# The Lambda Calculus

Simon Charlow (simon.charlow@rutgers.edu)

September 28, 2015

## 1 Motivation

Ways of specifying functions that we've seen:

- Prose: the function $f$ such that for any $x$, $f$ applied to $x$ is …
- "Combinator-style": $f(x) = \ldots$

What if we'd like to represent a fancier function? Say, for example, we wished to define a function $f$ that took another function $g$ and returned the result of applying $g$ to the identity function. How would we do this? Well, if we had a name for the identity function, e.g. IDENT, we could just write this:

$$f(g) \coloneqq g(\textsc{ident})$$

However, what if we hadn't gone to the trouble of defining IDENT before? Awkward!

The $\lambda$-calculus is a super-compact, super-powerful formal language for defining functions. For example, using $\lambda$s we can define the $f$ above like so (thus, functions can, but **needn't**, have names):

$$f \coloneqq \lambda g.\, g\,(\lambda x.\, x)$$

Like any proper formal language, the $\lambda$-calculus has a syntax and semantics. We'll specify these in turn (though our discussion of the semantics will be somewhat roundabout, in that we'll be talking about equivalence classes of $\lambda$-terms rather than their interpretations *per se*)…

## 2 Syntax

This is easy. An expression in the $\lambda$-calculus can either be a **variable**, a **function** (characterized by **abstraction**), or an **application** of two functions. In BNF notation:[1]

$$E ::= V \mid (\lambda V.\, E) \mid (E_1\, E_2)$$

[In practice, this is a little too limiting: we'll also admit a number of constants (e.g., numbers and previously-defined names for functions).]

Notice that parentheses are introduced both when we build abstractions and applications. Keeping track of expression grouping is super important in the $\lambda$-calculus. E.g., $(\lambda V.\, E_1)\, E_2 \neq \lambda V.\, E_1\, E_2$.

Explaining the horns of the BNF, from left to right:

- Variables are expressions in the $\lambda$-calculus. They exist to be **bound** by $\lambda$s (more on this in a second).

[1] N.B.: my syntax differs slightly from Gordon's. Can you see where? Can you see why?

- Abstractions allow us to represent functions **anonymously**. That is, we can just articulate a function on the fly; no need to have already defined it and given it a name. Intuitively, $\lambda V.\, E$ represents the function from arguments $V$ into results $E$ (we'll get more rigorous shortly). A bit of terminology that will be important in the next section: given $\lambda V.\, E$, we will call $E$ the **body** of $\lambda V$.
- Applications fittingly involve a function $E_1$ applying to an argument $E_2$.

Examples of well-formed $\lambda$-terms:

- $x$
- $(x\, y)$
- $(\lambda x.\, x)$
- $(\lambda x.\, (\lambda y.\, x))$
- $((\lambda f.\, (f\, (\lambda x.\, x)))\, (\lambda \kappa.\, (\kappa\, z)))$

Some widely-used abbreviatory conventions:

- Application associates to the left: instead of writing $((E_1\, E_2)\, E_3)$, we may simply write $E_1\, E_2\, E_3$.
- As in propositional logic, we'll omit parentheses whenever doing so doesn't create ambiguity. E.g., we will write $\lambda x.\, \varphi$ as a standalone expression rather than $(\lambda x.\, \varphi)$.
- A sequence of abstractions can be abbreviated: $(\lambda x.\, (\lambda y.\, \varphi))$ can be alternatively given as $\lambda x.\, \lambda y.\, \varphi$ or $\lambda xy.\, \varphi$. I would recommend using the $\lambda x.\, \lambda y.\, \varphi$ style.[2]

Many people write applications as $E_1(E_2)$, rather than $E_1\, E_2$. That is fine, and you're certainly allowed to use this convention if you wish. Just make sure expressions are unambiguous, and they mean what you want them to. This takes practice, but once you have the hang of it, you'll find it hard to believe you ever didn't.

## 3 "Semantics"

We're not quite in a position to give a real semantics for the $\lambda$-calculus. But we can still get a feeling for the meaning of expressions by giving syntactic rules for when two $\lambda$-expressions are equivalent. There are **three** core equivalences in the $\lambda$-calculus: $\alpha$, $\beta$, and $\eta$. We'll consider $\beta$ and $\eta$ now, and $\alpha$ in the next section.

The heart of the $\lambda$-calculus is the notion of $\beta$-equivalence (often called $\beta$-reduction), defined as follows:

$$(\lambda V.\, E_1)\, E_2 \underset{\beta}{\longrightarrow} E_1[E_2/V]$$

Where $E_1[E_2/V]$ is the expression just like $E_1$, but where every **free** occurrence of $V$ has been replaced by $E_2$.

$\beta$-conversion is intended to encode the simplification that happens when you apply a function to its argument. For example, here's what happens when you apply the identity function to some variable $y$:

$$(\lambda x.\, x)\, y \underset{\beta}{\longrightarrow} y$$

The definition refers to the difference between free and bound variables. A variable is **bound** in an expression $E$ iff within $E$, $\nu$ occurs in the body of a $\lambda\nu$, and is **free** otherwise. A simple example:

$$\overbrace{\lambda x.\, \underbrace{f(x)}_{x \text{ is free}}}^{x \text{ is bound}}$$

[2] The super-compact style is great, but less familiar in the field.

Because $\beta$-reduction involves replacing only the free occurrences of a variable with the argument, higher occurrences of $\lambda$s will be trumped by lower occurrences:

$$(\lambda x.\, \lambda x.\, x)\, y\, z \xrightarrow[\beta]{} (\lambda x.\, x)\, z \xrightarrow[\beta]{} z$$

To re-emphasize: in an abstraction $\lambda V.\, E$, $\lambda V$ only binds variables in $E$. This is important:

$$(\lambda x.\, x)\, x \neq \lambda x.\, x\, x$$

A second notion, $\eta$-equivalence, encodes the principle of **extensionality**, whereby two functions are equivalent iff they return the same values for every argument:

$$f \xleftrightarrow[\eta]{} \lambda x.\, f\, x$$

So, for example, applying $f$ to some argument $a$ yields $f\, a$, which is no different from what you get when you apply $\lambda x.\, f\, x$ to $a$. (Notice that $\eta$-equivalence goes in both directions, while $\beta$-equivalence — which is crucially about expression simplification — may be more intuitive to think of as going in a single direction, i.e., from more less simple to simpler.)

## 4 Variables

A $\beta$-equivalence is only considered **valid** if it does not result in any more variables being bound than were bound prior to the simplification. For example, by these lights, the following is not valid:

$$(\lambda x.\, \lambda y.\, x\, y)\, y \xrightarrow[\beta?]{} \lambda y.\, y\, y$$

Why not? Well, there's a $y$ that was free before the simplification that becomes bound after the simplification. This is known as **variable capture**, and it's **really bad**. Consider, after all, what happens if we embed the expression above in a larger one:

$$\lambda y.\, \underline{(\lambda x.\, \lambda y.\, x\, y)\, y} \xrightarrow[\beta?]{} \lambda y.\, \underline{\lambda y.\, y\, y}$$

If the prior simplification were valid, then this one should be too (since the underlined parts are the same). But clearly this is no good. If you apply the function on the left of the arrow to $a$, you end up with $(\lambda x.\, \lambda y.\, x\, y)\, a$, which reduces to $\lambda y.\, a\, y$. But if you apply the function of the right to $a$, you end up with $\lambda y.\, y\, y$. No!!!

So are we stuck? We aren't. $\alpha$-equivalence to the rescue:

$$\lambda V.\, E \xrightarrow[\alpha]{} \lambda V'.\, E[V'/V]$$

What this rule says is that, e.g., $\lambda x.\, x$ and $\lambda y.\, y$ are **the same function**. This should seem quite intuitive, since for any argument $a$, $(\lambda x.\, x)\, a = (\lambda y.\, y)\, a = a$.

This lets us avoid the problem of variable capture in the prior example:

$$(\lambda x.\, \lambda y.\, x\, y)\, y \xrightarrow[\alpha]{} (\lambda x.\, \lambda z.\, x\, z)\, y \xrightarrow[\beta]{} \lambda z.\, y\, z$$

The first $\alpha$-equivalence allows us to execute a $\beta$-reduction without ending up with a variable capture.

## 5 Order of reduction and normal form

Given what we've said so far, we are able to define the following function:

$$\Omega \coloneqq (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

Try to $\beta$-reduce this one. I'll wait…

That's right, you can't simplify $\Omega$. The "reduction" just gives you what you had before! This expression **doesn't terminate**. In other words, it **diverges**.

Now consider the following:

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

Hm. Looks like we have a choice of which expression to $\beta$-reduce — the application on the left, or the application on the right. We could either do the left one, in which case we end up with $z$. Or we could do the right one, in which case we end up with…the right one all over again!

So it turns out that which term we pick to reduce makes a difference. But don't despair! This turns out to be so **only** for non-terminating cases. However, for **any** terminating $\lambda$-expression, order of reduction makes no difference. For example, the following has the same choice-point (reduce left or right?), but as you can check for yourself, it makes no difference which one we pick to simplify first:

$$(\lambda f.\, \lambda x.\, f\, x)\, ((\lambda f.\, \lambda x.\, f\, x)\, g)$$

When all possible $\beta$-reductions have been performed, a $\lambda$-expression is said to be in **normal form**. Because of the aforementioned property, the terminating expressions of the $\lambda$-calculus are **confluent**: every one has exactly one normal form. This is known as the **Church-Rosser Theorem**.