Lambda Calculus with Ω

Some basic things and their equivalence in Omega.

Oscar Riveros

June 18, 2014

1 Syntax

The λ-calculus is the formal system of computation upon which functional programming languages such as Omega are based. Syntactically, expressions are built up from three kinds of terms:

$$\begin{array}{rcl} \mathit{Expr} & ::= & \mathit{Var} \\ & | & \lambda\, \mathit{Var}\ .\ \mathit{Expr} \\ & | & \mathit{Expr}\, \mathit{Expr} \end{array}$$

Here *Var* represents variables like x, y, z, and so on. Terms of the form $\lambda x.E$ are *anonymous* functions, known as λ-*abstractions*; the meaning is analogous to the Omega notation (`'(lambda (x) (E))`), where `E` is some subexpression. We call it a λ-abstraction because we use the symbol λ to indicate that we're *abstracting* the expression `E` over *all possible values* of the variable `x`. And thirdly, again like in Omega, we can write two expressions next to each other, ($\texttt{E}_1$ $\texttt{E}_2$), to indicate that the first (evaluates to) a function which is applied to the second.

2 Evaluating λ Expressions

Here's an example of a λ expression:

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

$$\Omega \rightarrow ((\texttt{'(lambda (x y) (y x))})\ (\texttt{'(lambda (z))})\ (\texttt{z})) \rightarrow \texttt{z}$$

The above expression consists of a function, $\lambda x.\lambda y.x$, applied to an argument—which in this case, is another function, $\lambda z.z\ z$. Informally, we would reduce the function application to the following: remove the "λx." and replace the occurrences of x in the body by the argument value, $\lambda z.z\ z$

$$\lambda y.(x[x \leftarrow \lambda z.z\ z])$$

Where the expression in brackets is a *substitution,* much like those we dealt with in unification. Thus the above finally simplifies to the following expression.

$$\lambda y.\lambda z.z\ z$$

Note that it reduced to a function; if we wanted to, we could apply this function to further arguments, again, much like in Omega.

3 Equivalence Rules

We can formalize what we did in the previous section, summing it up in terms of three equivalence rules. Equivalence rules specify when two different expressions can be read as equivalent. We can then choose a specific way to apply the equivalence rules to arrive at a simplification (or evaluation) algorithm for the λ-calculus.

3.1 α-equivalence

$$(\alpha) \quad \lambda x.E \quad \equiv_\alpha \quad \lambda y.(E[x \leftarrow y]) \quad y \notin \mathsf{fv}(E)$$

$$\Omega \rightarrow (= ((\prime(\texttt{lambda}\ (\texttt{x})\ \texttt{E}))\ ((\prime(\texttt{lambda}\ (\texttt{y})\ (\texttt{E}\ \texttt{y}))))) \rightarrow \texttt{true}$$

Our notion of α-equivalence is just *bound variable renaming:* if we consistently change the name of a local variable in a function, it doesn't change the behavior of the function at all. Note that we have to be careful not to change x to another variable z already appearing in E, as that would change the meaning of the function. Actually, more specifically, we just can't change x to another *free variable* in E; this is what the stipulation "$y \notin \mathsf{fv}(E)$" says. The idea is that the following equivalences hold:

$$\begin{aligned} \lambda x.x &\equiv_\alpha \lambda y.y \\ \lambda x.(\lambda w.f\ w)x &\equiv_\alpha \lambda y.(\lambda w.f\ w)y \end{aligned}$$

But the following is not an equivalence.

$$\lambda x.f\ x \quad \not\equiv_\alpha \quad \lambda f.f\ f$$

Note that in some cases, we may need to perform multiple α conversions to support a particular choice of a replacement for a bound variable. For example, say we want to change x to w below; then we first need to α-rename the occurrences of w to something else—say, u.

$$\begin{aligned} \lambda x.(\lambda w.w\ x)f &\equiv_\alpha \lambda x.(\lambda u.u\ x)f \\ &\equiv_\alpha \lambda w.(\lambda u.u\ w)f \end{aligned}$$

3.2 β-equivalence

$$(\beta) \quad (\lambda x.E_1)E_2 \quad \equiv_\beta \quad E_1[x \leftarrow E_2]$$

$$\Omega \rightarrow (= ((\prime(\texttt{lambda (x) E1})) \texttt{ E2}) (\texttt{E1 E2})) \rightarrow \mathit{true}$$

The core of function evaluation and simplification is β-equivalence, which can also be read as β-reduction, where we only replace expressions of the form on the lefthand side of the rule by expressions of the form on the righthand side. The basic idea of β-reduction has already been demonstrated in the examples above: given a function $\lambda x.E_1$, we read this as "a function that takes an argument x and then returns the value of E_1, with argument provided bound to the the variable x." Thus when we apply this function to some expression E_2, *i.e.,* the expression $(\lambda x.E_1)E_2$, what we can simplify the expression to "E_1, but with all free occurrences of x replaced by E_2." This is specified using the notation "$E_1[x \leftarrow E_2]$". Note that while occurrences of x in $\lambda x.E_1$ are *bound* by the "$\lambda x.$", if we look at E_1 alone, there may be some free occurrences. For example, if $E_1 = x\ x$, then both occurrences of x in E_1 are free; but they are considered *bound* when we look at the expression $\lambda x.x\ x$.

3.3 η-equivalence

$$(\eta) \quad \lambda x.Fx \quad \equiv_\eta \quad F$$

$$\Omega \rightarrow (= ((\prime(\texttt{lambda(x) (Fx)}))\texttt{(F)}) \rightarrow \texttt{true}$$

The concept of η-equivalence is a little bit different from α and β, because it's not generally used when evaluating expressions as in a program interpreter, but only for proving that different λ-calculus expressions are equivalent. SIDE NOTE: *in practice, you might need to prove equivalences if you're defining a new programming language and want to show that it works in some consistent fashion, or if you're designing a critical system, such as a controller for an aircraft or nuclear power plant, and want to show that it adheres to its specifications.*

The core idea of η-equivalence is simply that if you have an expression F, you can replace it by a function that takes a single argument x and applies F to it (i.e., wrap F up in a *lambda*-abstraction: $\lambda x.F\ x$). Essentially, what this is doing is delaying evaluation of F until we have an argument for it. You can also go the other direction: if you have a function of the form $\lambda x.F\ x$, replace it directly by the expression F.

4 Notes

This material was found with google, I just put the equivalences with Omega, I love to write my own material and in English, but my English is acquired and not learned, and grammatical errors would be counterproductive, but otherwise, it this is very basic material and the WWW is full of these things. I do not think that has any importance.