15-251: Great Theoretical Ideas in Computer Science 15251 Staff
Notes on The Lambda Calculus (draft!!) January 29, 2012

1 Syntax

Lambda expressions are defined recursively as follows:

Variable: $a, b, c, \ldots$ are lambda expressions.

Abstraction: If e is a lambda expression then $\lambda x.e$ is a lambda expression for any variable x. (It is a function taking one argument x and with body e.) When we have multiple lambdas in a row we abbreviate as follows, $\lambda xy.e := \lambda x.\lambda y.e$.

Application: If e_1 and e_2 are lambda expressions then $e_1\ e_2$ is a lambda expression. (It is applying e_2 as an argument to e_1.)

Generally, we are only interested in **closed lambda expressions**. These are lambda expressions in which every variable that appears is defined by a lambda to its left. (This is analogous to Java programs; we are only interested in Java programs that define all their variables.)

For example, the term $\lambda xy.x = \lambda x.\lambda y.x$ is a closed lambda expression since the only variable x is defined by a lambda. The sub-expression $\lambda y.x$ is *not* closed since the variable x in this sub-expression has not been defined. In the sub-expression x is said to be a **free variable**.

2 Scope and Parenthesis

Applications associate to the left, i.e., $e_1e_2e_3 = (e_1e_2)e_3$. Lambdas associate to the right, i.e., $\lambda x.e_1e_2 = \lambda x.(e_1e_2)$.

For example: If we fully parenthesize the expression

$$\lambda xy.x(\lambda z.zy)yy\lambda z.xy(xz)$$

we end up with

$$\lambda xy.((((x(\lambda z.(zy)))y)y)\lambda z.((xy)(xz)))$$

When two lambdas define the same variable name, the second lambda shadows the first, i.e., the expression $\lambda y\lambda y.y$ is alpha equivalent to $\lambda y.\lambda x.x$, not to $\lambda x.\lambda y.x$. This is just like Java where local variables shadow instance variables.

3 Alpha Reductions

Alpha reductions are very simple. All they are is renaming variables. Alpha reductions can be done at any time without changing the meaning of a lambda expression.

$$\lambda xy.x = \lambda zy.z = \lambda ax.a = \lambda xa.x = \ldots$$

If one term can be alpha reduced to another we say the terms are alpha equivalent (since we can trivially reduce in the other direction as well). Generally we only care to distinguish terms that are not alpha equivalent, i.e., we say that two terms that are alpha equivalent are equal and terms that are not alpha equivalent are not equal.

4 Beta Reductions

All beta reductions follow the rule

$$\frac{(\lambda x.e_1)e_2}{e_1[e_2/x]}$$

which means that if we have a lambda expression (or sub-expression) of the form $(\lambda x.e_1)e_2$, then we can reduce that expression or sub-expression to $e_1[e_2/x]$. The meaning of this is take the term e_2 and wherever x appears as a free variable in e_1, replace x with e_2.

We say an expression is in normal form if no beta reductions can be done on any sub-expression. Evaluating a lambda expression is the process of performing beta reductions.

4.1 What is the point of alpha reductions?

We need alpha reductions because the beta reduction rule does not always work correctly when we reduce an expression that defines the same variable name twice. For example, the term $K = \lambda xy.x = \lambda x.\lambda y.x$ is a function that takes an argument x and returns the constant function mapping all terms to x. Hence the term $\lambda y.Ky$ should take an argument y and return a constant function mapping all terms to y. However, if we reduce this naively we get

$$\begin{aligned}\lambda y.Ky &= \lambda y.(\lambda x.\lambda y.x)y \\ &= \lambda y.\lambda y.y\end{aligned}$$

This is alpha equivalent to $\lambda y.\lambda x.x$ which is a term that takes an argument and returns the identity function. This is not what we expected.

The problem comes from beta reducing the sub-expression $(\lambda x.\lambda y.x)y$ to $\lambda y.y$. In the first term the variable y is outside the scope of the y defined by the λy. However, in the second term it is inside the scope of the lambda. We say that y has been captured. To avoid this, we can alpha reduce in the midle of the process to get

$$\begin{aligned}\lambda y.Ky &= \lambda y.(\lambda x.\lambda y.x)y \\ &= \lambda y.(\lambda x.\lambda z.x)y \\ &= \lambda y.\lambda z.y\end{aligned}$$

We can avoid worrying about capture if we first perform alpha reductions to make all variables have different names before we start beta reducing.

5 Examples

Define

- **pair** $:= \lambda xyf.fxy$
- **first** $:= \lambda p.p\lambda xy.x$
- **second** $:= \lambda p.p\lambda xy.y$

Then for all expressions e_1, e_2 we have

$$\begin{aligned}
\textbf{first}(\textbf{pair }e_1e_2) &= \textbf{first}(\lambda f.fe_1e_2)\\
&= (\lambda f.fe_1e_2)\lambda xy.x\\
&= (\lambda xy.x)e_1e_2 = e_1\\
\textbf{second}(\textbf{pair }e_1e_2) &= \textbf{second}(\lambda f.fe_1e_2)\\
&= (\lambda f.fe_1e_2)\lambda xy.y\\
&= (\lambda xy.y)e_1e_2 = e_2
\end{aligned}$$

How did someone come up with **pair**, **first**, and **second** so that they would have these properties? The idea is that **pair** e_1e_2 needs to somehow store the terms e_1 and e_2 so that the terms **first** and **second** can extract them later.

The simplest way to store e_1 and e_2 is just to make **pair** $e_1e_2 = e_1e_2$. However, this doesn't give us any way extract the terms e_1 and e_2, i.e., there is no way to make a term **first** such that **first**$(e_1e_2) = e_1$. This is because there are terms e_1, e_2, e_3, e_4 such that $e_1e_2 = e_3e_4$ but $e_1 \neq e_3$ (exercise: construct some).

The next simplest thing to try is to wrap e_1 and e_2 in a λ. Trying **pair** $e_1e_2 = \lambda f.e_1e_2$ has the same problem as before, but if we try **pair** $e_1e_2 = \lambda f.fe_1e_2$ it works out. When we do this, we can construct **first** to use the f variable as a function that will decompose $fe_1e_2 = (fe_1)e_2$ into e_1. An f that works is $f = \lambda xy.x$.

Therefore, what we end up with is **first** $= \lambda p.p\lambda xy.x$. The variable p will be replaced by the term **pair** e_1e_2 which will then be applied to our $f = \lambda xy.x$. Constructing **second** is now trivial, we just change the f we want to use to $\lambda xy.y$ to get **second** $= \lambda p.p\lambda xy.y$.

You should do the same kind of thought process when you are trying to construct a multiplcation function for the last assignment. Walking through the process of why the addition function was defined the way it was may be halpful as well.

The terms **first** and **second** only really have "meaning" when applied to a term constructed by **pair**, but that doesn't mean we can't try to evaluate somewhat silly things like

$$\begin{aligned}
\mathbf{first}(\mathbf{second}(\lambda x.x)\mathbf{second}) &= \mathbf{first}((\lambda p.p\lambda xy.y)(\lambda x.x)\mathbf{second}) \\
&= \mathbf{first}((\lambda x.x)(\lambda xy.y)\mathbf{second}) \\
&= \mathbf{first}((\lambda xy.y)\mathbf{second}) \\
&= \mathbf{first}((\lambda xy.y)\lambda p.p\lambda xy.y) \\
&= \mathbf{first}\lambda y.y \\
&= (\lambda p.p\lambda xy.x)\lambda y.y \\
&= (\lambda y.y)\lambda xy.x \\
&= \lambda xy.x
\end{aligned}$$

Since there are no errors in the lambda calculus, we can evaluate any syntactically valid term whatsoever.