The Lambda Calculus

Mairead Meagher

February, 2019

1 Introduction

The lambda calculus is a model of computation devised in the 1930s by Alonzo Church. A calculus is a method of calculation or reasoning; the lambda calculus is one process for formalising a method.

Functional programming is a computer programming paradigm that relies on functions modelled on mathematical functions. The essence of functional programming is that programs are a combination of expressions. Expressions include concrete values, variables, and also functions. Functions have a more specific definition: they are expressions that are applied to an argument or input, and once applied, can be reduced or evaluated. In Haskell, and in functional programming more generally, functions are first-class: they can be used as values or passed as arguments, or inputs, to yet more functions.

Functional programming languages are all based on the lambda calculus. Some languages in this general category incorporate features into the language that aren't translatable into lambda expressions. Haskell is a pure functional language, because it does not.

The word purity in functional programming is sometimes also used to mean what is more properly called referential transparency. Referential transparency means that the same function, given the same values to evaluate, will always return the same result in pure functional programming, as they do in mathematics. Haskell's pure functional basis also lends it a high degree of abstraction and composability. Abstraction allows you to write shorter, more concise programs by factoring common, repeated structures into more generic code that can be reused.

2 What is a function?

If we step back from using the word "lambda," remember what a function is. A function is a relation between a set of possible inputs and a set of possible outputs. The function itself defines and represents the relationship. When you apply a function such as addition to two inputs, it maps those two inputs to an output:

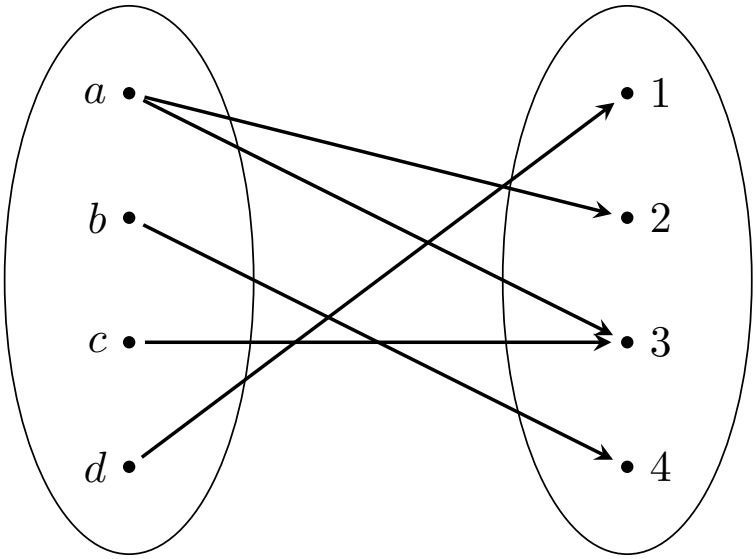

Figure 1: This is not a function

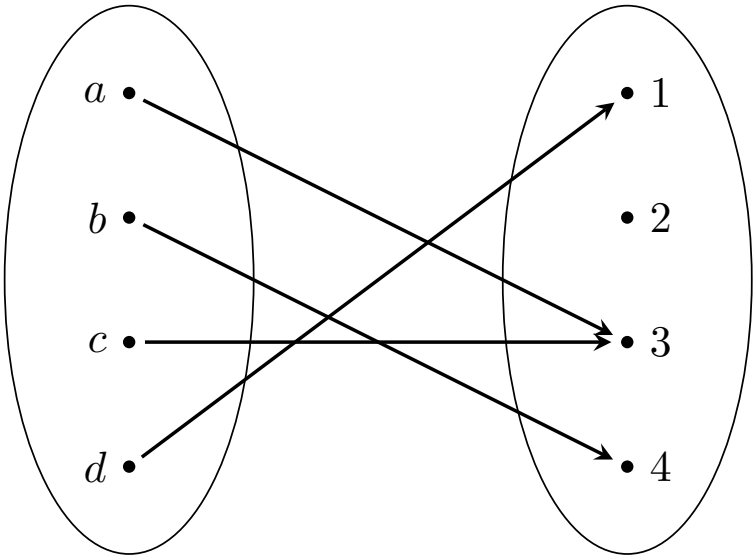

Figure 2: This is a function

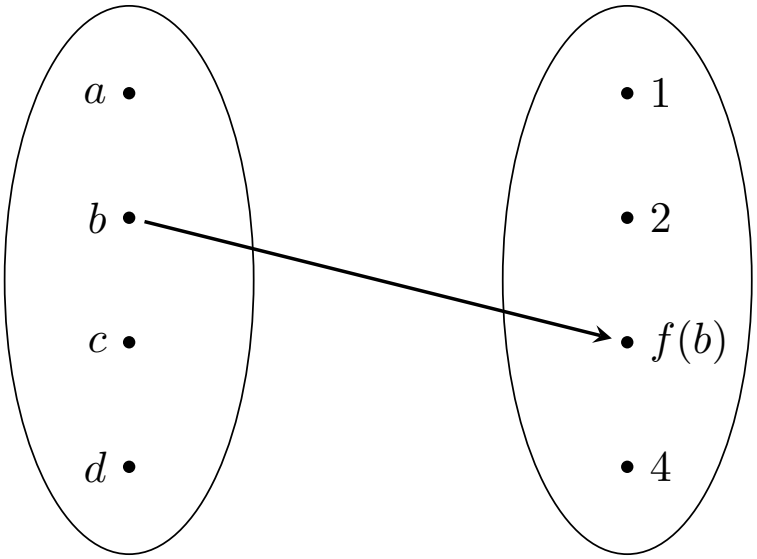

Figure 3: Function Application

What matters here is that the relationship of inputs and outputs is defined by the function, and that the output is predictable when you know the input and the function definition. The following example defines the relationship. This function is named f:

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

It is important to understand this structure.

3 The structure of lambda terms

BNF definition of the lambda calculus:

<λ-term >::= <variable >
| λ <variable >. <λ-term >
| (<λ-term ><λ-term >)

<variable >::= $x|y|z|\ldots$

Or, more compactly:

$E ::= V \mid \lambda\ V.E \mid (E1\ \ E2)$
$V ::= x \mid y \mid z \mid \ldots$

Where V is an arbitrary variable and E_i is an arbitrary λ-expression. We call λ V the head of theλ-expression and E the body.

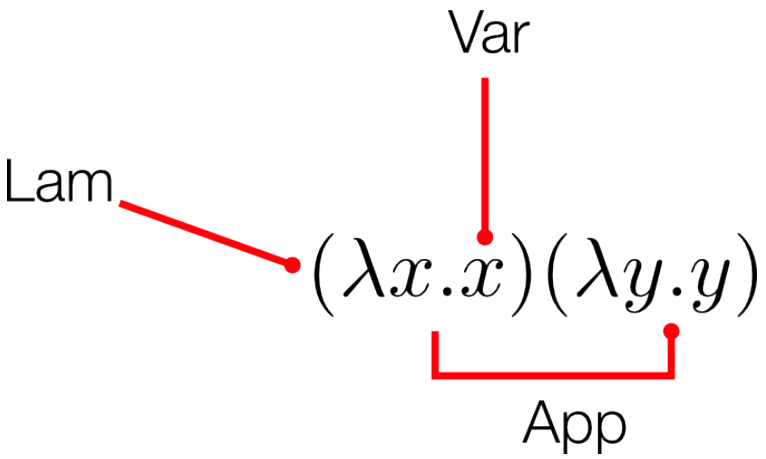

Figure 4: Structure of lambda terms

The head of the function is a λ (lambda) followed by a variable name. The body of the function is another expression. So, a simple function might look like this:

$$\lambda x.x$$

The variable named in the head is the parameter and binds all instances of that same variable in the body of the function. That means, when we apply this function to an argument, each x in the body of the function will have the value of that argument. We'll demonstrate this in the next section. In the previous section, we were talking about functions called f, but the lambda abstraction

$$\lambda x.x$$

has no name. It is an anonymous function. A named function can be called by name by another function; an anonymous function cannot.

4 Alpha equivalence

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 alpha equivalence. In other words,

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

all mean the same thing.

5 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. You also eliminate the head of the abstraction, since its only purpose was to bind a variable. This process is called beta reduction. Let's use the function we had above:

$$\lambda x.x$$

We'll do our first beta reduction using a number. We apply the function above to 2, substitute 2 for each bound variable in the body of the function, and eliminate the head:

$$\begin{aligned}(\lambda x.x)\ 2\\ 2\end{aligned}$$

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

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:

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

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

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

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

6 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.
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.

7 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

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

8 Syntax trees

In order to understand the structure of the terms, we look at syntax trees of lambda terms:

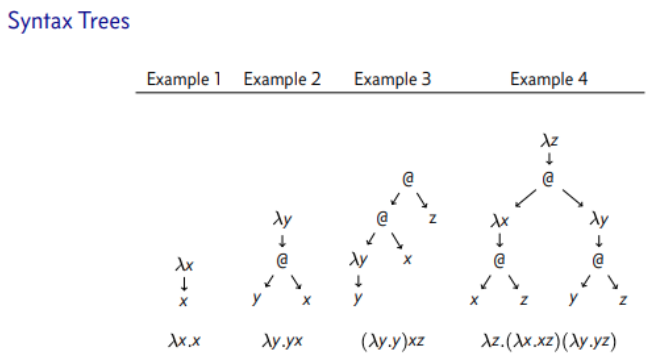

Figure 5: Examples of Syntax trees

Note that @ means function application.

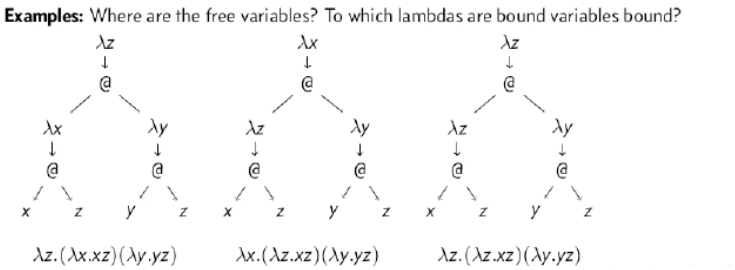

Figure 6: Syntax trees with free and bound variables

Can you figure which ar the free and bound variables from this diagram?