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 denoted by $x := y$ or "x defined to be y"). Now this application is illustrated by:

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

     Equivalently,

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

   - 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.

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

     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. 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.

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

6. In the last application example we dealt with this unknown y variable. Well that y is defined to be a free variable, and so in this section we will cover **free** and **bound** variables. But before we can do this, we must fist learn about scope.

   Scope is used in reference to λ-abstractions. The **scope** of a λ-abstraction, λx, in $\lambda x.M$ is the term M.

   - The scope of λx in $(\lambda x.x\ y)(xy)$ is $x\ y$, but not (xy).
   - The scope of λy in $\lambda y.((\lambda x.y)x)$ is $((\lambda x.y)x)$, and the scope of λx is y.
   - The scope of the *leftmost* λx in $\lambda x.(\lambda y.(\lambda x.x\ z\ y)y))$ is $(\lambda y.(\lambda x.x\ z\ y)y)$, while the scope of the *rightmost* λx is $x\ y\ z$.

   Now onto bound and free variables. A variable x in $\lambda x.M$ is said to be **bound** if it occurs inside the scope of λx, or it is the x in $\lambda \mathbf{x}.M$. If x occurs elsewhere then x is a **free** variable.

   - Consider the term $\lambda x.x\ y$. The variable x is bound, while y occurs as a free variable.
   - In the term $(\lambda x.y)(x)$ the *leftmost* x is bound (even though it does not actually occur in its scope), while y and the *rightmost* x are free.
   - In the term $\lambda x.\ (\lambda y.\ x(\lambda x.\ y))$ all occurrences of x and y are bound.

7. In this section we will learn about **β-Reduction**. We already used β-Reduction above in the second application example of $(\lambda x.x\,x)(\lambda x.x\,x)$. Essentially β-Reduction is when given a series of abstractions and applications, reducing this series to its simplest form. Relating back to math, if we have functions $f(x) = x$, $g(x) = x^2$, and $h(x) = x$ then $f(g(h(x))) = f(g(x)) = f(x^2) = x^2$. The function composition $f \circ g \circ h = x^2$ in a sense β-reduces to a function, $\phi(x)$, where $\phi(x) = x^2$. Going back to our λ expressions here are a few step-by-step examples.

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

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

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

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

  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. We will now talk about how λ-calculus relates to currying and partial applications that we see in Haskell.

   Partial application of a function is the ability to pass less than the required number of arguments to this function. We've seen this before in the Haskell function:

   ```
   addTwo :: Int → Int → Int
   addTwo x y = x + y
   ```

Well, if we called addTwo on the value 5, then this will return a function that takes a *single* integer, y, and returns $5 + y$. Such a function would be of the form:

$$\begin{aligned} &\text{addTwo} :: \text{Int} \to \text{Int} \\ &\text{addTwo y} = 5 + \text{y} \end{aligned}$$

In λ-calculus it is easy to see how this works. addTwo is represented by the λ-function:

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

Then the partial application of addTwo 5 is equivalent to

$$\begin{aligned} &(\lambda x.\, (\lambda y.\, x + y))(5) \text{ which } \beta\text{-reduces to} \\ &= (\lambda y.\, x + y)[x := 5] \\ &= (\lambda y.\, 5 + y) \end{aligned}$$

We now see that addTwo 5 is equivalent to the function $(\lambda y.\, 5 + y)$. More importantly going back to the whole function $(\lambda x.\, (\lambda y.\, x + y))$ we see that calling this function on two integers m and n requires two steps of β-reduction.

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

So the result of the first step of β-reduction is the partial application of this lambda function with our m. The next step grants us our final result.

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

Here are some examples of using the lambda calculus in order to do simple computations:

$$3 + 5 = (\lambda mn.m(m(m(n))))(\lambda pqr.q(pqr))(\lambda xy.x(x(x(x(x(y))))))$$

and

$$\begin{aligned}
2 \times 3 &= (\lambda pqr.p(qr))(\lambda mn.m(m(n)))(\lambda xy.x(x(x(y)))) \\
&= \lambda r.(\lambda mn.m(m(n)))((\lambda xy.x(x(x(y))))r) \\
&= \lambda rn.((\lambda xy.x(x(x(y))))r)(((\lambda xy.x(x(x(y))))r)(n)) \\
&= \lambda rn.(\lambda y.r(r(r(y))))(r(r(r(n)))) \\
&= \lambda rn.r(r(r(r(r(n))))) \\
&= 6
\end{aligned}$$

Currying is the process of taking a function that receives multiple arguments, and reducing this function to a series of functions that take only a single argument. Haskell is naturally currying, and so is λ-calculus. In the addTwo example we saw how the representative λ-function handled two arguments. Calling this original function resulted in two functions. The first function took the first argument, and returned a new function that took in a second argument of its own. The second function is able to add this second argument to the bound variable granted to it by the first function.

Here is another example of currying with a λ-function.

$$\lambda x\, y\, z.\, (x * y * z) \quad \equiv \quad \lambda x.\, (\lambda y.\, (\lambda z.\, (x * y * z)))$$

As we can see **currying** and **partial application** are closely resulted, and in a sense currying is made possible by partial application.

- Hindley, J. Roger, and Jonathan P. Seldin. "Lambda-Calculus and Combinators, an Introduction." (2008): n. pag. Web.
- Tromp, John. "Binary Lambda Calculus and Combinatory Logic." Randomness and Complexity, From Leibniz to Chaitin (2007): 237-60. Web.
- Berendregt, H. P. The Lambda Calculus: Its Syntax and Semantics. Amsterdam: North-Holland, 1985. Web.