Functional Programming

SETU - South East Technological University
Imbriani Paolo - W20114452
Professor Mairead Meagher

February 4, 2025

Contents

1 Lambda Calculus, α-equivalence, β-reduction

Lambda Calculus is a model of computation devised in the 1930s by Alonzo Church. Functional Programming languages are all based on the lambda calculus. For example, Haskell, is a **pure** functional programming language, all its features are translatable to lambda expressions. It allows a higher degree of abstractios and composability.
The lambda calculus is based on the concept of a function, and it is a simple mathematical model of computation. We have seen the function:

$$\lambda x.x$$

The variable x here is not semantically meaningful except in its role in that single expression. Because of this, there's a form of equivalence between lambda terms called α-equivalence. Two lambda terms are α-equivalent if they differ only in the names of their bound variables.

$$\lambda x.x$$

$$\lambda apple.apple$$

$$\lambda orange.orange$$

all mean the same thing. When we apply a function to an argument we

- substitute the input expression for all instances of bound variables within the body of the abstraction
- eliminate the head of the abstraction (its only purpose was to bind a variable)

This is called β-reduction.

Example

$$(\lambda x.x)3 \to 3$$

The $\lambda x.x$ is a function that takes an argument and returns it. When we apply it to 3, we substitute 3 for x and eliminate the head of the abstraction. This is function is also called the *identity function.*

$$\lambda x.x + 1$$

$$(\lambda x.x + 1)3 \to 3 + 1 = 4$$

This other function takes an argument and returns it incremented by 1.

We can also apply our identity function to another lambda astraction:

$$(\lambda x.x)(\lambda y.y) \to \lambda y.y$$

We can use another syntax here, $[x := z]$ to indicate that z will be substituted for x in the expression. Here z is the function $(\lambda y.y)$.
We reduce this application like this:

$$(\lambda x.x)(\lambda y.y)$$

$$
\begin{aligned}
&[x := \lambda y.y] \\
&\quad \lambda y.y
\end{aligned}
$$

Once more, but this time we'll add another argument:

$$(\lambda x.x)(\lambda y.y)z$$

Applications of the lambda calculus are *left associative.* That is unless specific paranthesis suggest otherwise, they associate, or group, to the left. So, it can be rewritten as:

$$((\lambda x.x)(\lambda y.y))z$$

$$
\begin{aligned}
&((\lambda x.x)(\lambda y.y))z \\
&[x := \lambda y.y] \\
&(\lambda y.y)z \\
&[y := z] \\
&z
\end{aligned}
$$

1.1 Variables and multiple parameters

Variables can be:

- **Free** if they are not bound by a lambda abstraction
- **Bound** if they are bound by a lambda abstraction

as the λ-calculus assumes an infinite universe of free variables. They are bound to functions in an environment, then they become bound by usage in an abstraction. For example, the following lambda expression:

$$\lambda x.x * y$$

x is bound by λ over the body $x*y$, but y is a free variable. When we apply this function to an argument, nothing can be done with y. It remains irredicible.

Example

The following lambda expression:

$$(\lambda x.x * y)z$$

We apply the lambda to the argument z:

$$
\begin{aligned}
&(\lambda x.x*y)z \\
&\quad [x := z] \\
&\qquad zy
\end{aligned}
$$

The head has been applied to the argument, and the body has been reduced. Since we know nothing about y and z, we can't reduce it further.

Each lambda can only bind one parameter. To bind multiple parameters, we can nest lambdas. For example, the following lambda expression:

$$\lambda x.(\lambda y.x + y)$$

This is also called a *curried* function. It takes one argument, and returns another function that takes another argument. This is useful for partial application of functions.

Example

$$(\lambda x.(\lambda y.x + y))3$$

We apply the outer lambda to 3:

$$\begin{aligned}(\lambda x.(\lambda y.x + y))3 \\ [x := 3] \\ (\lambda y.3 + y)\end{aligned}$$

We have reduced the outer lambda to a function that takes an argument and returns it incremented by 3.

1.2 Lambda Calculus in Haskell

How do we write lambda expression in Haskell?

Named function	Lambda Calculus	Haskell	Result
Identity	$\lambda x.x$	`\x -> x`	`id`
f x = x + 1	$\lambda x.x + 1$	`\x -> x + 1`	`(+1)`
f x y = x * y	$\lambda x.\lambda y.x * y$	`\x -> \y -> x * y`	`(*)`
f xs = 'c' : xs	$\lambda xs.'c' : xs$	`\xs -> 'c' : xs`	`('c':)`

Lambda functions are used extensively in Haskell, notably with high order functions.

1.3 Encoding Lambda Calculus

- Alonzo Church is credited with the invention of the lambda calculus in the 1930s.
- Church encoding were developed to encode data structures and operations in the lambda calculus.
- Church encoding are a very powerful tool for reasoning about programs.
- Church found out that every concept in programming languages can be represented using functions.
- Everything from boolean logic, conditionals, statements, numbers and even loops can be represented using functions.

But how we encode all these construcs using only functions?
We'll encode the behaviour of these constructs using functions. For example, we can encode the boolean values `True` and `False` as functions that take two arguments and return the first or the second argument respectively.
We can encode the boolean value `True` as:

$$\lambda x.\lambda y.x$$

and the boolean value `False` as:

$$\lambda x.\lambda y.y$$

To encode the logic of the NOT operator, we can define a function that takes a boolean value and returns the opposite value.
The NOT operator can be encoded as:

Definition 1.1.

$$NOT := \lambda b.b \text{ False True}$$

To encode the logic of the AND operator, we can define a function that takes two boolean values and returns `True` if both are `True`, otherwise it returns `False`.
The AND operator can be encoded as:

Definition 1.2.

$$AND := \lambda x.\lambda y.xy \text{ False}$$

To encode the logic of the OR operator, we can define a function that takes two boolean values and returns `True` if at least one of them is `True`, otherwise it returns `False`.
The OR operator can be encoded as:

Definition 1.3.

$$OR := \lambda x.\lambda y.x \text{ True y}$$