HamSkill: Run Haskell Anywhere with ANTLR and Scala

CS252 Project Final Report

Zayd Hammoudeh
(zayd.hammoudeh@sjsu.edu)

March 29, 2016

Contents

1 Running in the Java Virtual Machine

C is one of the most commonly used languages when the goal is maximum performance. However, C/C++'s "write once, compile anywhere" paradigm limits its portability. In contrast, the near ubiquity of the Java Virtual Machine (JVM) allows Java to be "write once, run anywhere."

On many occasions, developers have leveraged the JVM's "run anywhere" capability to run allow other languages. Examples include: JRuby for the Ruby programming language [2], Jython for the Python programming language [3], Renjin for the R programming language [4], and Scala [5].

Currently, there is no full implementation of Haskell in the JVM. One Haskell dialect that is runnable in Java is Frege [8].

In this project, I will implement, *HamSkill*, a dialect of Haskell that is runnable in the Java Virtual Machine.

2 Hamskill Program Structure

The HamSkill implementation consists of four major components. They are:

- ANTLR Lexer and Parser
- Haskell Antlr Grammar
- HaskellMain Java Class
- Scala Runtime Environment
- ScalaOutput Antlr Grammar
- ScalaOutput Java Class

The relationships between these four components are shown in figure 2.

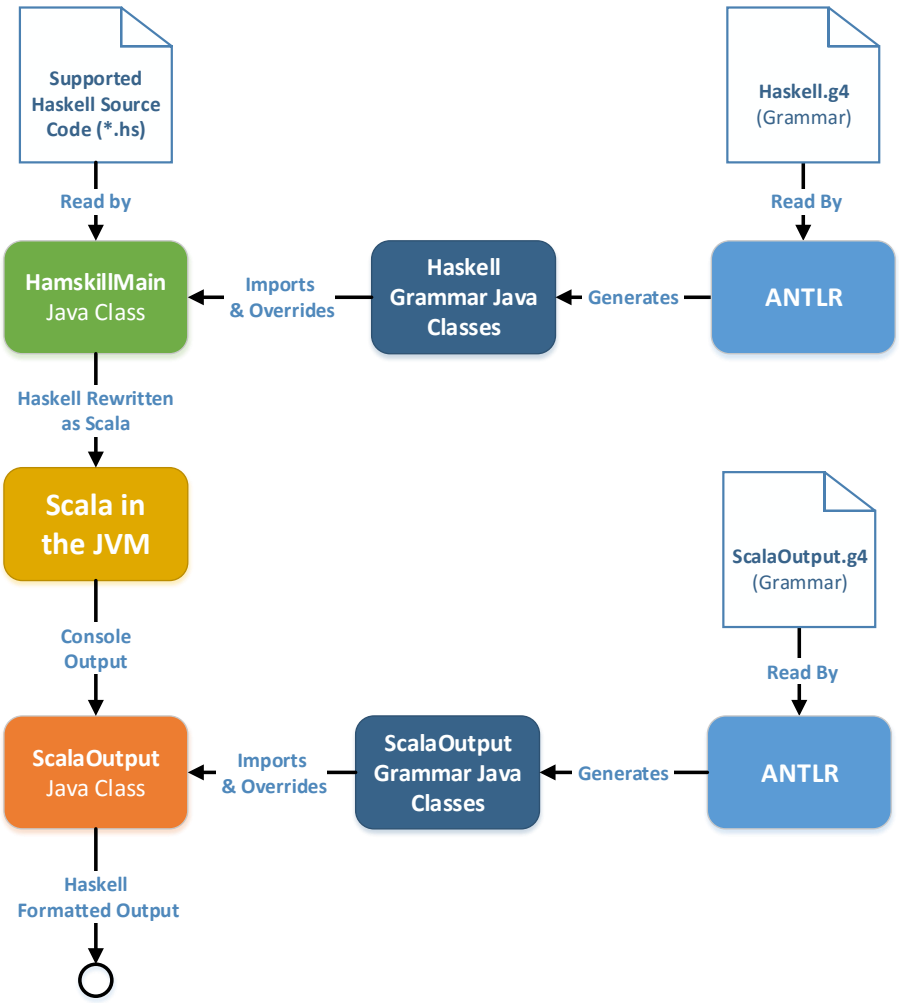

Figure 1: HamSkill Project Architecture

The following subsections describe each of the components of this architecture.

2.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is an adaptive left-to-right, left-most deriviation (LL(*)) lexer and parser written in the Java programming language. ANTLR's primary function is to read, parse, and process

structured text (e.g. Haskell code) [7].

2.2 ANTLR Version 4 Grammar

A grammar is a formal description of language; it is based off the concept of a context-free grammar in formal automata theory. ANTLR version 4 (v4) grammar files (denoted by the file extension *.g4*) explicitly define the how ANTLR will parse the structured text. The grammar file may contain token definitions (which always start with a capital letter) and/or parser rules (which always start with a lowercase letter). The tokens are groups of characters that form a single object; the parser then uses these tokens to recognize sentence structure within the document.

Two separate grammars are required for this project. They are described in the following two subsections.

2.2.1 Haskell Grammar

2.2.2 ScalaOutput Grammar

2.3 ANTLR Classes

Programs that use ANTLR do not operate directly on the grammar. Rather, the grammar is compiled by ANTLR into a set of class files; as such, given a grammar *GrammarName.g4*, the following Java files would be created

- *GrammarName*Lexer.java - This class is the the lexer definition for the input grammar file. It also extends ANTLR's base Lexer class.
- *GrammarName*Parser.java - Each rule in the original grammar constitutes a method in this class; it forms the parser class definition for the input grammar file.
- *GrammarName*.tokens - Assigns a token type (e.g. integer, identifier, floating point number etc.) to each token in the input grammar file.
- *GrammarName*Listener.java & *GrammarName*BaseListener.java - ANTLR builds applies the grammar to a text input to build an Abstract Syntax Tree (AST). While walking the tree, ANTLR fires events that can be captured by a listener[7].

2.4 JVM-Supported Programming Language

When selecting the implementation language, my criteria were: runnable in Java, and maximum similarity to Haskell. I initially considered Python (via Jython) due to its higher order function support and concise style.

In the end, I selected Scala for this project due to its syntax being more alignable to that of Haskell; an example of this is Scala's support for a function-based pattern matching style. What is more, critical aspects of Haskell (e.g. lazy evaluation, immutability of objects, etc.) have parallels in Scala. One major disadvantage of this decision is that Scala has a much weaker type inference system than Haskell or Python. An example of this is described in section **??**.

While it is my expectation that a fuller implementation of Haskell may be more achievable in Python, it would require substantially more effort to implement the functional programming aspects of Haskell that Scala comes with out of the box. In the end, when doing a programming language conversion, it is exceptionally unlikely that the destination language will perfectly support all aspects of the source language. As such, some degree of compromise is required which is why *HamSkill* is only a dialect of Haskell.

2.5 Immutability in Scala

One of the key features of Haskell that allows it to achieve referential transparency is the immutability of data. While it would be possible to develop an infrastructure in a language like Python to assure immutability, it is an encumbrance. In cases such as this, it is almost always better to take advantage of a language's native features when possible. In Scala, the `val` construct ensure the immutability of an object without any user intervention. For example, the code in figure 2 would raise a runtime error since it is trying to change the value of immutable data.

```
val x = 5
x = 3
```

Figure 2: Declaring Immutable Data in Scala

2.6 Lazy Evaluation

Another important aspect of Haskell is that it supports lazy evaluation. This entails that data's value is not calculated until it is not needed. Figure 3 is an example of lazy evaluation with Scala as when this code is run, it will print a negative elapsed time (which is clearly not correct).

```
def lazyTime(){
  lazy val t1 = System.nanoTime()
  val t2 = System.nanoTime()
  Thread.sleep(1000)
  println("Elapsed time is " + (t2-t1)/1000000 + "ms")
}
```

Figure 3: Lazy, Immutable Code in Scala

Scala also supports "call-by-name" to achieve laziness of function parameters. However, this feature is not truly lazy as it will recalculate the value each time the parameter is used in the function. This limitation often degrades the overall the performance; for this reason, I do not plan to implement laziness in *HamSkill* across functions.

2.7 Supported Types

HamSkill will only support a select subset of Haskell's available types. The list of planned types are: `Bool`, `Integer` (i.e. bounded), and `List` (currently only finite lists are planned, but that may change depending on the complexity of the implementation).

While implementing floating point numbers would not add substantial complexity at a basic level, ensuring that the floating point behavior of *HamSkill* (i.e. Scala) and Haskell are identical is beyond the scope of this project.

2.8 From `show` to `toString`

The function in Haskell to convert data to string is "`show`". In contrast, the syntax in Scala to convert an object (e.g. "x") to a string is "`x.toString()`".

Due to this, the ANTLR parser will need to be able to convert a prefix function to an object method call.

2.9 Nested Function Calls

Imperative languages (e.g. Java) are generally more verbose than functional languages; Haskell is no exception to this. Conciseness introduces significant challenges when writing a parser as the contextual information is reduced. For example, figure 4 is a simple line of Haskell code that prints to the screen the result of a function "`addTwoNumbs`" that takes two integers (e.g. "`x`" and "`y`") and sums them.

```
putStrLn $ show $ addTwoNumbs x y
```

Figure 4: Simple Function Call in **Haskell**

Similar code in Java is shown in figure 5

```
System.out.println( addTwoNumbs(x, y) ) ;
```

Figure 5: Simple Function Call in **Java**

The Java syntax explicitly shows that `addTwoInts` is a function since the parameters are inside parentheses and are comma separated. To simplify the parsing for this in Haskell, the *HamSkill* dialect will require function arguments to be succeeded by double parentheses "`((`" and "`))`". Figure 6 shows the *HamSkill* version of the Haskell code in figure 6.

```
putStrLn $ show $ addTwoInts ((x y))
```

Figure 6: Simple Function Call in ***HamSkill***

2.10 Defining Scope and Scala Object Name via `module`

A program in Haskell is composed of a set of "`module`" files. The "`module`" keyword is used to scope of functions (e.g. `public` or `private`) as well as for defining an abstract data type [1]. In *HamSkill*, I will use the Haskell module to define whether the Scala methods are private (since by default functions are `public`) as well as the name of the Scala object.

2.11 Partially Applied Functions

Haskell supports partially applied functions. Figure 7 shows the `addTwoInts` function with a single argument (i.e. "`5`") being stored in a variable "`addFive`".

```
let addFive = addTwoInts 5
```

Figure 7: Partially Applied addTwoInts Function in **Haskell**

Partially applied functions in Scala have advantages and disadvantages in comparison to Haskell. One of these disadvantages is evident when figures 7 and 8 are compared. Note that the Scala function requires an underscore ("_") for each missing argument as well as the type for that argument. This makes converting Haskell code to Scala problematic as the function prototype must be fixed and known at conversion time.

```
val addFive = addTwoInts(5, _: Int)
```

Figure 8: Partially Applied `addTwoInts` Function in **Scala**

To simplify this, *HamSkill* will require that any partially defined functions are declared in the same file/module. I will investigate using a predefined list of functions, but this may not be feasible or support will be very limited due to the requirement to define the parameter type. What is more, partially applied functions will use need to use the "double parentheses style" described in section 2.9.

2.12 Higher Order Function Support

Scala and Haskell are both functional programming languages; one important consequence of this is that both support higher order functions. *HamSkill* will support functions as input parameters to functions. If time allows, I will also investigate the ability to return functions from functions. The extent to which this is supported will be dependent on the extent to which partially applied functions are supported as defined in section **??**.

2.13 Haskell Lambda Function to Scala Anonymous Functions

There is significant similarity between a Lamda function in Haskell and an anonymous function in Scala. One primary difference is that Scala requires the developer to specify the types of the parameters in the anonymous function while Scala does not. For this project, I will implement support for anonymous functions either as parameters to other functions or for support for an operation such as folding or filtering a list.

2.14 Maybe Monad Support

3 Test Bench

3.1 Implementation

Bibliography

[1] A gentle introduction to haskell: Modules. https://www.haskell.org/tutorial/modules.html. (Accessed on 03/01/2016).

[2] Home jruby.org. http://jruby.org/. (Accessed on 02/24/2016).

[3] Platform specific notes. http://www.jython.org/archive/21/platform.html. (Accessed on 02/25/2016).

[4] Renjin.org — about. http://www.renjin.org/about.html. (Accessed on 02/25/2016).

[5] The scala programming language. http://www.scala-lang.org/. (Accessed on 02/25/2016).

[6] Thiago Arrais. haskell-lexer.g. http://eclipsefp.sourceforge.net/repo/net.sf.eclipsefp.haskell.core.jparser/antlr-src/haskell-lexer.g. (Accessed on 03/01/2016).

[7] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012.

[8] Ingo Wechsung. Frege/frege: Frege is a haskell for the jvm. it brings purely functional programing to the java platform. https://github.com/Frege/frege. (Accessed on 02/25/2016).