Programming Language Survey Paper on Haskell

Kurt O'Hearn
Nicholas Olesak

April 20, 2013

Contents

1 Introduction

This paper provides an overview of the Haskell programming language.

1.1 Origin and History

In the early 1930s, Alonzo Church formulated a mathematical abstraction known as lambda calculus, which involves the description and evaluation of functions in mathematics (Haskell). Lambda calculus became the foundation for functional programming ("Haskell"). However, as numerous functional languages began to develop, all of which were similar in expressiveness and semantics, there became the need for a standard functional language to aid the spread of functional programming (Hudak 1). A meeting was held in 1987 at a Functional Programming Languages and Computer Architecture conference to address this need (Hudak 1). The meeting resulted in the formation of a committee to design a standard functional language. In 1990, the committee produced their result: a consolidation of existing functional programming languages into an open standard, common language to be used for functional programming research: Haskell (Hudak 3).

Haskell is named after Haskell Curry, a logician whose work greatly contributed to functional programming. It has a lambda as its logo to symbolize lambda calculus as its foundation. Today, Haskell is widely used in many different aspects of computing. From AT&T's automated processing of abuse complaints in its network security division to the National Radio Astronomy Observatorys implementation of core science algorithms, Haskell can be found in many aspects of modern industry ("Haskell").

1.2 Functional Programming Paradigm

Haskell is a pure functional language, which means that its functions cannot change values, but rather only accept and return values. Every function acts independently of other functions and does not change the state of anything outside of its scope. This is referred to as having no "side-effects," and it allows Haskell to be exceptionally optimized as well as to permit parallel processing. In addition, functions do not have to be declared in a particular order since they all exist independently of one another.

1.3 Application Domains

[TODO] [?]

```
module Main where
import Data.List

main :: IO ()
main = putStrLn "Hello World!" >> test

test = do
    print 123
```

2 Data and Control Abstractions

2.1 Data Abstractions

2.1.1 Data Types

Haskell includes many of the data types that are standard in most languages, such as Integers, Floats, Characters, and Boolean values.

```
let x = 2
let pi = 3.14
let c = 'c'
let isHaskellAwesome = true
```

Two more essential data types in Haskell are lists and tuples. Lists are the most used data type due to their expendability and the wide range of functions that are available to manipulate them. They can easily be created, traversed, concatenated, reversed, sorted, and more with standard, built-in functions. Lists, however, are homogeneous and can only contain elements of the same data type. Tuples, on the other hand, can be heterogeneous. They are useful when the types and exact number of values to be combined are known. Like lists, tuples also comes with a surplus of built-in functions, one of which is the ability to zip two lists into one by joining matching elements into pairs. Simple yet elegant code can be written which combines the attributes of lists, tuples, and the other various other data types in Haskell.

```
let evens = [2, 4, 6, 8, 10]          -- list
let t = (3.14, 'pi')                  -- tuple
```

Haskell also allows the definition Strings, which are represented by surrounding text with double quotes. Strings, however, are not actually a data type. Although they can be initialized using the double quote syntax, the compiler converts the string into a list of characters at compile time. The use of the String notation is simply syntatic sugar for programmers. It also helps increase the readability of code. In the following example, the data is stored in the representation of str1: a list of characters. However, Haskell allows the definition of a string using double quotes, as seen in str2, which is much more understandable.

```
let str1 = ['H','a','s','k','e','l','l','!']
let str2 = "Haskell!"
```

In addition to these basic data types, Haskell also offers type variables. Type variables are variables whose type has not been explicitly declared; rather, they can be any type. By defining a function without declaring data types, the variables in the function can be used as any type and the function can be polymorphic. In the following example, the function can handle a list of any type because its uses a type variable.

```
lastTwo :: [a] -> [a]
lastTwo a = [a !! ((length a) - 2), a !! ((length a) - 1)]
```

2.1.2 Type Checking

Type checking in Haskell is performed statically at compile time. Haskell uses type-declarations for functions, in which the types of parameters and return values are explicitly defined. Any use of a function in which parameters do not meet the defined function type is flagged by the compiler, and compilation will be aborted. In addition, ues type inferencing by context. The type system in Haskell deduces obvious types and makes them concrete. In the following example, although the type of myNum is unknown at compile time, the compiler infers that the Int type is desired because myNum is added to an Int.

```
let myStr = "2"
let myNum2 = (read myStr) + 2
```

If the context does not suggest the type, as in the example below, then an exception will result because the compiler cannot infer what the type is going to be and is unable to handle it.

```
let myStr = "2"
let myNum = read myStr
```

Haskell uses typeclasses to categorize the type system. Typeclasses define the behavior of a general type. A type must support the definition of its typeclass, but may also be more specific. For example, all of the operators in Haskell are categorized under the Eq typeclass. Other examples of typeclasses in Haskell include Ord, Show, Read, Enum, Num, Integral, and Floating. Typeclasses can be used to allow multiple types to be passed to a function. For example, to create a function that will accept both Int and Integer types, it's as simple as declaring its typeclass as Integral.

2.2 Control Abstractions

2.2.1 Expressions

- Function definitions

  ```
  greet :: String -> String
  greet name = "Hello, " ++ name ++ "!"
  ```

- Bindings: let/in (valid expression), where (locally bound to a syntactic construct)

  ```
  rectPrismVol :: (Num a) -> a -> a -> a -> a
  rectPrismVol length width height = 4 * (faceArea length width)
      + 2 * (baseArea length height)
  where   faceArea l w = l * w
          baseArea = l * h

  myExp =
      let e = 2.7182
      in  e ** x
  ```

  NOTE: statements in let/in must be aligned in same column

2.2.2 Operators and Precedence

- Operators: postfix notation, naming typically consists of symbols
- Functions: infix notation, named using alphanumeric characters in camel case
- Any function can be used as an operator by calling it in postfix notation

```
f :: (Int a) => a -> a -> a
f a b = a + b
; invoke the function in postfix notation
print (1 `f` 2)
```

- Any operator can be called as a function in infix notation

```
print ((+) 2 5)
```

- Operators can be overloaded

```
(*) :: (Num a) => a -> a -> a
(*) a b = a * a * b
print (2 * 3)
```

2.2.3 Selection Constructs

- Pattern matching

```
favoriteLang :: String -> String
favoriteLang "Haskell" = "You have refined taste, my friend."
favoriteLang "Clojure" =
    "The wise do not build a house on unsturdy ground."
favoriteLang lang = lang ++
    " is a fine choice, but you should give Haskell a spin."
```

- if/then/else

```
guessGame :: Int -> String
guessGame x = if x == 42
    then "You have a deep understanding of life."
    else "You need to think more about life."
```

- Guards

```
let balance = 200
assessWorth :: Int -> String
assessWorth =
    | balance >= 100000 = "You're rich!"
    | balance >= 100 = "You're rich by student standards!"
    | otherwise = "Need a loan, friend?"
```

- Case expression

```
favoriteMathSymbol :: String -> String
favoriteMathSymbol x = case x of
    "Pi" -> "Geometry is your mind's eye."
    "e" -> "May the calculus forever be with you."
    "Epsilon" -> "I think I am within the limit, "
        ++ "therefore, I am within the limit."
    x = "You have much to learn of the maths."
```

2.2.4 Iterative Constructs

- List comprehensions

```
filterModThree :: [Int] -> [Int]
filterModThree xs = [x | x <- xs, x `mod` 3 == 0]
```

- Map/fold
- Recursion

2.2.5 Functions

- Definition/use
- Parameter passing techniques

2.2.6 Scoping

- Static