Notes on λ-calculus

Author: Amitabha Sanyal

Last updated: 24th August 2001

Abstract

In the attempt to formalise the intuitive notion of computability, Turing came up with the *Turing machine*. Later developments of computer hardware was closely based on this model. In particular, von Neumann's idea of a stored program computer came from the Universal Turing machine.

Though convenient to realise in hardware, this model is too operational in nature. What is required is a model that captures computations instead of computers. Church's model, the *lambda-calculus*, does this by providing two essential things: a way of constructing functions and a way of applying them. The *terms* in lambda-calculus allow us to create functions and the *β-reduction* rule models application.

Our interest in lambda calculus stems from the fact that functional programming closely follows this model. In fact functional programs are sugared lambda terms and interpretation of a functional program on actual hardware is an implementation of sequences of beta reductions.

1 Syntax

A term in lambda-calculus is defined by the following abstract syntax :

$e ::x \mid \lambda x.e \mid e_1 e_2 \mid (e)$

Each e generated by the above rule is called a *lambda term* and is either a variable, an abstraction ($\lambda x.e$) or an application ($e_1 e_2$). An abstraction is a term that stands for a function that takes an argument, binds it to the variable x, evaluates the expression e with the above binding for x and returns the value. To avoid ambiguities we parenthesize the expression. However too many brackets hinder readability and so we introduce the following scope rules :

- $\lambda x.e_1\ e_2\ e_3$ is an abbreviation for $\lambda x.(e_1\ e_2\ e_3)$, i.e. the scope of λx is as far to the right as possible until it is terminated (i) by a ')' whose matching '(' occurs to the left of the λ, or (ii) by the end of the term.
- Application associates to the left, i.e. $e_1\ e_2\ e_3$ is to be read as $(e_1\ e_2)\ e_3$, and not $e_1\ (e_2\ e_3)$.
- $\lambda xyz.e$ is an abbreviation for $\lambda x \lambda y \lambda z.e$, which in turn is actually $\lambda x.(\lambda y.(\lambda z.e))$

Though the language looks very small, we shall see later that most Haskell expressions can be translated into such a language.

2 β-reduction

The basic reduction rule in λ-calculus is the rule that tells us how to compute applications. The rule is quite intuitive: it says that if an abstraction $\lambda x.e_1$ is applied to a term e_2, then the result of the application is the body of the abstraction e_1 with all free occurrences of the formal parameter x replaced with e_2.

$$(\lambda x.e_1)e_2 \xrightarrow{\beta} e_1[e_2/x]$$

Read $\xrightarrow{\beta}$ above as *β-reduces to*. An expression like $(\lambda x.e_1)e_2$, which is a possible candidate for β-reduction, is called a *β-redex*(redex in short). We shall extend β-reduction to arbitrary lambda-terms by saying that $e_1 \xrightarrow{\beta} e_2$ if some redex in e_1 is reduced to e_2.

Here is an example of a series of $\beta-$reductions. The redex which is reduced at each step is boxed.

$$\begin{aligned}
&\boxed{(\lambda x\lambda y\lambda z.xz(yz))(\lambda x\lambda y.x)}\,(\lambda y.y)\\
\xrightarrow{\beta}\ &(\lambda y\lambda z.\,\boxed{(\lambda x\lambda y.x)z}(yz))(\lambda y.y)\\
\xrightarrow{\beta}\ &(\lambda y\lambda z.\boxed{(\lambda y.z)(yz))}\,)(\lambda y.y)\\
\xrightarrow{\beta}\ &\boxed{(\lambda y\lambda z.z)(\lambda y.y)}\\
\xrightarrow{\beta}\ &\lambda z.z
\end{aligned}$$

There is a caution which should be exercised while applying the β-reduction rule. An occurrence of a variable is said to be *bound* in an lambda term, if it is in the scope of an abstraction of a variable with the same name. Otherwise it is *free*. Thus in the lambda term $\lambda x\lambda y.\underline{x}\overline{z}$, the underlined x is bound and the overlined z is free. Define $A = \lambda x\lambda y.x$. Now reduce the term $A(\lambda x.y)$ and observe that the y that was initially free in this term is now bound in the reduced term. This is the problem of *variable-capture* and to see that allowing this is wrong define $B = \lambda x\lambda u.x$. This is the same function as A except for renaming of the variable y. However observe that the reductions $A(\lambda x.y)$ and $B(\lambda x.y)$ give rise to different results.
To avoid the variable-capture problem, we define the the notion of *α-renaming*(denoted as $\xrightarrow{\alpha}$ as follows :

$$\lambda x.t \xrightarrow{\alpha} \lambda y.t[y/x]$$

Variable capture in the example above can now be avoided by using α-renaming A to B.

Exercise: Parenthesize the following expressions correctly and reduce them as far as possible.

1. $(\lambda f \lambda x.f(f(x)))(\lambda y.xy)(\lambda y.xy)$
2. $(\lambda f \lambda g \lambda x.f(g(x)))(\lambda f \lambda g \lambda x.g(f(x)))(\lambda x \lambda y.x)$

β and α conversion rules represent different ways in which two lambda-terms can be considered equivalent. There is a third conversion rule called $\eta - reduction$ (denoted $\xrightarrow{\eta}$) which captures the idea that two functions are equal if they give the same result on application to any argument. Thus $\lambda x.fx$ and f, can be seen to be equivalent by applying both of them to a third term e. Formally,

$$\lambda x.fx \xrightarrow{\eta} f.$$

As in β-reduction, we can generalize α-renaming and η-reduction to arbitray terms. We shall use $\longrightarrow$ to mean $\xrightarrow{\beta}$, $\xrightarrow{\alpha}$ or $\xrightarrow{\eta}$. Unless otherwise stated, from now on by 'reduction' we shall mean $\longrightarrow$. Finally, a sequence of zero or more reductions will be represented as $\rightarrow\!\!\rightarrow$.

2.1 Orders of Reduction

We say that a term is in $normal\ form$ if it cannot be reduced any further, i.e. it does not contain any redexes. A reduction sequence terminates when it produces a term in normal form. If we consider a lambda term to be a program and β-reduction as execution, then the normal form of the term can be considered as the 'answer' of the program. We have seen that some reduction sequences, like the one shown above, terminate. However some of them may not. Show that the reduction sequence of the lambda term below never terminates.

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

In fact, the size of the term may keep on growing with every β-reduction. Verify that this is the case for

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

In general, a given term may have more than one β-redexes. Whether a particular reduction sequence terminates or not may depend on the order in which we choose these redexes for reduction. For the term given below, enumerate at least two reduction sequences, one of which terminates and one does not.
$(\lambda x \lambda y \lambda z.xz(yz))(\lambda x \lambda y.x)((\lambda x.xx)(\lambda x.xx))$

2.2 Church-Rosser Theorem

However, if two different reduction sequences terminate then they always terminate in the same term. The general statement of which the above is a corollary is called the *Church-*

Rosser Theorem. Formally,

Church Rosser Theorem: If $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$, then there is a lambda term e' such that $e_1 \twoheadrightarrow e'$ and $e_2 \twoheadrightarrow e'$.

In terms of a diagram, we have:

$$\begin{tikzcd}
 & e \arrow[dl, two heads] \arrow[dr, two heads] & \\
e_1 \arrow[dr, two heads] & & e_2 \arrow[dl, two heads] \\
 & \exists e' &
\end{tikzcd}$$

Exercise: Show that if two different reduction sequences terminate, then they always terminate in the same term.

Verify this result for the following terms.

1. $(\lambda x \lambda y.xy)((\lambda x \lambda y.x(xy))f)$
2. $(\lambda x \lambda yz.xy(yz))(\lambda x \lambda y.z)(\lambda y.y)$

A reduction sequence that always reduces the *leftmost outermost* redex is called a *normal order* reduction sequence.
A theorem called the *normalisation theorem* asserts that if a normal form exists at all,the normal order reduction sequence will find it.

Exercise: Verify this theorem for the examples given above.

3 Representation of 'data'

We now show how some commonly used data structures and operations can be defined using λ-calculus.

3.1 Numbers

One way of representing numbers is through *Church numerals*, in which the representation of a number n, denoted as $\overline{n}$, is $\lambda x \lambda y.x(x(\ldots n\ times \ldots x(y)\ldots))$. Thus $\overline{0} = \lambda x \lambda y.y$ and $\overline{2} = \lambda x \lambda y.x(xy)$.
Why does this representation make sense? It is because, based on this representation, we can now define most of the arithmetic functions on numbers. For instance:

1. We can define the successor function as. $succ = \lambda x \lambda y \lambda z.y(xyz)$. Verify that $succ\, \overline{n} = \overline{n+1}$
2. Similarly define add as $\lambda x \lambda y \lambda z \lambda w.x\ z\ (y\ z\ w)$ and verify once again that $add\, \overline{n}\, \overline{m} = \overline{n+m}$
3. Finally define $mult$ as $mult = \lambda x \lambda y \lambda z.x(yz)$ and show that $mult\ \overline{n}\ \overline{m} = \overline{n*m}$

3.2 Booleans

Define $\overline{true} = \lambda x \lambda y.x$ and $\overline{false} = \lambda x \lambda y.y$. With these representations it is possible to write the conditional function if such that $if\ c\ e_1\ e_2$ reduces to e_1 if c reduces to $\overline{true}$, and e_2 if c reduces to $\overline{false}$.

Exercise:

1. Write a λ-term for if.
2. Write a function $iszero$ which yields $\overline{true}$ when applied to $\overline{0}$ and $\overline{false}$ when applied to any other numeral.
3. What simple arithmetic function does the following lambda term represent?
   $\lambda n.n(\lambda p \lambda z.z(succ(p\ \overline{true}))(p\ \overline{true}))\ \ (\lambda z.z\ \overline{0}\ \overline{0})\ \overline{false}$

3.3 Lists

Define $Nil = \lambda x.\lambda y.y$, and $Cons = \lambda x \lambda y \lambda z.zxy$. Based on this, write the representation of the list $[a, b, c]$, which is actually $(Cons\ a\ (Cons\ b\ (Cons\ c\ Nil)))$.
Now we can write the functions $isNil = \lambda a.a(\lambda zpw.\overline{false})\overline{true}$. Show that $isNil\ Nil = \overline{true}$ and $isNil(Cons\ a\ (Cons\ b\ (Cons\ c\ Nil))) = \overline{false}$
Define $head = \lambda x.(x\ (\lambda x \lambda y.x))$. Show that $head\ (Cons\ a\ (Cons\ b\ Nil))) = a$. Similarly define $tail$ and show that $tail(Cons\ a\ (Cons\ b\ Nil))) = (Cons\ b\ Nil))$

Exercise: We want to define trees of the form:

```
data Tree = Node Int Tree Tree | Niltree
```

in lambda calculus, given the lambda representation for integers. To do this, give lambda terms for constructors $node$ and $niltree$ and the destructors $getvalue$, $getltree$ and $getrtree$. Additionally, define a function $isniltree$ which will return $\overline{true}$ if its argument is $niltree$, and $\overline{false}$ otherwise.

4 Recursion

The machinery seen so far does not allow us to define a value recursively, i.e. through an equation like $p = f\ p$. In this equation, it is p which is being defined. Any value which satisfies the equation when plugged in place of p, is called a $fixpoint$ of f. Thus any fixpoint of f is a solution of the equation $p = f\ p$.

Assume for the moment that there exists a function, historically called Y, which when applied to f, gives the fixpoint of f. Then clearly $Y\ f$ is a solution of $p = f\ p$, i.e. $(Y\ f) = f\ (Y\ f)$. This gives us a method to convert a recursively defined value into a non-recursive one. The recursive equation $p = f\ p$ defines the same value as the non-recursive lambda term $Y\ f$. But all this is true provided there exists a Y satisfying the crucial property $(Y\ f) = f\ (Y\ f)$.
We show first that there is such a Y. Actually there are many such Ys of which the most well-known is:

$$Y = \lambda f(\lambda x.f(xx))(\lambda x.f(xx))$$

Now,
$Yf = (\lambda f(\lambda x.f(xx))(\lambda x.f(xx)))f$
$\quad = (\lambda x.f(xx))(\lambda x.f(xx))$
$\quad = f((\lambda x.f(xx))(\lambda x.f(xx)))$
$\quad = f(Yf)$

Exercise: This is a great one. Define:
$T = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$
$Y_{funny} = TTTTTTTTTTTTTTTTTTTTTTTTTT$
Now show that Y_{funny} is a fixed point function, i.e. $(Y_{funny}\ f) = f\ (Y_{funny}\ f)$.

To see this theory at work, consider the recursively defined function
$\quad fact\ n = if(iszero\ n)\ \overline{1}\ (mult\ n\ (fact(pred\ n)))$
where if and $mult$ are as defined before, and $pred$ satisfies the equation $pred\ \overline{n} = \overline{n-1}$. One should look upon this equation as one defining the value $fact$.
Now let us first write this equation in the form $fact = g\ fact$ for some g. First bring n to the rhs.
$\quad fact = \lambda n.if(iszero\ n)\ \overline{1}\ (mult\ n\ (fact(pred\ n)))$
$\quad fact = (\lambda f \lambda n.if(iszero\ n)\ \overline{1}\ (mult\ n\ (f\ (pred\ n)))\ fact)$
$\quad fact = g\ fact$, where
$\quad g = \lambda f \lambda n.if(iszero\ n)\ \overline{1}\ (mult\ n\ (f(pred\ n)))$
Now according to the preceding discussion, the non-recursive lambda term $Y\ g$ should express the same factorial function which was being specified by the recursive equation above. Let us verify this by showing that $(Y\ g)\ \overline{2} = \overline{2}$.
$\quad\ \ (Y(\lambda f \lambda n.if\ (iszero\ \overline{2})\ \overline{1}\ (mult\ \overline{2}\ (f(pred\ n)))))\ \overline{2}$
$\quad = ((\lambda f \lambda n.if\ (iszero\ \overline{2})\ \overline{1}\ (mult\ \overline{2}\ (f\ (pred\ n))))(Y\ f))\ \overline{2}$
$\quad = (\lambda n.if\ (iszero\ n)\ \overline{1}\ (mult\ \overline{2}\ ((Y\ f)\ f\ (pred\ n))))\ \overline{2}$
$\quad = (if\ (iszero\ \overline{2})\ \overline{1}\ (mult\ \overline{2}\ ((Y\ f)\ (pred\ \overline{2}))))\ \overline{2}$
$\quad = (mult\ \overline{2}\ ((Y\ f)\ (pred\ \overline{2})))$
$\quad = (mult\ \overline{2}\ ((Y\ f)\ \overline{1}))$
$\quad \vdots$
$\quad = mult\ \overline{2}\ (if(iszero\ \overline{1})\ \overline{1}\ (mult\ \overline{1}((Y\ f)(pred\ \overline{1}))))$

$$\begin{aligned}
&\vdots \\
&= mult\ \overline{2}\ (mult\ \overline{1}\ (if (iszero\ \overline{0})\ \overline{1}\ (mult\ \overline{1}\ ((Y f)(pred\ \overline{0}))))) \\
&= mult\ \overline{2}\ (mult\ \overline{1}\ \overline{1}) \\
&= \overline{2}
\end{aligned}$$

Exercise

1. Fill in the missing steps in the reduction given above. Using the non-recursive specification of $fact$ in a similar way, reduce $fact\ 4$ to 24.
2. Using $iszero$, $pred$, $mult$ and Y, define exp satisfying $exp\ \overline{m}\ \overline{n} = \overline{m^n}$.