Untyped Lambda Calculus

August 29, 2013

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:

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

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 $(e_1 e_2)e_3$.

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

4. $\lambda xyx \cdot e$ is $(\lambda x\ (\lambda y\ (\lambda z\ xyz)))$.

Example 2.0.1. The proper parenthesization of

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

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 occurences, 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 $\text{FV}(M)$ is recursively defined as follows.

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

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.

Definition 3.0.3. A *closed λ-term* or a *combinator* is an λ-term with no free variables.

We can now define the substitution process.

Definition 3.0.4. The process of *substituting* N for free occurrences of x in M, denoted $M[x := N]$, is recursively defined as follows.

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N] \quad M_2[x := N]) \\ (\lambda y \cdot M)[x := N] &\equiv \lambda y \cdot M[x := N] \\ (\lambda x \cdot M)[x := N &\equiv (\lambda x \cdot M). \end{aligned}$$

Example 3.0.5. 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.

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. 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 3.0.6. 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 3.0.7. [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.

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

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

This reduction sequence 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.

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

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.

One way to think about this encoding is that it is the unary representation of natural numbers. Consider the expression $0xy = y$. This can be thought of as a "blank board". The expression nxy can be thought of as one obtained by making n number of x marks on the blank board.

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

  $$(\lambda nxy \cdot x(nxy))$$

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

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

  $$(\lambda mnxy \cdot mx \ \ (nxy) \ \ ).$$

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

  $$(\lambda mnx \cdot m(nx)).$$

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

$$(\lambda emn \cdot emn).$$

Verify that $(\lambda emn \cdot emn)\ \mathbf{T} = m$ and $(\lambda emn \cdot emn)\ \mathbf{F} = n$.

4.3 Lists

We define the empty list to be $\textsc{nil} = (\lambda xy \cdot y)$. 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.

$$\textsc{Cons} = (\lambda v\ell z \cdot zv\ell).$$

For example,

$$\textsc{Cons}\ a\ \textsc{nil} = (\lambda z \cdot za\textsc{nil})$$
$$\textsc{Cons}\ b\ (\textsc{Cons}\ a\ \textsc{nil}) = (\lambda z \cdot zb(\textsc{Cons}\ a\ \textsc{nil})).$$

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.

$$\textsc{Head} = (\lambda f \cdot f(\lambda xy.x)).$$

$$\textsc{Tail} = (\lambda f \cdot f(\lambda xy.y)).$$

For example,

$$\begin{aligned}
\textsc{Head}\ (\textsc{Cons}\ a\ \textsc{nil}\ ) &= \textsc{Head}\ (\lambda z \cdot za\ \textsc{nil}\ ) \\
&= (\lambda f \cdot f(\lambda xy \cdot x))(\lambda z \cdot za\ \textsc{nil}\ ) \\
&= (\lambda z \cdot za\ \textsc{nil}\ )(\lambda xy \cdot x) \\
&= (\lambda xy \cdot x)a\ \textsc{nil} \\
&= a.
\end{aligned}$$

Verify that $\textsc{Tail}\ (\textsc{Cons}\ a\ \textsc{nil}\ ) =\ \textsc{nil}$.

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.

$$\begin{aligned}
Yg &= (\lambda k \cdot (\lambda x \cdot k(xx))(\lambda x \cdot k(xx)))g && (1)\\
&= (\lambda x \cdot g(xx))(\lambda x \cdot g(xx)) && (2)\\
&= g((\lambda x \cdot g(xx))(\lambda x \cdot g(xx))) && (3)\\
&= g(Yg) \qquad [\text{from (2), (3)}] && (4)
\end{aligned}$$

References

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

[2] Amitabha Sanyal. Notes on lambda calculus.

Addendum

Recursion using the Y-combinator

How do we do recursion in the λ-calculus? For a moment, let us consider how recursion and self-referential data structures work in a programming language. Suppose you have

```
factorial(n) :
  if n==0 then 1 else n*factorial(n-1)
```

this needs the facility for a function to be defined in terms of itself. How do we accomplish this? One way to easily handle this in a programming language is using symbol tables and multi-pass processing. In the first pass, for example, `factorial` is processed as a function. So an entry is made in the symbol table corresponding to the name `factorial`. This function calls another function (in this case, itself, but this is not significant) - in the first pass, we can insert a mark to say that the call is to is some entry in the symbol table, which we will resolve later.

After the first pass, all names that the program defines will be known, so the symbol table will be complete.

In another pass, we will go through all the marks we made earlier, and fill in the corresponding entries from the symbol table. Thus, names, symbol table and multi-pass processing is one way to enable recursion in a language.

How will a λ-term refer to itself? It has no "name", hence it cannot refer to itself by name.

For this, we view a recursive λ-term as a fixed point of a sequence of definitions, and use the Y combinator. To see this, it is probably best to consider the following example, which is "almost" the definition of the factorial function.

$$f = (\lambda g n \cdot \text{IF } 0 \text{ THEN } 1 \text{ ELSE } (\text{MULTIPLY } n \ (g \ (\text{PREDECESSOR } n)))).$$

Now, consider Yf. This is a fixed-point of f - that is, $f(Yf) = Yf$. This means that

$$f(Yf) = (\lambda n \cdot \text{IF } 0 \text{ THEN } 1 \text{ ELSE } (\text{MULTIPLY } n \ ((Yf) \ (\text{PREDECESSOR } n)))) = Yf.$$

where the first equality follows by the rule for application, and the second equality follows since Yf is a fixed point of f. In particular,

$$Yf = (\lambda n \cdot \text{IF } 0 \text{ THEN } 1 \text{ ELSE } (\text{MULTIPLY } n \ ((Yf) \ (\text{PREDECESSOR } n)))).$$

Thus, Yf is the "factorial" function!

This trick can be applied to recursively define a λ-term in terms of itself - consider a λ-term f which takes an argument g and calls g wherever the final recursive call occurs. Now, Yf is the recursive definition we originally wanted.

How would you implement mutually recursive λ-terms? Explore this.