVM Challenges

The biggest challenges faced when implementing the STG machine on top of the JVM are handling tail calls, updating thunk, doing case analysis on algebraic data types, and contending with the limited expressivity of the JVM's type system.

³Things get slightly more complicated when you have partial applications, functions that aren't applied to all their arguments, but again, we don't worry about that for now.

The STG machine requires tail calls. Tail calls are how loops are encoded in functional languages. Without support for tail calls many functions would require stack space proportional to the size of their input (and in the case of lazy functional languages this could be infinite). The JVM does have any built-in support tail calls. We must therefore find some other way to implement tail recursion.

When the STG machine updates a thunk after evaluating it the heap object is physically overwritten with a pointer to the result. This way when the thunk is entered again the evaluated result is immediately returned rather than re-evaluating the thunk. In addition, the garbage collector knows to “look through” these updated thunks. So if object **A** points to object **B** which has been updated to point to object **C**, the garbage collector will see that there is no need to keep **B** around anymore and change **A** to point directly to **C** (allowing **B** to be reclaimed). This type of operation is certainly too low level to do on the JVM and there obviously isn’t any way to teach the JVM’s GC how to handle indirection so some other, efficient, way of updating thunks must be implemented.

Algebraic data types represent a union of different objects that all inhabit the same type. Evaluating an algebraic data type requires the implementation perform a “switch” on the data constructor contained within a heap object. The JVM doesn’t have any way to switch on types like this. The closest it has is `instanceof` which can only test one object type at a time. Some other method to switch on an object’s type in constant time is required.

Finally, JVM bytecode is statically typed. All bytecode must be verified to be typesafe. Unfortunately the JVM’s type system isn’t nearly as expressive as Haskell’s. Often we know (because the Haskell compiler proved it) what an object’s type is but we cannot prove that to the JVM within the constraints of the JVM type system. We can always use the `checkcast` sledgehammer but this is an expensive operation and is best avoided if possible.

4 Existing Work

Translating Haskell code for use on the JVM is not a new idea. It has been attempted a few times before [Tul96, CiLH01, Ste02]. The STG language has been used as the basis for each such work. This is mainly because

it is GHC's [JHH⁺93] intermediate language and using it allows GHC's frontend to be leveraged.

4.1 Mark Tullsen [Tul96]

Mark Tullsen implemented an STG to Java translator that would translate an STG-like (the implementation required some manual tweaking of GHC's output) language to Java source code (which would then be compiled with `javac`). It only supported a very small subset of Haskell, just enough to run some toy programs, but he explored some of the key issues raised above.

Tullsen didn't implement any form of tail calls. The standard Java invocation/return mechanisms were used.

Tullsen doesn't go into much depth about his implementation of closures but implies that he uses two Java objects to represent thunks. A simple indirection object initially points to the thunk object and when the thunk is evaluated the indirection object is updated to point to the result so the thunk can be reclaimed. This, of course, doubles the number of objects in the heap.

Tullsen represented algebraic data types by using one Java class per data constructor, each with a dummy class as their supertype. (The dummy class seems to be more for documentation purposes as it is never created directly.) He explored numerous methods for doing case analysis including:

- Testing `o.getClass()` in succession against `ConstructorName.class` for each constructor.
- Blindly casting the object to each constructor class in succession, using `ClassCastException` to fall through on failure. This obviously had terrible performance as exceptions are very expensive on the JVM.
- Using `instanceof` checks on each constructor class in succession.
- Adding an integer "tag" to each type and doing a normal Java switch on the tag followed by a `checkcast`.

The first three methods obviously required time proportional to the number of constructors and weren't ideal. Using the integer tag allowed the

operation to be done in constant time (`tableswitch` is a constant time operation on the JVM). This may not be the “java way” of doing it but it was the most efficient.

Tullsen makes no attempt to play nice with the Java type system. He simply passes everything around as `Object` and casts when necessary.

4.2 Kwanghoon Choi, Hyum-il Lim, and Taisook Han [CiLH01]

Choi, et al, like Tullsen, implemented an STG to Java translator that would translate an STG-like (the implementation too required some manual tweaking of GHC’s output) language to Java source code (which would then be compiled with `javac`). Their implementation too only supported a small subset of Haskell. They use a second intermediate language (called “L-Code”). Their STG subset is translated to L-Code which is then translated to Java (actually almost pretty printed as it maps almost directly to Java).

Choi, et al. fully support tail calls. They support them by using an explicit stack (an `Object []` array) rather than the JVM stack. As each function no longer depends on the JVM stack it can safely return to it’s caller who is running a “mini-interpreter” (an idea credited initially to Guy Steele [GLS78] and reinvented numerous times). The mini-interpreter is simply a loop of the form:

```
while(!done) whatnext = evaluate(whatnext);
```

When a function wants to make a tail call it simply returns the closure to execute next.

Choi, et al. Use a similar method to Tullsen to implement updatable closures. Two objects, one indirection, and one thunk, are allocated for each thunk. Algebraic data types are, too, implemented as one object per constructor with case analysis done as a switch on an integer tag. Finally, mainly as a result of having to implement their own stack as in `Object []` array, they too use `Object` throughout their code and do `checkcasts` as necessary.

4.3 Don Stewart [Ste02]

Don Stewart seems to have implemented one of the most promising Haskell to Java translators. He too converts the STG code output by GHC to Java

source code, however, unlike earlier works, he actually supports the full STG language. In fact, his code was eventually integrated into the GHC tree (although unfortunately it has bitrotted since then).

Stewart implements tail calls using an explicit stack and the “mini-interpreter” technique. However, to avoid the extra heap allocation required by using a polymorphic stack implemented as an `Object []` array he actually uses 4 stacks, one for each of `Object`, `int`, `float`, and `char`. (Why he omits `long` and `double` and why he doesn’t store `chars` in the `int` stack is unclear.)

Updates are handled similarly to previous works. However, rather than allocating a second indirection object for each thunk he simply stores the indirection in the thunk itself, reducing the object count by one (but potentially increasing space usage in the case of a large thunk that must now be kept around simply for its `ind` pointer).

Algebraic data types are implemented as one class per data constructor with switching on integer tags as in previous works.

Stewart’s use of multiple stacks helps avoid some of the excess `checkcast` operations normally associated with using a non-native stack. Other than that things are again passed around mostly as `Objects`.

5 Project Definition

We intend to:

- Implement a fully functional, efficient, JVM backend in the Glasgow Haskell Compiler [JHH⁺93].
- Provide a easy to use interface to existing Java libraries based on the Haskell Foreign Function Interface [Cha03]
- Use this interface to fully implement the *base* library package.

One of our key contributions is a more efficient implementation of the STG machine on the JVM. Existing work leaves much to be desired. We think the use of a non-native stack in all the existing work has been the primary factor in their performance problems. The requirement of using a non-native stack is mainly a side effect of using the push/enter evaluation model. The push/enter model requires direct access to the stack which is impossible

on the JVM. We propose to use the eval/apply [MJ06] evaluation model which frees us from this limitation. This in turn allows us to prove more typing properties of the generated code to the JVM as we're not using a polymorphic array as our stack which should significantly increase performance.

Our second key contribution is the totality of the conversion. This will not be a proof of concept implementation only capable of running toy programs. There will be no post processing required. There will be no additional tools to run. Everything will be fully integrated into GHC (and if time permits possibly integrated upstream) and building a JVM version of your Haskell application will simply be a matter of adding `-java` to your GHC command line:

```
ghc -java --make Main.hs
```

6 Deliverables

We will submit the following at the completion of the project:

- Additional sections to the GHC Users Guide describing our modifications and how to use them.
- A set of patches to the Glasgow Haskell Compiler and libraries implementing our system.

7 Final Report

The final report will consist of:

- An overview of the internals of the Glasgow Haskell Compiler and discussion of which areas we will be working and why.
- An overview of the Spineless Tagless G-Machine, the language we will be translating.
- Discussion of existing work in the area
- A thorough discussion of each of the major design decisions made during the course of the project. Including:

- Tail call implementation
- Closure representation
- Thunk representation
- How to perform updates
- Algebraic data type representation
- Partial applications representation
- Performance measurements
- Future work

8 Schedule

Weeks -9 - 1: Write the Compiler

Weeks 2 - 10: Write the Report

References

- [Cha03] Manuel M. T. Chakravarty. The haskell98 foreign function interface 1.0, 2003.
- [CiLH01] Kwanghoon Choi, Hyun il Lim, and Taisook Han. Compiling lazy functional programs based on the spineless tagless *G*-machine for the Java virtual machine. *Lecture Notes in Computer Science*, 2024:92-??, 2001.
- [G⁺] James Gosling et al. *The Java Language Specification*. GOTOPI Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan.
- [GLS78] Jr. Guy L. Steele. Rabbit: A compiler for scheme. Technical report, Cambridge, MA, USA, 1978.
- [JHA⁺99] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely

Functional Language. Technical report, February 1999. Available at <http://www.haskell.org>.

- [JHH⁺93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 93.
- [Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
- [LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, 1999.
- [MJ06] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006.
- [Ste02] Don Stewart. Multi-paradigm just-in-time compilation. November 2002.
- [Tul96] Mark Tullsen. Compiling haskell to java. Technical Report YALEU/DCS/RR-1204, Yale University, May 1996.