Combinators

Daniel Mandragona

02/22/17

1. The goal of this post is to give an introduction to combinators. Informally, combinators are functions that manipulate and combine other functions to produce a new function. Unfortunately, before we give a formal definition to these combinators, we must learn a few things.

2. Insert **Lambda Calculus**. λ-Calculus is a system for devising computation. The formality of this system allows the study of lambda calculus to inherit many tools from different mathematical areas. Lambda calculus has been proven to be Turing Complete, and so any programming language that is Turing Complete can be simulated using λ-Calculus.

   Okay, now we must learn what exactly λ-Calculus is. There are 3 rules:

   - A variable $\mathbf{x}$ is a valid λ-Calculus term. (We will show what this means quite soon).
   - If $\mathbf{f}$ is a valid λ-Calculus term, and $\mathbf{x}$ is a valid variable, then $\boldsymbol{\lambda}\mathbf{x}.\mathbf{f}$ is a valid term. (This is called a λ-abstraction).
   - if $\mathbf{f}$ and $\mathbf{g}$ are lambda terms, then $(\mathbf{fg})$ is a valid lambda term. (This is called an application).

   Now we get to learn how these rules work with everybody's favorite learning method, examples!

3. The first rule is about variables, so we will create two variables $\mathbf{x}$ and $\mathbf{y}$. Think of these in terms of math declarations, so if we have a function $f(\mathbf{x})$ then the $\mathbf{x}$ we just created will be the variable that f is acting upon. Similarly for the variable $\mathbf{y}$ in the function $g(\mathbf{y})$.

4. Now onto the second rule: **$\boldsymbol{\lambda}$-abstractions**. Directly above we just created these two variables x and y, but we never actually created the functions $\mathbf{f}$ and $\mathbf{g}$. Well what good are these variables if nothing is using them. Insert λ-abstractions, if $\mathbf{f}$ is a valid λ-Calculus term, and $\mathbf{x}$ is a valid variable, then $\boldsymbol{\lambda}\mathbf{x}.\mathbf{f}$ is a valid term. The f in the rule definition is essentially a function of x, and so $\lambda x.f$ is the function $\mathbf{f}(\mathbf{x})$.

$$\lambda x.f \quad \Rightarrow \quad f(x)$$

5. Now for the final rule. This rule is a bit more involved than it appears. So this means more examples. It says that if we have valid terms f and g then (fg) is also a valid λ term, but to me its name gives a better intuition for this rule. The name for this is an **application**, and that is because we are taking our valid term f and applying it with the specific input g. Think of this like the function $f(x) = x^2$ evaluated at a specific input, say $x = 4$. Then this becomes $f(4) = 4^2$. This will make more sense after a few examples:

   - Take the λ-abstraction $\lambda x.x + x$. This is a function that takes a variable x and returns $x + x$ or $2x$. Then the application of this abstraction with the variable y would take x and substitute in the variable y. This is illustrated by:

     $$(\lambda x.x + x)y \;=\; (x + x)[x := y] \;=\; y + y$$

     $$f(x) = 2x \;=\; f(x := y) = 2(x := y) \;= 2(y)$$

   - You can apply lambda functions with more than just variables too. Take the λ-abstraction $\lambda x.x\ x$. We can apply this with many different abstractions, but for now lets apply it with itself.

     $$(\lambda x.x\ x)\ (\lambda x.x\ x) \quad \rightarrow \quad (\lambda x.x\ x)\ [x := (\lambda x.x\ x)]$$

     $$= \quad (\lambda x.x\ x)\ (\lambda x.x\ x)$$

     This is an interesting example because we can see with an application such as this, the application never reduces or simplifies. We will discuss applications like this later.

   - The last example shall serve as a segue into our next topic that we must learn for our ultimate goal of understanding combinators. Take the application $(\lambda x.y)z$.

     This is an abstraction that takes a variable x and returns the variable y. In lambda calculus, there is no such thing as variable declaration, i.e. we don't know explicitly what x or y are, but we do know that this given abstraction does not use the x variable, because it always just returns this unknown y variable.

     $$(\lambda x.y)(z) \quad \rightarrow \quad y[x := z] \quad \Rightarrow \quad f(x) = y \quad \rightarrow \quad f(z) = y$$

6. In the last application example we dealt with this unknown y variable. Variables that are "unknown" in a λ-abstraction are referred to as **free** variables, while variables such as x in $\lambda x.2x$ are referred to as **bound** variables. The abstraction $\lambda x.x + y$ has free variable y and bound variable x.

   Free variables can be defined more formally in a similar manner to our λ-Calculus definition.

   - The set of free variables in the variable x is just x. (This is a little weird but from our above definitions we can see that x by itself is not bound by anything.)
   - The set of free variables in $\lambda x.f$ is the set of free variables in f except x. Hence if $f = wxyz$ then the set of free variables is $\{w, y, z\}$ since x is bound by this lambda definition.
   - The set of free variables in the application fg is the union of free variables of f and g. For example if $f = \lambda x.x\,w$ and $g = \lambda y.x\,w$ then the set of free variables in f is w and for g it is x, w so the resulting set is the union of these which is$\{x, w\}$.

7. Moving forwards we are almost at our goal of learning combinators. We just need to learn one more concept, ***β*-Reduction**. We already used β-Reduction slightly above in the second application example of $(\lambda x.x\,x)(\lambda x.x\,x)$. Essentially β-Reduction is given a series of abstractions and application, reducing this series to it's simplest form. Relating to math, if we have functions $f(x) = x$, $g(x) = x^2$, and $h(x) = x$ then $f(g(h(x))) = f(g(x)) = f(x^2) = x$. The function composition $f \circ g \circ h = x^2$ in a sense β-reduces to a function say, ϕ, where $\phi(x) = x^2$. Going back to our λ expressions here are a few step-by-step examples.

   -

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

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

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

  We saw earlier how this application never stops, or never reduces. λ-expressions are not guaranteed to reduce. We can think about this in a programming sense as non-terminating recursion. In this view we also see that each step of β-reduction can be interpreted as a step of computation. The motivation behind β-reduction is that it shows that some λ expressions are equivalent, such as the earlier example with function $\phi(x)$ being equivalent to $g(x)$.

8. Well we are finally here! We now have all the necessary knowledge to formally learn about combinators. **Combinators** are functions that possess only bound variables. Combinators do not have free variables. These functions produced a field of study called Combinatory Logic. An interesting fact about λ-Calculus is that although every lambda function is created by means of abstraction and application, abstraction is not required. Because of this fact, a similar calculus called combinatorial calculus was created. This calculus is computationally equivalent to λ-Calculus and thus Turing Complete, but only relies on combinators for means of computation. There are many different important combinators that we will now discuss.

   - The identity combinator, **I**. $\lambda x.x$. If we apply this combinator with any term t then we get t back. This combinator is usually denoted as $Ix = x$ or $I(x) = x$.
   - The next combinator is the constant combinator, **K**. $\mathbf{K} = \lambda x.(\lambda y.x)$. This combinator takes two arguments x and y, and returns x regardless of what the argument y is. This is why it is referred to

as the constant combinator. For all terms x and y in the lambda calculus, $\mathbf{K}\,x\,y = x$ or in alternative notation $\mathbf{K}(x, y) = x$.

- The third combinator that we will cover is the substitution combinator, $\mathbf{S} = \lambda x.[\lambda y.[\lambda z.(xz)(yz)]]$. This combinator takes three arguments,x, y, z, and begins by applying the first two arguments with the third and then combines these two results. We need this combinator to make up for the lack of the abstraction tool in our combinatorial calculus. ????NOT SURE IF THIS IS TRUE THIS IS JUST A GUESS(need to check with fontaine)???? The reason why **S** is able to help us is that our current calculus is left associative, but certain computations require right associativity. Without **S** we could attempt to create the result of this combinator with just $xzyz$, but by association this would become $(((xz)y)z)$.
- We can solve this abstraction issue with different combinators though. Consider X(X(X(XX))) = X(X(X[(((xS)KS)K)] [(((((XS)KS)KS)KS]K

•