Untyped Lambda Calculus

September 11, 2012

1 Introduction

The λ-calculus is a way of describing computations. It provides two facilities - a way to construct functions, and a way to apply (evaluate) them. The λ-calculus is simple, yet powerful enough that most pure functional features are just syntactic sugar defined over it. Thus it is one of the simplest languages which you can do functional programming in. [1]

2 Syntax

The syntax in BNF:

$$\begin{aligned} \text{variable } &= \text{`}v\text{'} \mid \text{variable `}{'}\text{'} \\ \lambda\text{-term } &= \text{variable } \mid \text{`('}\lambda\text{-term } \lambda\text{-term`)'} \mid \text{`(' `}\lambda\text{' variable } \cdot \lambda\text{-term`)'} \end{aligned}$$

e.g. The following are λ-terms

$$v, v', (v'v), (\lambda v(v'v))$$

Since the fully parenthesized syntax is tedious to read, we adopt a few conventions.

1. We will use letters in lower case to indicate variables and letters in upper case to indicate λ-terms in general.
2. $(\lambda x \cdot e_1 e_2)$ is $(\lambda x(e_1 e_2))$ - that is, the scope of a variable extends as far as possible, either to the first close parenthesis ’(’ symbol whose opening ‘)‘ occurs to the left of λ, or, to the end of the expression, whichever occurs first. [2]
3. Application associates to the left. Thus $e_1 e_2 e_3$ is $(e_1 e_2) e_3$.
4. $\lambda xyx \cdot e$ is $(\lambda x\ (\lambda y\ (\lambda z\ xyz)))$.

Example 2.0.1. The proper parenthesization of

$$(\lambda f \lambda x \cdot f(f(x)))$$

[1] The material in this is taken from various sources, but mostly it is an adaptation of [1] and [2].

is

$$(\lambda f \;\; (\lambda x f(fx)) \;\;) \;\; .$$

Evaluating a λ-term is through a process called β reduction. Intuitively, this corresponds to "executing" a program in a programming language. We will now introduce the theory behind β-reductions.

3 Reductions

The basic evaluation procedure in λ-calculus is that of substitution.

We are already familiar with this process in mathematics. For example, we can talk of the integral $\int_a^b f(x,y)dx$. If we substitute $y = 7$ in the integral, we get $\int_a^b f(x,7)dx$.

Further substitutions can be made, say $f = \sin$, $a = 0$ and $b = 1$. This yields the integral $\int_0^1 sin(x,7)dx$. This outlines a process of calculation through a series of substitutions.

Do all kinds of substitutions make sense? For example, substituting $y = 7$ in the expression $\int_a^b f(x,y)dx$ to obtain $\int_a^b f(x,7)dx$ made sense. However, substituting $x = 7$ in the resulting expression to obtain $\int_a^b f(7,7)d7$ obviously does not make sense: the distinction is that x is a bound variable in $\int_a^b f(x,y)dx$ where as y is free in it. Substitution can be done only for free occurrences of variables.

We now define the substitution process in λ-calculus. There are two ways in which to do this. Both essentially try to define which variables can be substituted, and which cannot be.

The first approach directly defines the free occurrences, that is, the ones which can be substituted. The second ensures that the sets of free variables and that of bound variables are disjoint, by a process of renaming conflicting variables.

The first approach is as follows. First, we need to define what are the free variables in a given λ-term.

Definition 3.0.2. The set of *free variables* in a given term M, denoted $\textsc{fv}(M)$ is recursively defined as follows.

$$\begin{aligned}\textsc{fv}[x] &= x\\ \textsc{fv}[(MN)] &= \textsc{fv}(M) \cup \textsc{fv}(N)\\ \textsc{fv}[(\lambda x M)] &= \textsc{fv}(M) \setminus \{x\}.\end{aligned}$$

All variables which are not free in an expression M is called *bound*. Alternatively, a variable is bound if it occurs under the scope of some λ in M.

Example 3.0.3. In the term $\lambda xy \cdot xyz$, the free variable is z and the bound variables are x and y.

In the term $\;(\lambda x \cdot \;\; (\lambda y \cdot yz) \;\; x) \;$, the variable z is free, and x and y are bound.

Definition 3.0.4. A *closed λ-term* or a *combinator* is an λ-term with no free variables.

We can now define the substitution process. This is the basic rule for "evaluating" λ-terms, similar to invoking functions in a functional language.

Definition 3.0.5. The process of *substituting* N for free occurrences of x in M, denoted $M[x := N]$, is recursively defined as follows. [2]

$$
\begin{aligned}
x[x := N] &\equiv N && (1)\\
y[x := N] &\equiv y && (2)\\
(M_1 M_2)[x := N] &\equiv (M_1[x := N] \quad M_2[x := N]) && (3)\\
(\lambda x \cdot M)[x := N] &\equiv (\lambda x \cdot M) && (4)\\
(\lambda y \cdot M)[x := N] &\equiv (\lambda y \cdot M[x := N]),\ \text{if } x \neq y \text{ and } y \notin FV(N). && (5)
\end{aligned}
$$

Definition 3.0.6. The β-reduction rule says that the result of the application $(\lambda x \cdot M)N$ is the substitution $M[x := N]$.

The second process of defining β-substitution is as follows. The rule called α-renaming, says that bound variables can be renamed to get equivalent expressions. For example,

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

The formal definition of α renaming is given inductively, similar to that of the process of identifying free variables. Once all bound variables are α-renamed so that the set of bound variables and free variables are disjoint, we can substitute directly without additional checks needed in rule (5). We omit this approach, preferring the first approach of directly identifying free occurrences.

Using the substitution rule, we can now explain λ-application. This is the way in which λ-terms are "reduced", typically to simpler forms.

Definition 3.0.7. The result of the *application* $(\lambda x M)N$ is the λ-term obtained by replacing all free occurrences of x in M by N. We say that $(\lambda x M)N$ *β-reduces* to $M[x := N]$, which we denote by

$$(\lambda x M)N \xrightarrow{\beta} M[x := N].$$

Example 3.0.8. [2] Let

$$
\begin{aligned}
M &= (\lambda f x \cdot f(f(x)))\\
N &= (\lambda y \cdot xy).
\end{aligned}
$$

β-reduce the λ-term MNN. That is, β-reduce $(MN)N$.

There is a third rule in the lambda calculus, called η-conversion. It is a rule that says that f and $(\lambda x \cdot fx)$ are equivalent. This is because for any λterm C, fC is the same as $(\lambda x \cdot fx)C$.

Thus there are just three rules for defining the semantics in lambda calculus - α conversion, β-substitution and η-conversion.

[2] The condition in rule (5) is to prevent free occurrences of y in N being bound by λy.
For example, $(\lambda y \cdot x)$ is a "constant function that returns constant x when given any argument y". Thus, what makes sense is $(\lambda y \cdot x)[x := y] = (\lambda z \cdot x)[x := y] = (\lambda z \cdot y)$, a term that evaluates to y on application with any argument. It should not reduce to $(\lambda y \cdot x)y = (\lambda y \cdot y)$, the identity function.

3.1 Reduction orders

Definition 3.1.1. A term is said to be in *normal form* if it cannot be β-reduced any further.

When a sequence of reductions encounters a normal form, the reduction terminates.

This introduces two natural questions: First, does every sequence of reductions terminate? If every sequence terminates, does every reduction sequence terminate in the same normal form?

The reduction sequence might not terminate.

For example, consider

$$(\lambda x \cdot xx)\ (\lambda x \cdot xx).$$

β-reduction of this results in the same expression.

Even worse, the size of the term may keep increasing.

Example 3.1.2. [2]

$$(\lambda f \cdot \ \ (\lambda x \cdot f(xx)) \quad (\lambda x \cdot f(xx))\ \ ) \xrightarrow{\beta} (\lambda f \cdot f(\ \ (\lambda x f(xx)) \quad (\lambda x f(xx))\ \ ))$$

So the answer to our first question is no. Now, is the reduction order unique? There might be multiple terms to be substituted in a given term. Depending on the order of substitution, some reductions might terminate, and others might not.

Example 3.1.3. Recall the earlier example. If $M = (\lambda x \cdot xx)$, then the β-reduction of MM does not terminate.

Suppose we have a projection function $P = (\lambda xy \cdot x)$, that projects out one of the arguments alone, then we know that the reduction PMM terminates in M.

You can use these two observations to construct two reduction sequences for the following expression, one which terminates, and another which does not.

$$(\lambda xyz \cdot xz(yz))PMM.$$

So the answer to even our second question is no.

3.2 Church-Rosser Theorem

However, we could ask for a simpler requirement. If two reduction sequences terminate, do they terminate in the same normal form? The following is a classical result in λ-calculus.

Theorem 3.2.1. *If $M \xrightarrow{*} M_1$ and $M \xrightarrow{*} N_1$, then there exists a P such that $M_1 \xrightarrow{*} P$ and $N_1 \xrightarrow{*} P$.*

That is, there is at most one normal form of any λ-term. Of course, some terms may not have normal forms at all, as discussed in the previous subsection.

We omit the proof of this theorem.

4 Abstract Data Types

We now see how to implement some abstract data types in λ-calculus.

4.1 Numbers

One encoding of the numbers we consider is the following.

$$\begin{aligned} \textsc{True} &\implies 0 \quad\quad \equiv \lambda xy.y \\ n = \lambda xy.M &\implies n+1 \equiv \lambda xy.x(M) \end{aligned}$$

Thus

$$\begin{aligned} 1 &\equiv \lambda xy \cdot x(y) \\ 2 &\equiv \lambda xy \cdot x(x(y)) \\ 3 &\equiv \lambda xy \cdot x(x(x(y))) \end{aligned}$$

and so on. It is easy to see that we are encoding the unary notation of natural numbers - the number of applications of x in the λ-term denoting n, is equal to n. This notation makes it possible for us to do arithmetic operations.

- Successor is defined to satisfy Successor(n) = n+1. This can be encoded as the function

  $$(\lambda xyz \cdot y(xyz))$$

  Verify that Successor(n) is a representation of $n+1$.

- Addition of two numbers m and n should return $m+n$.

  $$(\lambda xyzw \cdot xz \ \ (yzw) \ \ ).$$

- Multiplication of two numbers m and n should return their product.

  $$(\lambda xyz \cdot x(yz)).$$

4.2 Booleans

Define False to be $(\lambda xy \cdot y)$. We denote this by the reserved term **F**. Note that False has the same representation as 0. True is defined as $(\lambda xy \cdot x)$. We denote it consistently by **T**.

Now, it is possible to encode conditional blocks in the λ-calculus.

An if block, of the form [**if** E is true **then** M **else** N] can be encoded as

$$(\lambda xyz \cdot xyz).$$

4.3 Lists

We define the empty list to be $\text{NIL} = (\lambda xy \cdot y)$. Again, this is the same representation as that of 0 and of False.

A list is essentially a pair of values. The first is a "head" element of the list. The second element is the "tail" of the list, which is the list without the first element.

Cons, the list constructor, is defined as follows.

$$\text{CONS} = (\lambda xyz \cdot zxy).$$

For example,

$$\text{CONS}\ a\ \text{NIL} = (\lambda z \cdot za\text{NIL})$$
$$\text{CONS}\ b\ (\text{CONS}\ a\ \text{NIL}) = (\lambda z \cdot zb(\text{CONS}\ a\ \text{NIL})).$$

The reason CONS has been defined in this manner, is that you can apply the following two functions to a list to get the head, and the tail of the list, respectively. This encoding uses what are called "Church pairs".

Every list has a head, which is the first element of the list, and a tail, which is the rest of the list.

$$\text{HEAD} = (\lambda f \cdot f(\lambda xy.x)).$$
$$\text{TAIL} = (\lambda f \cdot f(\lambda xy.y)).$$

For example,

$$\begin{aligned} \text{HEAD}\ (\text{CONS}\ a\ \text{NIL}\ ) &= \text{HEAD}\ (\lambda z \cdot za\ \text{NIL}\ ) \\ &= (\lambda f \cdot f(\lambda xy \cdot x))(\lambda z \cdot za\ \text{NIL}\ ) \\ &= (\lambda z \cdot za\ \text{NIL}\ )(\lambda xy \cdot x) \\ &= (\lambda xy \cdot x)a\ \text{NIL} \\ &= a. \end{aligned}$$

Verify that $\text{TAIL}\ (\text{CONS}\ a\ \text{NIL}\ ) = \ \text{NIL}\ $.

5 Recursion and the Y combinator

The λ-calculus does not have the provision of a function referring to itself. We will construct a function which will enable us to do recursion in the λcalculus.

Any recursive function f can be written in the form $f = gf$. That is, f is a fixed point of the function g.

There is a function Y which finds the fixed point of any function. In particular, for the function g, $Yg = gYg$. Thus, $f = Yg$ is a definition of f.

The Y combinator is $(\lambda k \cdot (\lambda x \cdot k(xx))(\lambda x \cdot k(xx)))$.

We can verify that $Yg = gYg$, thus proving that Yg is a fixed point of g.

$$
\begin{aligned}
Yg &= (\lambda k \cdot (\lambda x \cdot k(xx))(\lambda x \cdot k(xx)))g && (6)\\
&= (\lambda x \cdot g(xx))(\lambda x \cdot g(xx)) && (7)\\
&= g((\lambda x \cdot g(xx))(\lambda x \cdot g(xx))) && (8)\\
&= g(Yg) \qquad \text{[from (7), (8)]} && (9)
\end{aligned}
$$

References

[1] H. Barendregt and E. Barendsen. Introduction to the lambda calculus.

[2] Amitabha Sanyal. Notes on lambda calculus.

6 Appendix

We will attempt *deriving* the expressions for the numerical functions.

§1. The successor function maps a number

$$n \quad = \quad \lambda xy \cdot \overbrace{x(x(x(\ldots x(y)\ldots)))}^{M},$$

where x occurs n times in the inner expression, and outputs

$$n+1 \quad = \quad \lambda xy \cdot x(M).$$

Intuition (by Amey Karkare) The expression

$$\overbrace{x(x(x(\ldots x}^{n}(y)\ldots)))$$

can be interpreted as - y is a board, on which I make x number of markings. Thus, the successor function has to do the following: take a board on which n markings have been made, that is, nxy, and now make one more mark x. This means we need the following as arguments: the board with n markings made (n), and the additional mark (x). This yields the expression

$$(\lambda n \text{ mark board} \cdot \text{mark } (n \text{ mark board})).$$

Verification.

$$
\begin{aligned}
\text{Successor } 0 &= [\lambda mst \cdot s(mst)] \qquad (\lambda xy \cdot y)\\
&\xrightarrow{\beta} [\lambda st \cdot s``((\lambda xy \cdot y))st)]\\
&\xrightarrow{\beta} [\lambda st \cdot s(t)]\\
&= 1.
\end{aligned}
$$

In general,

$$\begin{aligned}
\text{Successor } n &= [\lambda mst \cdot s(mst)] \qquad (\lambda xy \cdot x^{(n)}(y)) \\
&\xrightarrow{\beta} \left[\lambda st \cdot s \text{``} \left((\lambda xy \cdot x^{(n)}(y))st\right)\right] \\
&\xrightarrow{\beta} \left[\lambda st \cdot s \left(s^{(n)}(t)\right)\right] \\
&= \lambda st \cdot s^{(n+1)}(t) \\
&= n+1.
\end{aligned}$$

§2. Multiplication of m and n can be similarly thought of in the following terms: first, we form a composite symbol N which consists of writing n symbols on a blank board. This is

$$\overbrace{nxy}^{N}.$$

Then we form the expression which expresses, write N repeatedly m times on a blank board. This is

$$mNy$$

.

Thus, we have the expression for multiplication,

$$(\lambda mnxy \cdot m\ (nxy)\ y).$$

§3. For addition, we write n symbols on a blank board, and then write m more of the same symbol. This yields the expression

$$(\lambda mnxy \cdot m\ x\ (nxy)).$$