Programming Language Survey Paper on Haskell

Kurt O'Hearn
Nicholas Olesak

April 17, 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. Each functions 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 and permit parallel processing. In addition, functions do not have to be declared in a particular order since they all exist independently of one another.

In addition, it is also a lazy language; expressions in Haskell are not evaluated at binding time, but rather are evaluated only when the result is

needed.

Type checking in Haskell is static and is performed 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. For example, the following function defines

1.3 Application Domains

[TODO] [1]

```
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

- Basis data types: Int, Float, Char, Boolean, lists, tuples

  ```
  let i = 2
  let x = 3.14
  let c = 'c'
  let str = ['H','a','s','k','e','l','l','!']
  ; shorthand for the above
  let str2 = "Haskell!"
  let t = (3.14, 'pi')
  ```

- Type variables: variables that can be any type (used in polymorphic functions)

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

2.1.2 Type Checking

- Haskell: static typing, type inferencing by context

```
let myStr = "2"
; will result in an exception as the desired type to convert to is
; ambiguous since Haskell has no context to infer the type
let myNum = read myStr
; not ambiguous, as the compiler infers the Int type is desired
let myNum2 = (read myStr) + 2
```

- Type classes: Eq, Ord, Show/Read, Bounded, Enum, Num/Integral/Floating

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

2.2.7 Modules

2.2.8 Exception Handling

3 Advanced Topics

3.1 Inheritance

3.2 Concurrency Support

3.3 Introspection

References

[1] M. Lipovaa. Learn you a haskell for great good! http://learnyouahaskell.com/, 2011.