CPSC 201

A Brief and Informal Introduction to the Lambda Calculus

Paul Hudak
Spring 2008

There are three kinds of *expressions* (also called *terms*) in the pure lambda calculus:

x	(variables)
$\lambda x.\ e$	(abstractions)
$e_1\ e_2$	(applications)

where x, y, etc. are variables and e, e_1, etc. are (nested) expressions.

Intuitively, abstractions represent *functions*, and applications represent the application of a function to its *argument*. In this sense variable names are arbitrary, so that, for example, $\lambda x.\ x$ represents the "same" function as $\lambda y.\ y$.

Syntactic conventions: (1) $e_1\ e_2\ e_3$ is short-hand for $(e_1\ e_2)\ e_3$. (2) The binding of "λ" extends as far to the right as possible (overidden in the usual way by parentheses). So $\lambda x.\ x\ x$ is the same as $\lambda x.\ (x\ x)$, but is different from $(\lambda x.\ x)\ x$.

A particular instance of a variable is "free" in a lambda expression if it is not "bound" by a lambda. For example, x is free in the expressions x and $\lambda y.\ x$, but not in $\lambda x.\ x$. Because variables can be repeated, care must be taken to know which variable one is referring to. For example, in this expression, the underlined occurrences of x are free, the others are not:

$$\lambda y.\ \underline{x}\ (\lambda x.\ x\ x)\ \underline{x}$$

A term is *closed* if it has no free variables; otherwise it is *open*.

The only *reduction rule* of interest in this course is called *beta-reduction*, and is defined by:

$$(\lambda x.\ e_1)\ e_2 \quad \Rightarrow \quad e_1[e_2/x]$$

where the notation $e_1[e_2/x]$ denotes the result of substituting e_2 for all *free* occurrences of x in e_1. For example:

$$\begin{array}{lcl} (\lambda x.\ x)\ (\lambda y.\ y) & \Rightarrow & (\lambda y.\ y) \\ (\lambda x.\ x\ x)\ (\lambda y.\ y) & \Rightarrow & (\lambda y.\ y)\ (\lambda y.\ y) \\ (\lambda x.\ x\ (\lambda x.\ x))\ y & \Rightarrow & y\ (\lambda x.\ x) \end{array}$$

However, care must be taken to avoid "name capture" of bound variables. For example, this reduction would be "wrong":

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

because the outer y is presumably different from the inner y bound by the lambda. In cases like these, the bound variable is *renamed* to obtain the "correct" behavior:

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

A *reducible expression*, or *redex*, is any expression to which the above (beta-reduction) rule can be *immediately* applied. For example, $\lambda x.\ (\lambda y.\ y)\ z$ is not a redex, but its nested expression $(\lambda y.\ y)\ z$ is. The *location* of a redex relative to another is the point at which it begins in a left-to-right scan of the text.

Some expressions have more than one redex, and it turns out to be important to decide which redex to reduce first. There are many different reduction strategies, but the three of most interest in this course are:

1. Normal-order reduction: Choose the *left-most* redex first.
2. Applicative-order reduction: Choose the *right-most* redex first.
3. Haskell evaluation (more or less): Choose the left-most redex first (as in normal-order reduction), but only if it is *not* contained within the body of a lambda abstraction.

An expression with no redex is said to be in *normal form*.

Some expressions do not terminate using any of the above reduction strategies. For example:

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

Some terminate under normal-order reduction, but not under applicative-order. For example, this expression:

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

has two redexes. If we choose the right-most one (applicative-order), it will not terminate. But if we choose the left-most one, it will reduce in one step to the expression y.

It turns out that if an expression has a normal form, then normal-order reduction will find it. In other words, normal-order reduction terminates most often.

When constants or primitive values and functions are desired, special reduction rules can be added which create opportunities for other redexes. Examples include:

$$\begin{array}{lll}
(+\ 1\ 1) \Rightarrow 2 & & \text{(addition)} \\
(+\ 1\ 2) \Rightarrow 3 & & \\
\ldots & & \\
(*\ 1\ 1) \Rightarrow 1 & & \text{(multiplication)} \\
(*\ 1\ 2) \Rightarrow 2 & & \\
\ldots & & \\
(\mathit{head}\ (\mathit{cons}\ x\ xs)) \Rightarrow x & & \text{(lists)} \\
(\mathit{tail}\ (\mathit{cons}\ x\ xs)) \Rightarrow xs & & \\
\ldots & & \\
(\mathit{cond}\ \mathit{true}\ e_1\ e_2) \Rightarrow e_1 & & \text{(conditional)} \\
(\mathit{cond}\ \mathit{false}\ e_1\ e_2) \Rightarrow e_2 & & \\
\ldots & & \\
\text{etc.} & &
\end{array}$$

Alternatively, we can simulate these primitives using just the pure lambda calculus:

$$\begin{array}{lcl}
0 & \equiv & \lambda f.\lambda x.\ x \\
1 & \equiv & \lambda f.\lambda x.\ f\ x \\
\cdots & & \\
n & \equiv & \lambda f.\lambda x.\ f\ (\cdots (f\ x)\cdots) \\
 & & \\
succ & \equiv & \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x) \\
add & \equiv & \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x) \\
 & & \\
cons & \equiv & \lambda x.\lambda y.\lambda s.\ s\ x\ y \\
head & \equiv & \lambda l.\ l\ (\lambda x.\lambda y.\ x) \\
tail & \equiv & \lambda l.\ l\ (\lambda x.\lambda y.\ y) \\
 & & \\
cond & \equiv & \lambda p.\lambda c.\lambda a.\ p\ c\ a \\
true & \equiv & \lambda c.\lambda a.\ c \\
false & \equiv & \lambda c.\lambda a.\ a
\end{array}$$

In order to get the effect of recursion, one can define what is called the Y *combinator*, or *fixed point operator*, as:

$$Y \ \equiv \ \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Recursion is not something that is directly supported by the lambda calculus. But any recursive definition such as:

$$g \ \equiv \ \cdots g \cdots$$

can be rewritten as:

$$g \ \equiv \ Y\ (\lambda g. \cdots g \cdots)$$

which is non-recursive, and is thus a valid lambda expression. For example, here is a recursive and then a non-recursive definition of the factorial function:

$$fact \ \equiv \ \lambda n.\ cond\ (=\ n\ 0)\ 1\ (*\ n\ (fact\ (-\ n\ 1))))$$

$$fact \ \equiv \ Y(\lambda f.\ \lambda n.\ cond\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))$$