The Lambda Calculus

From "Haskell Programming from First Principles"

Mairead Meagher

February, 2019

Lambda Calculus

- lambda calculus is a model of computation devised in the 1930s by Alonzo Church.
- Functional programming languages all based on the lambda calculus
- Haskell is a *pure* functional language because all its features are translatable into lambda expressions
- allows higher degree of abstraction and composability

Functions

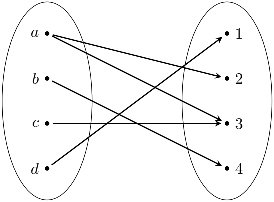


Figure: This is not a function

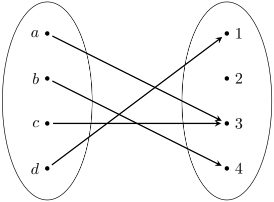


Figure: This is a function

Functions

- relationship of inputs and outputs is defined by the function
- output is predictable when you know the input and the function definition e.g. f:

$$f(x) = x + 1$$

Definition of lambda terms

BNF definition of the lambda calculus:

$$
\begin{aligned}
< \lambda\text{-term} > ::= & < \text{variable} > \\
& \mid \lambda < \text{variable} >.< \lambda\text{-term} > \\
& \mid (< \lambda\text{-term} >< \lambda\text{-term} >)
\end{aligned}
$$

$< \text{variable} > ::= x|y|z| \dots$

Or, more compactly:

$E ::= \ V \mid \lambda \ V.E \mid (E1 \ \ E2)$

$V ::= x \mid y \mid z \mid \ \dots$

Where

- V is an arbitrary variable and
- E_i is an arbitrary λ-expression.

We call λ V the head of the λ-expression and E the body.

Structure of lambda terms

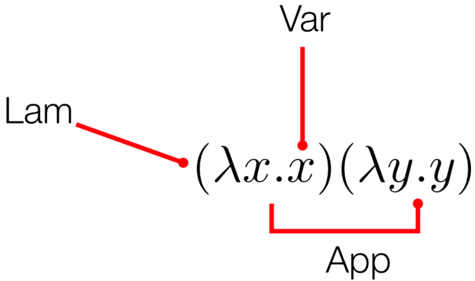


Figure: Structure of lambda terms

e.g. $(\lambda x.x)\ 2$

Alpha Equivalence

We have seen the function

$$\lambda x.x$$

There is a form of equivalence between lambda terms called alpha equivalence. So:

$$\lambda x.x$$
$$\lambda apple.apple$$
$$\lambda orange.orange$$

all mean the same thing.

Beta Reduction

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 process is called beta reduction.

More Beta reduction

$$\lambda x.x$$

- We apply the function above to 2
- substitute 2 for each bound variable in the body of the function, and
- eliminate the head:

$$(\lambda x.x)\ 2$$
$$2$$

The only bound variable is the single x, so applying this function to 2 returns 2. This function is the *identity* function.

More Beta reduction

Other Examples:

$$(\lambda x.x + 1)$$

What happens if we apply this to 2?
We can also apply our identity function to another lambda abstraction:

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

In this case, we substitute the entire abstraction in for x. We use a new syntax here, $[x := z]$, to indicate that z will be substituted for all occurrences of x (here z is the function $(\lambda y.y)$). We reduce this application like this:

$$\begin{array}{r} (\lambda x.x)(\lambda y.y) \\ [x := (\lambda y.y)] \\ (\lambda y.y) \end{array}$$

Our final result is another identity function. There is no argument to apply it to, so we have nothing to reduce.

More Beta reduction

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

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

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

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

The β-reduction is as follows:

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

We can't reduce this any further as there is nothing left to apply, and we know nothing about z.

Variables

Variables can be bound or free 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, in the λ-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 the y. It remains irreducible.

Variables contd.

Look at the following when we apply such a function to an argument:

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

We apply the lambda to the argument z.

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

The head has been applied away, and there are no more heads or bound variables. Since we know nothing about z or y, we can reduce this no further.

Multiple Arguments

Each lambda can only bind one parameter and can only accept one argument. Functions that require multiple arguments have multiple, nested heads. When you apply it once and eliminate the first (leftmost) head, the next one is applied and so on. This means that the following

$$\lambda xy.xy$$

is simply syntactic sugar for

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

Syntax Trees

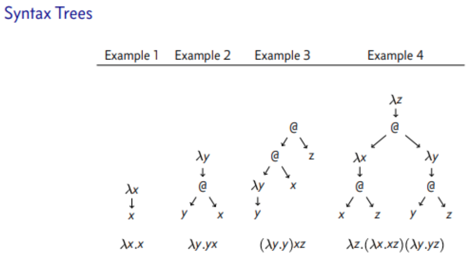
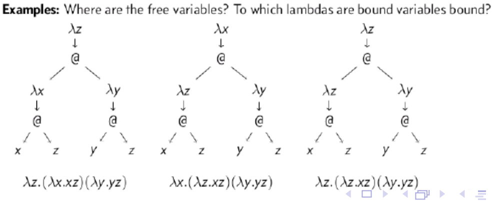


Figure: Examples of Syntax trees

Note that @ means function application.



Lambda Caculus in Haskell

How do we write lambda expressions in Haskell?

Named Function	Lambda Calculus (maths)	Lambda Calculus (Haskell)	Re
$f\ x = x + 1$	$(\ \lambda x.x + 1)\ 2$	$(\backslash x \rightarrow x + 1)\ 2$	
$f\ x\ y = x * y$	$(\ \lambda x\ y.x * y)\ 2\ 3$	$(\backslash x\ y \rightarrow x * y)\ 2\ \ 3$	
$f\ xs = `c` : xs$	$(\ \lambda xs.`c` : xs)\ \ "at"$	$(\backslash xs \rightarrow `c` : xs)\ \ "at"$	"

Lambda functions are used extensively in Haskell, notably with Higher Order Functions.