CS 329 Notes
on
Untyped Lambda Calculus

revised Sept 2007, oct 2008

Rushikesh K. Joshi
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai - 400 076, India.
Email: rkj@cse.iitb.ernet.in

1 Introduction

Lambda calculus is a calculus with its core features of function definition (abstraction) and function application by variable substitution. The untyped lambda calculus was invented by Church about more than 75 years ago. Anything that is computable can be expressed in lambda calculus and it is equivalent to Turing Machine. The basic calculus is untyped, and has a very meager syntax. It's interesting to see how one can explain the computational structures that you see in programming languages in terms of expressions in lambda calculus.

2 Basic Syntax

Every abstraction in lambda calculus takes a single argument. (Note: But functions with multiple arguments can be transformed to an expression involving functions with single argument. This transformation process is called *currying*.) Some example functions definitions (abstractions) are given below. In the terms below, x is used as argument, and the body computes a value with an expression in terms of the argument.

2.1 Abstractions

- $\lambda x.x + x$
  function body doubles the argument and the computed value is returned.
- $\lambda x.x$
  function returns the argument itself. It's identity function.
- $\lambda x.xx$
  the function takes an argument and applies that to the argument itself.

2.2 Applications

The abstractions can be applied to values as in examples below. Note that an application has 2 terms, the first term is applied to the second term by substitution of the second term as argument in the first term which is an abstraction.

- $(\lambda x.x)\ \ p$
  produces p as its result by substituting x in the abstraction on the left by argument value p.
- $(\lambda x.xx)\ \ y$
  produces result as $y\ y$ by substituting y for x. The term $y\ y$ is not reducible.
- $(\lambda x.xx)\ \ (\lambda x.x)\ \ p$
  reduces to p

You will see that we have used the operator '+' inside the body of lambda abstractions. Is '+' keyword and its associated meaning included in the language of lambda expressions? Well, the answer is no. We will have to show that '+' can be somehow implemented in terms of lambda expressions. What is the syntax for lambda expressions? It's quite minimal. The core constructs are just three: (1) variable (2) abstraction (3) application. This is explained below.

The syntax of the calculus can now be given. A lambda term is either a variable (e.g. x, y, z,..), or an abstraction, or an application. i.e.,
$variable\ = x\ |y\ |z\ |...$
(1) $term\ \ =\ \ variable$
(Just a variable)
Examples: u, v, x, y
(2) $term\ \ =\ \lambda variable.term$
(Abstraction: Function definition with 1 argument and body of the function)
Examples: $\lambda x.x\ x$ in which $x\ x$ is the body of the function. The body needs rule 3 below to parse correctly. Another example is $\lambda x.x$ which is identity function.
(3) $term\ \ =\ term\ \ term$ (Application of one function to an argument)
Examples: $x\ y$, in which, x is applied to y. In this particular case, x and y cannot be reduced any further, and the term is in its normal form and not a redex. Similarly, in $(\lambda x.\lambda y.x\ y)\ (\lambda x.x)$ 2. that first term is applied to second, and the result of the application is applied to the third term.
Additionally, we use brackets to disambiguate, which gives us an additional term:
$term\ \ =\ \ variable\ \mid\ \lambda variable.term\ \mid\ term\ \ term\ \mid\ (term)$

3 Some syntactic conventions

In expressions, whenever brackets are not used for the sake of readability, the following conventions are to be kept in mind. We will follow these conventions in order to disambiguate in absence of brackets. (If some particular meaning is intended which does not get expressed without brackets, you must use brackets to disambiguate.)

- Application binds more tightly than abstraction. For example, expression $\lambda x.x\ y$ means $\lambda x.(x\ y)$ and not $(\lambda x.x)\ y$. In other words, body of abstraction is taken as far to the right as possible, or till that closing bracket which terminates the abstraction when parenthesis are applicable. As another example, $\lambda x.e_1\ e_2\ e_3$ means $\lambda x.(e_1\ e_2\ e_3)$.
- Application associates to left. For example an expression $a\quad b\quad c\quad d$ is the same as $(((a\ \ b)\ \ c)\ \ d)$ and not $(a\ \ (b\ \ (c\ \ d)))$. So if you intend the latter, use brackets.
- $\lambda x.\lambda y...\lambda z.body$ means $(\lambda x.(\lambda y...(\lambda z.body)..))$

4 Free Variables and Bound Variables

When a variable occurs in the body of an abstraction that uses the variable in its parameter, the occurrence of this variable inside the body is called *bound*, because the occurrence is bound to this binder abstraction. If a variable occurs at a position such that there is no such outer binding abstraction, the occurrence of the variable is called *free*. For example, in term $(\lambda x.x)\ y$, y is free. In $\lambda y.xy$, x is free. In term $\lambda x.x\ x$, all occurrences of x inside the body are bound. In $(\lambda x.x)x$, first occurrence of x (inside the body) is bound, and the second occurrence is free. In $\lambda z.\lambda x.\lambda y.x\ y\ z$, the occurrences of x, y and z in the body have their respective binder abstractions. After reducing $(\lambda z.\lambda x.\lambda y.x\ y\ z)a$, where a is just a variable term, we obtain $(\lambda x.\lambda y.xya)$, in which, a is a free variable.

4.1 Closed Terms or Combinators

A term that has no free variables is called *closed*. A closed term is also called *Combinator*. For example, while $\lambda x.\lambda y.x\ y$ is a combinator, $\lambda x.\lambda y.x\ y\ z$ is not a combinator. A combinator combines its arguments in some way. We will come across many interesting combinators as we work out more examples and features.

5 β Reduction models Computation

Computation of terms occurs by application of functions to their arguments. Each step of computation is a reduction step. The right hand term e_2 in expression $(\lambda x.e_1)\ e_2$ substitutes all bound occurrences of x in body e_1 of the left hand abstraction term. An expression that is

reducible thus is called *redex*, i.e. a reducible expression. The substitution reduction is called *β Reduction*, which is stated formally as follows.
$(\lambda x.e_1)\ e_2 \ \rightarrow\ e_1[e_2/x]$.
which means, the application reduces to body e_1 with bound occurrences of x in body e_1 substituted by e_2.
An expression which cannot be further reduced is said to be in *normal form*, otherwise it's a redex. In other words, to be in normal form, the lambda expression does not allow beta reduction, i.e. it has no subexpression of the form $(\lambda x.e_1)e_2$. For example, $\lambda x.x$ is not a redex and hence is in normal form. Similarly, $\lambda x.\lambda y.x\ y$ is not a redex, and hence is in normal form. But $(\lambda x.\lambda y.x\ y)a\ b$ is a redex and is not in its normal form. After reducing it in two steps, (once for every binder abstraction), it comes to its normal form which is $a\ b$.

In a given redex, there may be more than one possible reducible terms. So, in which order should the term be reduced? We will answer this question during the discuss on reduction strategies.

6 Currying

You must have noticed (from the syntax rules) that every lambda abstraction accepts exactly one argument. How do we represent functions with multiple arguments? Consider function f(x,y) { return x+y }. This function has two parameters. By currying, we can express the same function with the help of two functions, each of which takes exactly one parameter as below.
$\lambda x.\lambda y.(x + y)$
The above expression can be rewritten as $\lambda x.(\lambda y.(x + y))$

We can see that the outer function takes x as parameter and returns a function which in turn receives y as parameter and adds this x to y. Thus we have the whole function represented as $int \rightarrow (int \rightarrow int)$ if we consider integer addition. Note that in the non-curried form, as in C-like code given above, the function was represented as $(int\ X\ int) \rightarrow int$. Even from the type signatures, we can see the difference. The curried form returns a function, which is a function of the first input argument. Let's apply the curried expression to two values 2 and 3 and see through β reduction steps, how the computation progresses.
$(\lambda x.\lambda y.(x + y))1\ 2$ reduces to $(\lambda y.(1 + y))\ 2$, which then reduces to 1+2.
For the sake of better understanding, a code equivalent to the above solution to modeling functions with multiple arguments in terms of functions with one argument (currying) is given below.
Function $f()$ takes an input parameter x and returns another function. In a code fragment

called *main*, the f is being given the first argument, and the return result fro, f is stored in variable h. h is now a function. We apply h to the second argument i.e. 3, and the final result of computation $2 + 3$ is stored in variable *result*. We can see that the first function creates another function with the first argument embedded in it. This newly created function can now be sent with the second argument to complete the computation.

```
f (x) {

    return  ( g (y) { return (x+y) } );
}

main () {


 h = f(2)
 result =  h(3);

}
```

7 Church Booleans

Lambda Calculus has no built-in terms for representing boolean values. But they can be encoded in pure lambda forms as abstractions. Boolean values encoded as lambda expressions are called Church Booleans in honor of Church who invented lambda calculus.
How to go about coming up with abstractions (i.e. functions) which will represent boolean values? The way is shown by answers to following three questions:

1. What is type?
2. What is value?
3. What is function?

We want that *true* and $false$ be represented as functions. What should they accept as argument, and what should they return? If we observe carefully, a boolean type contains 2 values, one of them, say the first one can be called as true and the second one can be called as false.

(That we represent them by 1 bit in a machine is a different matter. As an abstraction, it's enough to know that boolean type contains 2 values and the first one can be termed as true and the second as false). Now a function that takes 2 values as its parameter and returns the first one can be treated as true, when there exists also another function called false that returns the second one when arguments are passed in the same order. In terms of lambda expression, this motivation gets us to the following two functions:
$true = \lambda x.\lambda y.x$
$false = \lambda x.\lambda y.y$
These are the Church Booleans, which are boolean values encoded as two different lambda abstractions, i.e. as two different functions which take two arguments. Each function does something different than what the other function does. Once we decide that these functions are going to represent our booleans, we can now look forward to implementing simple boolean condition checking, that can be used to build control constructs such as conditional branching. We will also work out logical operators such as AND, OR and NOT in terms of lambda expressions.

7.1 Conditional Expression

Let's define a combinator ifThenElse which can be applied to three arguments such as in application $ifThenelse\ b\ v\ w$ such that if b is true, the result is v, else the result is w.
$ifThenElse = \lambda l.\lambda m.\lambda n.l\ m\ n.$
Now when the above function is applied on true v w, the result will be v.
If it is applied on false v w, result will be w. Let's see the reductions for both the cases.

Application $ifThenelse\ true\ v\ w$ is: $(\lambda l.\lambda m.\lambda n.l\ m\ n)true\ v\ w$, which reduces to
$(\lambda m.\lambda n.true\ m\ n)v\ w$, which reduces to
$(\lambda n.true\ v\ n)w$, which reduces to
$true\ v\ w$, which reduces to
$(\lambda x.\lambda y.x)v\ w$, which reduces to
$(\lambda y.v)w$, which reduces to
v.

Similarly,

Application $ifThenElse\ false\ v\ w$ is: $(\lambda l.\lambda m.\lambda n.l\ m\ n)false\ v\ w$, which reduces to
$(\lambda m.\lambda n.false\ m\ n)v\ w$, which reduces to
$(\lambda n.false\ v\ n)w$, which reduces to
$false\ v\ w$, which reduces to

$(\lambda x.\lambda y.y)v\ w$, which reduces to
$(\lambda y.y)\ w$, which reduces to
w.

7.2 Logical Operators

$and\ =\ \lambda b_1.\lambda b2.b1\ b2\ false$
$or\ =\ \lambda b_1.\lambda b2.b1\ true\ b2$
$isfalse\ =\ \lambda b.b\ false\ true$
$istrue =\ \lambda b.b\ true\ false$
$not\ =\ isfalse$
$xor\ =\ \lambda b_1.\lambda b_2.b1(not\ b2)b2$

Verify these by evaluating the following applications:

- *and true true*
- *and false true*
- *or true true*
- *or false true*
- *xor true true*

8 Pairs

We will now take an example of a non-primitive, i.e. composite value, a pair. What should we consider while trying out for a solution for representing pairs in terms of lambda calculus? The intuition can be built as follows. Firstly, we should be able to make pairs of arbitrary values (strictly they are lambda terms). Secondly, we should be able to extract the first and the second value from a pair. So we get the below usages of pairs:
pair u v (constructs a pair)
$first\ pair\ u\ v$ should return u.
second pair u v should return u.

First solution which may come to our mind is $\lambda x.\lambda y.x\ y$ as representation of function *pair*. This is not quite what we want for reasons explained in class. The solution is given below.

pair = $\lambda x.\lambda y.\lambda f.f\ x\ y$.
first = $\lambda p.p\ true$.

second = $\lambda p.p\ false$.

Now try below exercises:
(1) first pair u v (2) second pair (pair u v) (pair x y) (3) Give an abstraction $apply1^{st}To2^{nd}$ that applies first element in a given pair to its second element.

9 Church Numerals

With the same intuition that we came across in formulating booleans and pairs, we can now represent natural numbers in lambda calculus. How do we represent numbers such as 0, 1, 2, ... as lambda abstractions i.e. functions? Each one should be different from the rest. Also, applying a successor function to any number function should get us a function that represents the successor number. Then we should be able to add the numbers, multiply them, raise them to power and so on.

The core concept behind Church's solution is this: A number n can be thought of as a function that accepts another function f and applies f to another argument z exactly as many times as the value of number n. So,

C0 = $\lambda f.\lambda z.z$
C1 = $\lambda f.\lambda z.f\ z$
C2 = $\lambda f.\lambda z.f\ (f\ z)$
C3 = $\lambda f.\lambda z.f\ (f\ (f\ z))$
C4 = $\lambda f.\lambda z.f\ (f\ (f\ (f\ z)))$ and so on

9.1 Adding two Numerals

Function add can be given as:
$add = \lambda m.\lambda n.\lambda f.\lambda z.m\ f\ (n\ f\ z)$

Observe that in the inner application of the body of the add function, f is applied n times to z. To the result produced, f is applied m times. Effectively, f is being applied $(m + n)$ times to z, which is what represents the natural number $(m + n)$.

9.2 Successor Function

Function successor can be given as follows:
$successor = \lambda m.\lambda f.\lambda z.f\ (m\ f\ z)$

How does it work? Look at the body of the abstraction. In the application inside the brackets, function f is applied m times to z. To the result, f is applied once more by the outer application.

One may write the successor function as: $add = \lambda m.\lambda f.\lambda z.m\ f\ (f\ z)$

Both the forms can be obtained from the function 'add' by replacing one of m and n with C_1.

9.3 Multiplying two Numerals

Function mult is given as:
$mult = \lambda m.\lambda n.\lambda f.\lambda z.m\ (n\ f)\ z.$

In the above expression, $n\ f$ is applied m times to z. If it is applied once, it will result in $n\ f\ z$, i.e. n applications of f to z. So overall, we obtain $(m * n)$ successive applications of f to initial value z, which represents natural number $(m * n)$.

9.4 Equality with Zero

Function isZero can be worked out as:
$isZero = \lambda m.m(and\ false)true$, which can be further reduced as done soon below.

In this case (and false) is applied m times to true. If m happens to be 0, it is not applied at all, and the result is $true$. If m is non-zero, it will be applied non-zero times, giving the result as $false$.
We can reduce $(and\ false)$ further as follows:

$and\ false \rightarrow (\lambda x.\lambda y.x\ y\ false)false$
$\rightarrow \lambda y.false\ y\ false \rightarrow \lambda y.false.$
Thus we can conveniently rewrite the above function $isZero$ as:
$isZero = \lambda m.m(\lambda x.false)true.$

9.5 Implementing Subtraction

Subtraction is not straightforward in lambda calculus. The reason is you need to model subtraction through an operation which you can do n times. Observe that a number is represented as a function that accepts a function and an initial value, and applies the argument function to the value exactly as many times as the number itself. The same property of numbers needs to be used to implement subtraction as mentioned below.

subtract m n = n predecessor m

9.6 Implementing Predecessor function

We cannot directly subtract 1 from the number since we don't have any such primitive with us. We need to build the predecessor in such a way that when it is applied on a Church numeral parameter, it computes a Church numeral that represents the predecessor of the parameter. We will use the pair construct defined earlier. The strategy is demonstrated in the implementation given below, which we will convert in lambda calculus. Function next simply shifts the pair left and increments the right number in the pair.

```
predecessor (n) {
 initialize p = pair 0 0
 do n times  { p = next p }
 return first (p)
}
```

A functional representation for the above is:
$predecessor\ n = first(n\ next\ (pair\ 0\ 0))$
Before we complete the solution with lambda expressions, let's see how this solution works. to get predecessor of 3, we call predecessor. The value returned by n^{th}, i.e. third call to next can be obtained from the below trace:

```
        p = 0 0         initialized
p = 0 1 after 1st call to next p
p = 1 2 after 2nd call to next p
p = 2 3 after 3rd call to next p
```

Thus we have the predecessor in the first element of the pair.
The complete solution is:
next(p) = $\lambda p.pair(second\ p)(successor(second\ p))$
predecessor (n) = $\lambda p.first(n\ next\ (pair\ C0\ C0))$
subtract (m,n) = $\lambda m.\lambda n.n\ predecessor\ m$

10 Recursion using the Y Combinator

$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
Applying combinator Y to some function F gets us to:
YF

$\rightarrow (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F$
$\rightarrow (\lambda x.F(xx))(\lambda x.F(xx))$
$\rightarrow F((\lambda x.F(xx))(\lambda x.f(xx)))$
$\rightarrow F(YF)$
$\rightarrow F(F(YF))$... and so on

To obtain a recursive solution to $factorial$ function, we use this property of combinator Y to get us recursive calls to any function F supplied as argument to Y.

Let F be a function
$\lambda f.\lambda x.if(x = 0)C_1\ else(x * (f(x - 1)))$

(Note that though the above expression uses expressions such as if, equality with zero, subtraction and multiplication, for all of them we know we have formulations in lambda calculus. Also the infix style for * and - has been used for convinience.)

Let's now see how we compute $factorial(n)$ as $Y\ F\ n$

$Y\ F\ 3$
reducing outer Y, we get ..
$\rightarrow F(YF)3$
reducing outer F, we get ..
$\rightarrow 3 * (F(YF)2)$ through the else part. Further reducing inner F, we get ..
$\rightarrow 3 * (2 * (F(YF)1))$ and then ..
$\rightarrow 3 * (2 * (1 * (F(YF)0)))$ which reduces through if part to ..
$\rightarrow 3 * (2 * (1 * (1)))$
Thus we have obtained the recursive expansion, and we have been able to terminate the recursion by conditional executing the incoming (YF) into parameter f from inside the body of F.
f(x) = x
x is a fixpoint of f
We have fix (f) that computes fixpoint of function f
So, f (fix f) = x
i.e. f (fix f) = fix f
or fix f = f (fix f)
and so on.

11 α Conversion (or α Renaming)

We can see that $\lambda x.x$ and $\lambda y.y$ are equivalent. If the name of the formal parameter is changed and the names of the corresponding bound variables are also changed accordingly, we still have the same function. Another example of a valid renaming is a renaming of $\lambda x.x\ y$ to $\lambda z.z\ y$, but not to $\lambda y.y\ y$ (why?).
When we apply β reduction and substitute variables in abstraction by arguments, it may so happen that free variables suddenly become bound. This changes the meaning of a lambda expression. Therefore care should be taken to rename expressions appropriately. For example, consider
$(\lambda x.\lambda y.x\ y)y$. Without renaming, the application will reduce wrongly to $\lambda y.y\ y$. If the inner lambda abstraction λy is renamed to λz, we obtained $(\lambda x.\lambda z.x\ z)y$, which then reduces to $\lambda z.y\ z$, which not quite the same as $y\ y$. This process of changing the name of formal parameter in a abstraction and its corresponding bound occuranaces inside the abstraction is called α-Renaming. Perform it during reduction sequences if you find that there's a conflict between names such as the one discussed above.

12 η Conversion

Eta reduction brings out equivalence of functions that generate the same result when applied to same argument. Consider expression $\lambda x.fx$. Can we say that it's the same as f? Note that we can not reduce it to f by β reduction. But observe that if the function is applied as in $(\lambda x.fx)y$, it reduces to $f\ y$.

The η rule reduces $\lambda x.f\ x$ to f provided that x is not free in f.

Observe that if x was free in f, we cannot apply this rule. For example, if term f is $\lambda y.xy$, we get the whole term as $\lambda x.(\lambda y.x\ y)x$, which can be reduced to $\lambda x.xx$, which is not the same as $\lambda y.x\ y$. Try with a few examples of your own. Try both kinds of cases, the ones on which the rule works and the ones which violate the condition stated in the rule.
[Another way to understand the η rule is through expression $(\lambda g.\lambda x.g\ x)f$. The η rule reduces the abstraction on the left to id. Thus the result of the whole application is f. In this statement, we will not have to consider the extra condition stated in the rule above, since it is taken care of by the α rule.]

13 Evaluation Order

At any given reduction step, more than one redexes may be available for reduction. Which one should be reduced? Let's answer this question now. (Note: *Reduction* is also sometimes called *Evaluation*)
Let's consider the following expression:
$(\lambda x.x)$ ($(\lambda x.x)$ $(\lambda z.\lambda x.x)$ z)).
i.e. $id(id(id(\lambda z.idz)))$. There are more than one redex term available in this term. In the next step, which one should we reduce?
We have reduction strategies such as full beta, normal order, call by name and call by value reduction sequences.

1. Full beta reduction: Any redex may be reduced at any time.
2. Normal order reduction: Leftmost outermost redex is reduced first.
3. Call by name reduction: No reduction is allowed inside abstraction. Apply normal order, but donot evaluate under abstraction.
4. Call by value reduction: Outermost redexes are reduced, and when its argument is reduced to a value that cannot be reduced any further.

Try these strategies on the above term.
We also note the following:

– not all reduction sequences terminate
– If a normal form exists, normal order reduction finds it
– If two different reduction sequences terminate, they both find the same normal form. This is illustrated by the diagram below.

... notes incomplete. visit later. ..
In the whole expression, during reduction sequence, at any step, any one redex can be picked up and reduced.

13.1 Normal order reduction

In the normal order evaluation, the leftmost outermost redex is reduced first.
For example:
$(\lambda x.\lambda y.(\lambda p.\lambda q.p)yx)((\lambda z.z\ z)(\lambda z.z\ z))s$
$\rightarrow (\lambda y.(\lambda p.\lambda q.p)y((\lambda z.z\ z)(\lambda z.z\ z)))s$
note that we have not reduced the parameter, and only the leftmost outermost abstraction has

been reduced (with its argument as call by name). Further we get,
$\rightarrow (\lambda p.\lambda q.p)s((\lambda z.z\ z)(\lambda z.z\ z))$
$\rightarrow (\lambda q.s)((\lambda z.z\ z)(\lambda z.z\ z))$
$\rightarrow s$

13.2 Call by Name

$id(id(\lambda z.id\ z))$
reducing the outer,
$\rightarrow id(\lambda z.id\ z)$
$\rightarrow \lambda z.id\ z$
No more reductions.

13.3 Call by Value

$id(id(\lambda z.id\ z))$
reducing the argument,
$\rightarrow id(\lambda z.id\ z)$
$\rightarrow \lambda z.id\ z$
No more reductions.