1 Notation

Lambda calculus is both a programming language *and* and a mathematical notation for writing functions. (When used as a programming language, it's fully parenthesized.) A programming language version of the identity function would be represented as: $(\lambda x x)$. The mathematical version of the identity function, represented using lambda calculus, would look like:

$$f = \lambda x \in T \ . \ x$$

where T is the domain of f.

2 Recursion

One shortcoming of the lambda calculus is its inability to easily express recursive functions. For example, consider a recursive function which computes the factorial of an integer:

$$Factorial(x) = \begin{cases} 1 & \text{if x = 0} \\ x * Factorial(x-1) & \text{otherwise} \end{cases}$$

We would like to define this function in the lambda calculus by saying:

$$\mathit{FACT} \triangleq (\lambda\ x\ (\mathit{IF}\ (\mathit{ZERO?}\ x)\ 1\ (*\ x\ (\mathit{FACT}\ (\ -\ \ x\ 1)))))$$

The problem with this statement is that it is an equation, not a definition. What we would like to find is the solution to this equation.

2.1 Defining a Recursive Function

We need to somehow remove the recursion within the definition. We will do this by defining a new variant of *FACT*, called *FACT*, which will be passed a function f such that $((f f)x)$ computes the factorial of x.

$$\mathit{FACT} \triangleq (\lambda\ f\ (\lambda\ x\ (\mathit{IF}\ (\mathit{ZERO?}\ x)\ 1\ (*\ x\ ((f\ \ f)\ (-\ x\ 1))))))$$

Now the actual factorial function we are seeking is *FACT* applied to itself.

$$\mathit{FACT} \triangleq (\mathit{FACT}'\ \ \mathit{FACT})$$

If we expand these *FACT* expressions, we find that in one step, we have three occurrences of *FACT*. In the next step, we will have four, and so on. So clearly, *FACT* diverges. However, the application of *FACT* to an integer does not, since the expansion of *FACT* ceases once x reaches a value of 0.

As an example, lets see what happens when we evaluate (*FACT* 3)

$$\begin{aligned}
(\mathit{FACT}\ 3) &= ((\mathit{FACT}\ \mathit{FACT})3) \\
&= (((\lambda\ f\ (\lambda\ \ (\mathit{IF}\ (\mathit{ZERO?}\ x)\ 1\ (*\ x\ ((f\ \ f)\ (-\ x\ 1))))))\ \mathit{FACT}\ )\ 3) \\
&= ((\lambda\ x\ (\mathit{IF}\ (\mathit{ZERO?}\ x)\ 1\ (*\ x\ ((\mathit{FACT}\ \mathit{FACT})\ (-\ x\ 1)))))\ 3) \\
&= (\mathit{IF}\ (\mathit{ZERO?}\ 3)\ 1\ (*\ 3\ ((\mathit{FACT}\ \mathit{FACT})\ (-\ 3\ 1)))) \\
&= (*\ 3\ ((\mathit{FACT}\ \mathit{FACT})\ 2)) \\
&= (*\ 3\ (*\ 2\ (*\ 1\ (*\ 1)))) \\
&= (6)
\end{aligned}$$

2.2 Recursion Removal Transformations

We can summarize what we just did to the *FACT* function to remove recursion as a three-step process.

1. Add an argument variable f to the recursive function.
2. Replace any internal references to the recursive function with an application of the argument variable to itself (i.e. $(f\ f)$).
3. Replace any external references to the recursive function with an application of our new function applied to itself.

2.3 Abstracting with the Fixed Point Operator

Recall our original recursive description of the factorial function:

$$\mathit{FACT} \triangleq (\ \lambda\ x\ (\mathit{IF}\ (\mathit{ZERO?}\ x)\ 1\ (*\ x\ (\mathit{FACT}\ (\ -\ \ x\ 1\ )))))$$

This description is an equation, whose solution is the factorial function
Note that we can simplify this equation by introducing a new function, *FACTEQN*:

$$\mathit{FACTEQN} \triangleq \lambda\ f\ (\ \lambda\ x\ (\ \mathit{IF}\ (\mathit{ZERO?}\ x)\ 1\ (*\ x\ (f\ (\ -\ \ x\ 1\ )))))$$

The resulting equation involving *FACT* can then be written as

$$\mathit{FACT} = (\mathit{FACTEQN}\ \mathit{FACT})$$

.

Suppose we had an operator Y that found the fixed point of functions. In other words, for any function f and any function argument x,
$((Y\ f)\ x)\ =\ (f\ ((Y\ f\ )\ x))$, and $(Y\ f)\ =\ f(Y\ f)$.
We could use such an operator to solve the *FACT* equation above:

$$\mathit{FACT} \triangleq (Y\ \ \mathit{FACTEQN})$$

.

It remains to be seen whether such a Y is actually computable. It is certainly possible that we cannot actually express the Y operator as a lambda expression. The reason for this is as follows. We can encode lambda calculus expressions as integers; thus the set of lambda expressions has the cardinality ω. However, it is easy to show that the number of functions from $Z \rightarrow Z$ is uncountable. In fact, only an infinitesimal fraction of all functions are computable.

However, it turns out that we can in fact define Y. First observe that $Y = (\lambda\ f\ (f\ (Y\ f)))$. (To see this, consider any new function *STAR*. $(\lambda\ f\ (f\ (Y\ f)))\ \mathit{STAR}\ =\ (\mathit{STAR}\ (Y\ \mathit{STAR}))\ =\ (Y\ \mathit{STAR})$.) Now we can apply the recursion removal technique we used above. Doing so yields
$Y' \triangleq (\lambda\ y\ (\lambda\ f\ (f\ (Y\ f))))$, and
$Y \triangleq (Y'\ Y')$.
The traditional form of Y, which requires call-by-name, is

$$Y \triangleq (\lambda\ f\ ((\ \lambda\ x\ (\ f\ (x\ x))\ (\ \lambda\ x\ (\ f\ (x\ x))))))$$

.

3 Substitution

Substitution turns out to be less straightforward that it might at first appear. Recall our rule for evaluating a substitution, the β-reduction, looks like

$$((\lambda\ x\ e_1)\ e_2)\ \rightarrow\ e_1\{e_2/x\}$$

First of all, we cannot simply replace every occurrence of variable x with e_2. For example,

$$(x\ (\lambda\ x\ x))\{a/x\}\ \neq\ (a\ (\lambda\ x\ a)).$$

Even if we avoid this problem by not substituting a for x in any lambda expression over x, we still have the problem of variable capture. For example,

$$(y\ (\lambda\ x\ (x\ y)))\{x\ /\ y\}\ \neq\ (x\ (\lambda\ x\ (x\ x)))$$

since in this case, the last occurrence of x has been incorrectly captured by the lambda expression.

To define substitution so that it does what we want, we will have to distinguish between *free* and *bound* identifiers. To avoid variable capture, we will only substitute an expression into a context where its variables are free.

To this end, we will define a function $FV[\![e]\!]$ which returns the set of all free variables in e. The special brackets $[\![\cdot]\!]$ are called *semantic brackets*. They wrap syntactic arguments, and so $FV[\![e]\!]$ operates on the abstract syntax tree for e, as opposed to the result of evaluating e. Sometimes we will use $[\![e]\!]$ when we mean $FV[\![e]\!]$.

We define $FV[\![\cdot]\!]$ inductively as follows:

- $FV[\![x]\!] = x$
- $FV[\![e_0\ e_1]\!] = FV[\![e_0]\!] \cup FV[\![e_1]\!]$
- $FV[\![\lambda\ x\ e]\!] = FV[\![e]\!] - \{x\}$

Similarly, we will define substitution inductively.

We will use $e\{e'/x\} \rightarrow e''$ when we want to say that e'' can be the result of substituting e' for x.

- $x\{e/x\} \rightarrow e$
- $y\{e/x\} \rightarrow y \qquad$ (if $y \neq x$)
- $(e_0\ e_1)\{e_2/x\} \rightarrow (e_0\{e_2/x\}\ e_1\{e_2/x\})$
- $(\lambda\ x\ e_0)\{e_1/x\} \rightarrow (\lambda\ x\ e_0)$

We need to be more careful when substituting e_1 for x inside a lambda expression term:

- $(\lambda\ y\ e_0)\{e_1/x\} \rightarrow (\lambda\ y\ e_0\{e_1/x\}) \qquad$ (if $y \neq x$ and $y \notin FV[\![e_1]\!]$)
- $(\lambda\ y\ e_0)\{e_1/x\} \rightarrow (\lambda\ y'\ e_0\{y'/y\}\{e_1/x\}) \qquad$ (if $y' \notin FV[\![e_1]\!]$, $y' \notin FV[\![e_0]\!]$, and $y' \neq x$)
- Note that this last rule makes sense because if $y' \in FV[\![e_1]\!]$, then y' is a free variable in e_1, and the renaming would artificially bind it to the argument of the lambda expression. We have the same problem if $y' \in FV[\![e_0]\!]$.

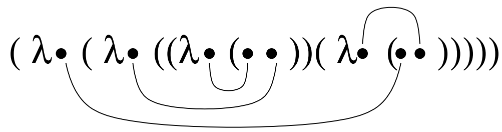

Figure 1: A Simple Stoy Diagram

4 Variable Binding

How can we tell which variable is denoted by an identifier in some expression? The scope of a variable is a description of the situations in which the identifier with the name of the variable is bound to that particular variable. In a language with lexical scope, the scope is purely *lexical*, i.e., consists of certain parts of the program text. In the lambda calculus, *lexical scope* applies, for example, to an expression like:

$$(\lambda\ p\ (\lambda\ q\ ((\lambda\ p\ (p\ q))\ (\lambda\ r\ (p\ r)))))$$

However, in the lambda calculus, as in most PLs, the identifiers have no inherent meaning. The expression above can also be represented by a *Stoy diagram* (where the dots are connected):

Intuitively, the meaning of a lambda expression does not depend on the name of the argument variable: $(\lambda\ x\ x) = (\lambda\ y\ y) = (\lambda\ \bullet\ \bullet)$. In the process of α-reduction, a function argument is renamed, yielding an equivalent expression:

$$(\lambda\ y\ e_0) \xrightarrow{\alpha} (\lambda\ y'\ e_0\{y'/y\}) \text{ where } y' \notin FV[\![e_0]\!]$$

Two lambda expressions are α-equivalent if they can be converted to each other using α-reductions, or, equivalently, if they have the same Stoy diagrams. Lambda expressions for *equivalence classes* defined by their Stoy diagrams.