1 Lambda Calculus

1.1 Syntax

Recall the syntax of lambda terms:

$$\Lambda ::= x \mid (MN) \mid (\lambda x.M)$$

where

$x \in \mathcal{V}ar$ is a variable from a set $\mathcal{V}ar = \{w, x, y, z, w_1, x_1, y_1, \cdots\}$.

$M, N \in \Lambda$ are previously constructed lambda-terms.

The term (MN) is called an *application* and the term $(\lambda x.M)$ is called an *abstraction*. Some conventions for reading syntax are that application associates to the left, thus MN_1N_2 is the term $(MN_1)N_2$. Also, application has higher precedence than abstraction so $\lambda x.MN$ is the term $\lambda x.(MN)$ not $(\lambda x.M)N$.

The idea of the abstraction terms is that it models a function of one argument. So, if you define a function f, so that $f(x) = x$ then the lambda abstraction which is the same function as f is $\lambda x.x$. The symbol λ is a binding operator. In the term $\lambda x.M$, the variable x (the binding occurrence) is *bound* in the term M (called the *body*).Occurrences of the variable x in M are *bound occurrences*. Also, if you wrote $g(y) = y$, you would say that f and g are the same function. Similarly, $\lambda y.y$ is the same function as $\lambda x.x$. Renaming bound variables does not change the meaning of a function. An application MN means to apply the term M to the term N. If M happens to be an abstraction, we'll see below that we can do some computation. In a sense, if M is an abstraction, we know what to do.

The *scope* of the binding x in the term $(\lambda x.M)$ is the body M minus any parts of M that might rebind x. To see this consider the following term.

$$\begin{array}{cccccc} \lambda x & . & (\lambda x. & x) & x \\ \uparrow & & \uparrow & \uparrow & \uparrow \\ 1 & & 2 & 3 & 4 \end{array}$$

Occurrence (1) of x is a binding occurrence associated with the outermost λ. Occurrence (2) is another binding occurrence and binds occurrence (3). Occurrence (4) of x is bound by (1).

The free variables of a term are collected by the following function which is defined by recursion on the structure of the term:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \\ FV(\lambda x.M) & = & FV(M) - \{x\} \end{array}$$

If S and T are sets, then $S - T$ is the set difference defined as follows:

$$S - T \stackrel{\text{def}}{=} \{y \in S \mid y \notin T\}$$

1.2 Capture Avoiding Substitution

Computation in the lambda-calculus is performed by a kind of *reduction* – by computing away applications of functions (lambda abstractions) to their arguments. This operation is defined by capture avoiding substitution. We write $M[x := N]$ to denote the operation of replacing all free occurrences of x in M by the term N.

The definition is given to avoid *capture*. This is the situation where N contains a free variable (say y) and N replaces an occurrence of x in M which is in the scope of a binding occurrence of y. For example, if I naively apply the substitution $[x := y]$ to the abstraction $(\lambda y.xy)$ I get $\lambda y.yy$. This second abstraction has two bound occurrences of y while the first only had one – we have captured y. This changes the meaning of the function. To avoid capture we can rename, thus, before performing the substitution $[x := y]$ in the term $(\lambda y.xy)$, first can rename y to some new name, say z. We get the new term $(\lambda z.xz)$. Now, naive replacement of all x's by y does not capture y; we get $(\lambda z.yz)$ preserving the structure of bound variables in the new term. This idea is captured by the line $(3c)$ in the definition given below.

Note that $[x := N]$ is a substitution - a function that maps the variable x to the term N. We are using the conventional postfix notation (the substitution comes after the term it is being applied) to denote the application of the substitution to the term; thus, if σ is a substitution and M ts a term, we write $M\sigma$ (instead of $\sigma(M)$) to denote the term resulting from the application of the substitution σ to the term M. Note that substitution associates binds tightly and associates to the left.

$$\begin{array}{lll} M[x := N_1][y := N - 2] & \text{means} & (M[x := N_1])[y := N - 2] \\ MN[x := N_1] & \text{means} & M(N[x := N_1]) \\ \lambda y.N[x := N_1] & \text{means} & \lambda y.(N[x := N_1]) \end{array}$$

Capture avoiding substitution is defined by recursion on the structure of the lambda term. Since there are three ways to construct a lambda term

there are three cases in the definition; depending on whether the term being substituted into is a variable (lines $1a$ and $1b$), or it is an application (2) or it is a lambda term (lines $3a, 3b$ and $3c$). We write $x \doteq y$ to mean that - the meta-variables x and y denote syntactically identical (concrete) variables.

$$
\begin{array}{rcll}
y[x := N] & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll} N & \text{if } x \doteq y \\ y & \text{otherwise} \end{array} \right. & \begin{array}{l} (1a) \\ (1b) \end{array} \\
(M_1\, M_2)[x := N] & \stackrel{\text{def}}{=} & (M_1[x := N]\, M_2[x := N]) & (2) \\
(\lambda y.M)[x := N] & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lll} \lambda y.M & \text{if } x \doteq y & (3a) \\ \lambda y.(M[x := N]) & \text{if } x \neq y \wedge y \notin FV(N) & (3b) \\ \lambda z.(M[y := z])[x := N] & \text{if } x \neq y \wedge y \in FV(N) \wedge z \text{ new} & (3c) \end{array} \right. &
\end{array}
$$

Lines (1a) and (1b) describe what to do when a substitution of the form $[x := N]$ is applied to a variable y. Line (1a) says that if x and y are the same variable then go ahead and do the substitution, *i.e.* return N. If x and y are distinct variables, line (1b) says, do nothing, just return y unchanged. The variable case is the only case where a variable is replaced by a term, the rest of the definition just passes through the term structure, possibly doing some renaming if necessary to avoid variable capture further down in the term tree.

Line (2) describes what happens then the substitution $[x := N]$ is applied to an application of the form $(M_1\, M_2)$. In this case, just construct the application formed from the recursively constructed terms $M_1[x := N]$ and $M_2[x := N]$.

The lambda case is more complex and is described in lines (3a), (3b) and (3c). Line (3a) shows that if the bound variable in the term being substituted into is identical to the variable being substituted for, then do nothing. Recall, the idea is to replace all free occurrences of the variable x in $(\lambda x.M)$ by N. But there are no free occurrences of x in $(\lambda x.M)$ – so there is no point in looking any further, just return $(\lambda x.M)$. Lines (3b) and (3c) show the cases where the variable being substituted is different from the bound variable. Line (3b) shows how to proceed if there is no danger of capture and line (3c) shows what to do if there is the danger. In line (3b), if we know that $y \notin FV(N)$ then we know that replacing free occurrences of x by N will not capture any occurrences of y – since there are no free y's in N. This means, if there is an x in M and we replace it with N, we will not capture a y. To compute the answer in this case, construct the abstraction with bound variable y whose body is the result of recursively substituting N for x in M. Line (3c) is the most complex. We know that replacing a free occurrence of x in M will result in a captured instance of y.

To fix the problem, we choose a completely new variable name (say z) and then rename all the y's in M to be z and then (and only then), go ahead an recursively substitute $[x := N]$ into the renamed term. This renaming idea is based on the fact that changing the names of the arguments to a function and carefully renaming all references to the old name to the new name will preserve the behavior of the function. The function $\lambda x.x$ is the same function as $\lambda y.y$.

The condition z new means z is a new variable name, *i.e.* z is defined to be such that

$$z \text{ new } \stackrel{\text{def}}{=} z \in Var \wedge z \notin FV(M) \wedge z \notin FV(N) \wedge z \notin \{x, y\})$$

i.e. z does not occur free in either of the terms M or N and is not the variable x or the variable y.

Problem 1.1. Your assignment is to take the base code provided with this assignment and extend the definition of *subst* to implement the case for lambda.

1.3 Beta-reduction (reading for next class)

Computation proceeds by eliminating applications of abstraction (say, of the form $(\lambda x.M)$) to some other term (say N). The reduction is performed by capture avoiding substitution, replacing bound variable x in M (the body of the abstraction) by the term N. This process of reducing an application is called β-reduction (beta-reduction) and is defined as follows.

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

The term $M[x := N]$ is the result of the capture avoiding substitution. A term of the form $((\lambda x.M)N)$ is called a *redex* and the corresponding term $M[x := N]$ is called the *contractum*.

Here are some example step-by-step computations showing which line of the substitution algorithm is being applied at each step:

$$\begin{array}{rcl} (\lambda x.x)y & \rightarrow_\beta & \overbrace{x[x := y]}^{(1a)} \\ & = & y \end{array}$$

$$
\begin{aligned}
(\lambda x.xy)y &\to_\beta \overbrace{(xy)[x := y]}^{(2)} \\
&= \overbrace{x[x := y]}^{(1a)}\ y[x := y] \\
&= y\ \overbrace{y[x := y]}^{(1b)} \\
&= yy
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.\lambda y.xy)y &\to_\beta \overbrace{(\lambda y.xy)[x := y]}^{(3c)} \\
&= \lambda z.\overbrace{((xy)[y := z])}^{(2)}[x := y] \\
&= \lambda z.(\overbrace{x[y := z]}^{(1b)}\ y[y := z])[x := y]) \\
&= \lambda z.((x\ \overbrace{y[y := z]}^{(1a)})[x := y]) \\
&= \lambda z.\overbrace{(x\ z)[x := y]}^{(2)} \\
&= \lambda z.\overbrace{x[x := y]}^{(1a)}\ z[x := y] \\
&= \lambda z.y\ \overbrace{z[x := y]}^{(1b)} \\
&= \lambda z.y\ z
\end{aligned}
$$

For arbitrary terms M and N (*e.g.* where M is not necessarily an abstraction we write $M \to_\beta N$ if and only if there is a sub-term R of M such that R is a redex and if $R \to_\beta R'$, N is the term obtained by replacing the redex R in M by R'. This sounds more complicated that it is, all we are saying is that $M \to_\beta N$ if N is the result of β-reducing some subterm of M in place.

In the following examples we have underlined the redex being reduced and then show the computation performing the substitution.

$$
\begin{array}{rcl}
z\overbrace{((\lambda x.x)y)}^{redex} & \to_\beta & zy \\
\overbrace{((\lambda x.xy)y)}^{redex}\, z & \to_\beta & (yy)z \\
(z\overbrace{((\lambda x.\lambda y.xy)y)}^{redex})w & \to_\beta & (z(\lambda z.yz))w \\
\overbrace{(\lambda x.xx)(\lambda x.x)}^{redex} & \to_\beta & \overbrace{(\lambda x.x)(\lambda x.x)}^{redex} \\
 & \to_\beta & (\lambda x.x)
\end{array}
$$

Note that for some lambda terms, reduction will not terminate. Run the provided test cases. Be sure to submit you code (in ASCII) and yor test runs.

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