HamSkill: Run Haskell Anywhere with ANTLR and Scala

CS252 Project Final Report

Zayd Hammoudeh
(zayd.hammoudeh@sjsu.edu)

March 30, 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 [9].

In this project, I will implement, *HamSkill*, which is a transpiler from Haskell to Scala; *HamSkill* enables a dialect of Haskell to be runnable in the JVM.

2 Key Project Requirements

When designing and implementing this project, there were four primary goals:

- **Runnable in the Java Virtual Machine** - As explained in section 1, Java's Virtual Machine enables significant machine independence, which Haskell does not currently enjoy.
- **Minimal JVM Requirements** - In addition to just running in the JVM, HamSkill was intended to be as standalone as possible. For example, for most applications, it was not expected that the user would have Scala installed on their machine. To achieve this, some advanced features may not be supported.
- **Identical Input and Output Between Haskell and *HamSkill*** - In many scenarios, it may not be sufficient for a program to run in Haskell and in JVM. Rather, it may often be required that the generated output for the two programs be identical as well. As such, *HamSkill* will utilize an additional post processing step to ensure its output is identical to that generated by Haskell.
- **Human Readable Output Code** - A transpiler is any program that takes source code from one programming languages and outputs as code in another programming language with a similar level of abstraction [7].

To enable increased reuse of the outputted code, HamSkill will use proper indenting, newlines, etc. to maximize readability the generated output. While this is not a requirement for the complete system to work properly, it can enhance the potential of the tool.

3 *HamSkill*'s Software Architecture

HamSkill is a transpiler that takes input Haskell code, converts it to Scala, and then runs it in the JVM. 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 3.

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

3.1 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) [8].

3.1.1 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

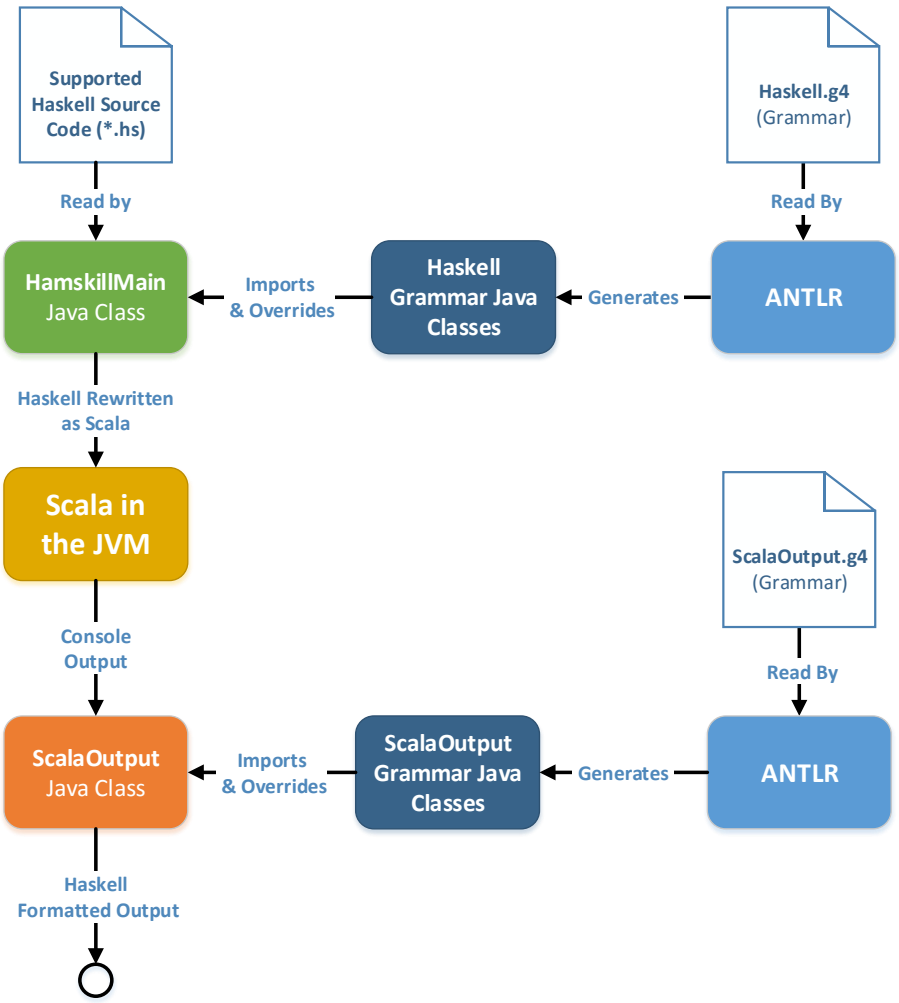

Figure 1: HamSkill Project Architecture

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.

This project utilizes two separate grammars namely: `Haskell` and `ScalaOutput`. They are described in the following two subsections.

3.1.1.1 Haskell Grammar

The Haskell parsing grammar is stored in a file named "`Haskell.g4`." This grammar defines both the tokens and abstract syntax tree for supported Haskell code. Some of Haskell features it supports include:

- `case` Statement
- `if`, `then`, `else` Conditional
- `Maybe` Monad
- Currying
- Partially Applied Functions
- Higher Order Functions
- Lazy Evaluation

For the complete feature set as well as specific Haskell syntax requirements, see section 4.

3.1.1.2 ScalaOutput Grammar

As mentioned in section 2, one of the key requirements of this project was that the outputs from Haskell and HamSkill should be identical. Figure 2 and 3 show the code to print a list containing "2", "3", and "4" in Haskell and Scala respectively. Note that the subsequent lines should the results of the print statement for the two languages are very different.

```
Prelude> putStrLn $ show [2, 3, 4]
[2,3,4]
```

Figure 2: Printing a Three Element List in Haskell

```
scala> println(List(2,3,4))
List(2, 3, 4)
```

Figure 3: Printing a Three Element List in Scala

The `ScalaOutput` grammar (contained in the file "`Haskell.g4`") parses the output of the Scala program and converts it to a syntax that is more similar to that of Haskell.

3.1.2 ANTLR Classes

Programs that use ANTLR do not generally operate directly on the grammar. Rather, the grammar is converted by ANTLR into a set of Java files; given an example grammar, *GrammarName.g4*, ANTLR would create the following files:

- *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[8].

Most programs that want to use ANTLR override the functions in the file *GrammarName*BaseListener.java.

3.1.2.1 HaskellTokensToScala

3.1.2.2 ScalaOutputTokensToHaskellFormat

3.2 Transpiler Output Language: Scala

As mentioned previously, a transpiler takes source code from one programming language and converts it to code in another language. It was also previous explained that the primary criteria when deciding on the output language was that it needed to be runnable in the Java Virtual Machine. Other important criteria were:

- **Higher Order Function Support** - Any language that is devoid of higher order support would not be a good match with a purely functional language like Haskell.
- **Similar Syntactic Structure** - Closer alignment between the input and output languages will simplify the trans-compilation. Inevitably though, some amount of restructuring and reformatting will be required.
- **Personal Interest** - Increased personal investment generally leads to a more fulfilling outcome for the student and a better project overall.

The language that best fit all three of these criteria is Scala, which is a functional language which can be fully run inside the JVM [6]. What is more, it natively supports many of Haskell's core features including: functions written in a pattern matching style, immutability of objects, currying, partially applied functions, lazy evaluation, static typing, etc. One of the disadvantages of Scala is its weaker type inference in comparison to Haskell; due to this specific requirements were placed on the requirements of the Haskell dialect supported by Hamskill as described in section 4.

It is also important to note that the author has had a personal predisposed interest in learning Scala given its extensive support by Apache Spark. As such, given the "personal interest" requirement mentioned previously, the selection of Scala for this project became an even more obvious choice.

3.2.0.1 *HamSkill* Standard

3.2.0.2 *HamSkill+*

4 HamSkill Features

This section enumerates the Haskell features that are supported by HamSkill. It also describes any formatting requirements that may apply to these features.

4.1 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 4 would raise a runtime error since it is trying to change the value of immutable data.

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

Figure 4: Declaring Immutable Data in Scala

4.2 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 5 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 5: 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.

4.3 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.

4.4 Conversion from Functions to Methods

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.

4.5 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 6 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 6: Simple Function Call in **Haskell**

Similar code in Java is shown in figure 7

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

Figure 7: 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* requires function arguments to be preceded and succeeded by triple parentheses "`(((`" and "`)))`". Figure 8 shows the *HamSkill* version of the Haskell code in figure 8.

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

Figure 8: Simple Function Call in ***HamSkill***

4.6 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.

4.7 Partially Applied Functions

Haskell supports partially applied functions. Figure 9 shows the `addTwoInts` function with a single argument (i.e. "`5`") being stored in a variable "`addFive`".

```
let addFive = addTwoInts 5
```

Figure 9: 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 9 and 10 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 10: 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 4.5.

4.8 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 **??**.

4.9 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.

4.10 Maybe Monad Support

5 *HamSkill* Architecture

Software testing is important for detecting defects in software programs. What is more, there is different types of software testing including unit testing, module level testing, and system tests. Given that ANTLR relies heavily on the Listener pattern, it reduces the ease at which unit testing can be performed. What is more, given that *HamSkill* relies on the end to end operation of programs written in Haskell, ANTLR, Java, and Scala, it really emphasizes the importance of system level test.

5.1 System Level and Regression Testing via Use Cases

5.2 Testbench Implementation Overview

The *HamSkill* test architecture is written as a `bash` script (see the file in the submission `Test_Bench/test_bench.sh`. The function "`perform_hamskillStd_and_hamskillPlus_test`" is the most important in the script as it is responsible for performing most of the test; the basic operational flow of this function is:

- **Step #1**: The function is provided the name of a Haskell program written in the dialect supported by *HamSkill*, but without the file extension.
- **Step #2**: The Haskell program is run in Haskell, and the program's output is stored to a specific file on disk using the `bash` command "`>`".
- **Step #3**:

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] What is scala? — the scala programming language. http://www.scala-lang.org/what-is-scala.html. (Accessed on 03/30/2016).

[7] Remo H. Jansen. *Learning TypeScript*. Packt Publishing, 2015.

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

[9] 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).