Functional Programming in Haskell

The University of Glasgow

Get Unlimited learning

€23.99/month

TP

- Your learning
- Wishlist
- Profile
- Settings
- Account
- Sign out

Get Unlimited learning

€23.99/month

- To do
- Conversations
- Progress

Week 1

Haskell First Steps

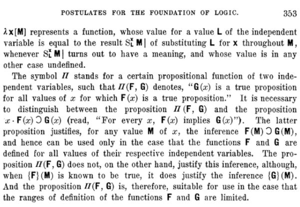
Introduction

1.1

Welcome to the Course

6.5

You've completed 4 steps in Week 6


POSTULATES FOR THE FOUNDATION OF LOGIC. 353

$\lambda \mathbf{x}[\mathbf{M}]$ represents a function, whose value for a value $\mathbf{L}$ of the independent variable is equal to the result $S^{\mathbf{x}}_{\mathbf{L}}\mathbf{M}|$ of substituting $\mathbf{L}$ for $\mathbf{x}$ throughout $\mathbf{M}$, whenever $S^{\mathbf{x}}_{\mathbf{L}}\mathbf{M}|$ turns out to have a meaning, and whose value is in any other case undefined.

The symbol Π stands for a certain propositional function of two independent variables, such that $\Pi(\mathbf{F}, \mathbf{G})$ denotes, "$\mathbf{G}(x)$ is a true proposition for all values of x for which $\mathbf{F}(x)$ is a true proposition." It is necessary to distinguish between the proposition $\Pi(\mathbf{F}, \mathbf{G})$ and the proposition $x \cdot \mathbf{F}(x) \supset \mathbf{G}(x)$ (read, "For every x, $\mathbf{F}(x)$ implies $\mathbf{G}(x)$"). The latter proposition justifies, for any value $\mathbf{M}$ of x, the inference $\mathbf{F}(\mathbf{M}) \supset \mathbf{G}(\mathbf{M})$, and hence can be used only in the case that the functions $\mathbf{F}$ and $\mathbf{G}$ are defined for all values of their respective independent variables. The proposition $\Pi(\mathbf{F}, \mathbf{G})$ does not, on the other hand, justify this inference, although, when $\{\mathbf{F}\}(\mathbf{M})$ is known to be true, it does justify the inference $\{\mathbf{G}\}(\mathbf{M})$. And the proposition $\Pi(\mathbf{F}, \mathbf{G})$ is, therefore, suitable for use in the case that the ranges of definition of the functions $\mathbf{F}$ and $\mathbf{G}$ are limited.

Excerpt from "Postulates for the foundation of Logic" by A. Church, 1932

1.10
Test Your Understanding
Quiz
1.11
Summary
Article
Finding Out More
1.12
Recommended Reading
Video
1.13
Spot the Difference
Discussion
1.14
End of Week 1
Video
Week 2
Haskell Building Blocks
More Basic Haskell
2.1
Welcome to week 2
Video
2.2
Do it Yourself: Boolean Values and Expressions
Exercise
2.3
Zip that List

Introduction to the Lambda calculus

0 comments

Introduction to the Lambda Calculus

- The lambda calculus was developed in the 1930s by Alonzo Church (1903–1995), one of the leading developers of mathematical logic.
- The lambda calculus was an attempt to formalise functions as a means of computing.

Significance to computability theory

- A major (really *the* major) breakthrough in computability theory was the proof that the lambda calculus and the Turing machine have exactly the same computational power.
- This led to *Church's thesis* — that the set of functions that are effectively computable are exactly the set computable by the Turing machine or the lambda calculus.
- The thesis was strengthened when several other mathematical computing systems (Post Correspondence Problem, and others) were also proved equivalent to lambda calculus.
- The point is that the set of effectively computable functions seems to be a fundamental reality, not just a quirk of how the {Turing machine, lambda calculus} was defined.

Significance to programming languages

- The lambda calculus has turned out to capture two aspects of a function:
  - A mathematical object (set of ordered pairs from domain and range), and
  - An abstract black box machine that takes an input and produces an output.
- The lambda calculus is fundamental to denotational semantics, the mathematical theory of what computer programs mean.
- Functional programming languages were developed with the explicit goal of turning lambda calculus into a practical programming language.
- The ghc Haskell compiler operates by (1) desugaring the source program, (2) transforming the program into a version of lambda calculus called *System F*, and (3) translating the System F to machine language using *graph reduction*.

Abstract syntax of lambda calculus

- We will work with the basic lambda calculus "enriched" with some constants and primitive functions (strictly speaking, that is not necessary).
- The language has constants, variables, applications, and functions.

```
exp
  = const
  | var
  | exp exp
  | \ var -> exp
```

Variables

- Each occurrence of a variable in an expression is either *bound* or *free*
  - In $\backslash x \rightarrow x+1$, the occurrence of x in $x+1$ is *bound* by the $\backslash x$.
  - In $y * 3$, the occurrence or y is *free*. It must be defined somewhere else, perhaps as a global definition.
- In general, an occurrence of a variable is bound if there is some enclosing lambda expression that binds it; if there is no lambda binding, then the occurrence if free.

We need to be careful: the first occurrence of a is free but the second occurrence is bound.

```
a + (\ a -> 2^a) 3  -- >  a + 2^3
```

Being free or bound is a property of an *occurrence* of a variable, not of the variable itself!

Conversion rules

- Computing in the lambda calculus is performed using three *conversion rules*.
- The conversion rules allow you to replace an expression by another ("equal") one.
- Some conversions simplify an expression; these are called *reductions*.

Alpha conversion

- Alpha conversion lets you change the name of a function parameter consistently.
- But you can't change free variables with alpha conversion!
- The detailed definition of alpha conversion is a bit tricky, because you have to be careful to be consistent and avoid "name capture". We won't worry about the details right now.

```
(\x -> x+1) 3
(\y -> y+1) 3
```

Beta conversion

- Beta conversion is the "workhorse" of lambda calculus: it defines how functions work.
- To apply a lambda expression an argument, you take the body of the function, and replace each bound occurrence of the variable with the argument.

```
(\x -> exp1) exp2
```

is evaluated as $exp1[exp2/x]$

Example:

```
(\x -> 2*x + g x) 42
```

is evaluated as $2 * 42 + g\ 42$

Eta conversion

- Eta conversion says that a function is equivalent to a lambda expression that takes an argument and applies the function to the argument.

```
(\x -> f x)
```

is equivalent to f

Example (recall that $(*3)$ is a function that multiplies its argument by 3)

```
(\x -> (*3) x)
```

is equivalent to $(*3)$

Try applying both of these to 50:

```
(\x -> (*3) x) 50
```

is the same as $(*3)\ 50$

Removing a common trailing argument

There is a common usage of Eta conversion. Suppose we have a definition like this:

```
f x y = g y
```

This can be rewritten as follows:

```
f = \x -> (\y -> g y)
f = \x -> g = f x = g
```

Thus the following two definitions are equivalent:

```
f x y = g y
f x = g
```

In effect, since the last argument on both sides of the equation is the same (y), it can be “factored out”.

Share this article:

-
-
-
-

0 comments

Previous

Mark as complete

Next

Mark as complete

Comments

Show:

2.12
What do you know about Haskell?
Quiz
2.13
End of Week 2
Video
Week 3
Data Structures and Types
Functions on Lists
3.1
Welcome to Week 3
Video
3.2
Recursive Functions on Lists
Article
3.3
Functional Maps and Folds versus Imperative Loops
Video
3.4
Do it Yourself: Lists and Recursion
Exercise
3.5
Do it Yourself: Function Composition
Exercise
3.6
What Have We Learned About Lists?
Quiz
3.7

All comments
g Book Generator

Sort by:

Newest

Filter

3.8

- Help Centre
- Child safety
- Privacy
- T&Cs

Summary

Article

Custom Data Types

Contact FutureLearn for Support

Our website is updated regularly so this content may now be out of date, please go to https://www.futurelearn.com for the most up to date information.

3.9

Define Your Own Data Types

Video

3.10

Grow a Tree

Article

3.11

Type Classes

Video

Haskell History

3.12

Interview with Simon Peyton Jones

Video

3.13

Brief History of Haskell

Article

3.14

Course Feedback

Article

3.15

End of Week 3