CMSC 336: Type Systems for Programming Languages

Lecture 4: Programming in the Lambda Calculus

Acar & Ahmed 22 January 2008

Contents

1 Announcements **1**

2 Solution to the Exercise **1**

3 Introduction **1**

4 Multiple Arguments **2**

5 Church Booleans **2**

6 Pairs **3**

7 Church Numerals **4**
7.1 Recursion 5

8 Nameless Representation of Terms **6**
8.1 de Bruijn Indices 6
8.2 Shifting and Substitution 8

9 Homework Exercise **11**

1 Announcements

Homework 1 (second homework) is now available. It is due Tuesday. The homework has both a written part and a programming part. You should bring the written part to class next Tuesday, unless you do it electronically. In that case you can send it electronically later in the evening.

In this homework, you will learn two new tools, ml-lex and ml-yacc, so please do start early—you will not be able to finish if you start within the last two days.

2 Solution to the Exercise

Define a term t to be in normal form if there is no t' such that $t \rightarrow_\beta t'$. We say that a term t is *normalizable* if there is some t' such that $t \rightarrow_\beta^* t'$ and t' is in normal form.

1. Are there any normalizable terms?

   Yes there are many normalizable terms. Variables are in normal form, because we cannot reduce them further, but they are not that interesting. The interesting terms are closed terms that we cannot reduce further. For example, the term $\lambda\ x.x$. In general, if you define a language and a set of values that are consistent with its operational semantics, then any value will be in normal form.

2. Are there any non-normalizable terms?

   Yes, there are many non-normalizable terms. In general, the terms that "diverge" or reproduce themselves are not in normal form. For example, the term $(\lambda\ x.x\ x)\ (\lambda\ x.x\ x)$ diverges, because it beta-reduces to itself.

3 Introduction

We have defined lambda calculus and talked about how we can evaluate its terms by applying β reduction. When we started talking about lambda calculus, I claimed that it is Turing complete (even though it has no notion of numbers, primitive operations such as addition, subtraction). In this class, we will see how lambda calculus can be used to writes various short programs.

4 Multiple Arguments

Many programming languages have mechanisms to pass multiple arguments to a function. In lambda calculus, every function (lambda abstraction) has only one argument. How can we write a function that takes multiple argument?

As an example, suppose we want to write a function that sums its two arguments (let's assume for convenience that we have such a sum operator). In analogy to the languages that you have programmed in, we expect to write such a function as $\lambda(x, y).x + y$. Such a function can then be applied by writing $(\lambda(x, y).x + y)\ (3, 5)$. Since we cannot pass multiple arguments to a lambda abstraction we will instead use "currying" (named after Haskell Curry) and write the function as $\lambda x.\lambda y.x + y$. We apply the function as $(\lambda x.\lambda y.x + y)\ 3\ 5$.

$$\begin{aligned}(\lambda x.\lambda y.x + y)\ 3\ 5 \quad &\to_\beta \quad (\lambda y.3 + y)\ 5 \\ &\to_\beta \quad 3 + 5 = 8.\end{aligned}$$

Note that the result of the first application itself is a function. This is critical to the effectiveness of lambda calculus. Functions (lambda abstractions) can return functions. This is sometimes referred as having functions as first-class values. Languages based on this principle allow you to treat functions just like any other data (e.g., you can place functions in data structures).

5 Church Booleans

How can we represent booleans in lambda calculus? To find out how let's first think about how booleans are used. The "elimination" for a boolean is the `if` statement. What does an `if` statement do? It takes a boolean and two branches and picks one of the branches. Thus, if we represent boolean values as functions that select their first or second argument depending on their value, we can simulate the behavior of an `if` statement by applying the branches to the boolean value.

Based on this intuition, let define `tru` and `fls` as

$$\begin{array}{rcl} tru & := & \lambda x.\lambda y.x \\ fls & := & \lambda x.\lambda y.y. \end{array}$$

We can now define an `if` statement as a function that takes a boolean and two branches and selects the right branch as follows:

$$test := \lambda x_b.\lambda y_1.\lambda y_2.x_b\ y_1\ y_2.$$

For example, consider the term $test\ tru\ x\ y$

$$\begin{array}{rcl} test\ true\ x\ y & = & (\lambda x_b.\lambda y_1.\lambda y_2.x_b\ y_1\ y_2)\ tru\ x\ y \\ & \to_\beta & (\lambda y_1.\lambda y_2.tru\ y_1\ y_2)\ x\ y \\ & \to_\beta & (\lambda y_2\ tru\ x\ y_2)\ y \\ & \to_\beta & tru\ x\ y \\ & \to_\beta & (\lambda x.\lambda y.x)\ x\ y \\ & \to_\beta & ([x/x]\lambda y.x)\ y \\ & \to_\beta & (\lambda y.x)y \\ & \to_\beta & [y/y](\lambda y.x) \\ & \to_\beta & x \end{array}$$

How about some operations on booleans? For example, how can we write the "not" operation? Remember that booleans are function that take two arguments and select one. So we can write "not" as function that takes a boolean and two arguments and supplies the arguments to the boolean in reversed order.

$$not := \lambda x_b.\lambda y.\lambda z.x_b\ z\ y.$$

To see how "not" works, consider

$$\begin{array}{rcl} not\ tru & = & (\lambda x_b.\lambda y.\lambda z.z\ x_b\ z\ y)\ tru \\ & \to_\beta & \lambda y.\lambda z.tru\ z\ y \\ & \to_\beta & \lambda y.\lambda z.(\lambda x.\lambda y.x)\ z\ y \\ & \to_\beta & \lambda y.\lambda z.z \\ & = & fls. \end{array}$$

We can also define "not" as $not := \lambda x.x\ fls\ tru$. If the argument ($x$) is tru, then this function will return fls and will return tru otherwise.

How about "and"? Again, "and" will take two booleans and return a boolean

$$and := \lambda x_1.\lambda x_2.x_1\ x_2\ false.$$

Similarly, we can write "or" as $or := \lambda x_1.\lambda x_2.x_1\ tru\ x_2$.

6 Pairs

Suppose we want to have pairs in our language. For example, in SML the pair of the numbers 3 and 5 are written as $(3, 5)$. In addition to the ability to construct pairs, we also want to be able to project out the first and second parts using primitives such as fst and snd, e.g., $fst(3, 5) = 3$ $snd(3, 5) = 5$. Can this be done using lambda calculus.

Like with booleans, lets think about the elimination form for pairs. The elimination form for pairs are the primitives for taking a pair apart, i.e., projecting their first and second components. Thus we can think of a pair as a function that takes the elimination form as an argument and applies it to its components. The eliminations forms fst and snd simply take the pair and apply the tru and fls to project out the first and second components of the pair.

$$\begin{array}{rcl} pair & = & \lambda x_1.\lambda x_2.\lambda y.yx_1x_2 \\ fst & = & \lambda x_p.x_p(\lambda x.\lambda y.x) \\ snd & = & \lambda x_p.x_p(\lambda x.\lambda y.y) \end{array}$$

Exercise: Write the ML code for $pair$, fst, and snd for pairs of integers. Try now for a pair of a boolean and an integer (of type `bool*int`).

Answer:

```
- val pair = fn (x: int) => fn (y: int) => fn (s: int -> int -> int) => s x y;
val pair = fn : int -> int -> (int -> int -> int) -> int
- pair (3,5);
val it = fn : (int -> int -> int) -> int
- val first = fn (x: (int -> int -> int) -> int) => x (fn x => fn y => x);
- val second = fn (x: (int -> int -> int) -> int) => x (fn x => fn y => y);
- first it;
  3
- second it;
  5
```

7 Church Numerals

How can we represent natural numbers using lambda calculus? Let's again think of the elimination form for numbers. For the case of numbers, there are many of them. But they all can be characterized as computing some property of the number (e.g., comparisons, sums etc). What do we need to know to compute a property of a natural number?

We need to know what the property is for number zero and how can we update the property for each additional increment over zero. Based on this intuition we can think of a natural number n as a function that takes a function

s (successor) and z zero and applies s to z n times.

$$\begin{array}{rcl} c_0 & = & \lambda s.\lambda z.z \\ c_1 & = & \lambda s.\lambda z.s\ z \\ c_2 & = & \lambda s.\lambda z.s\ (s\ z) \\ \vdots & & \end{array}$$

This representation for natural numbers is often referred as *church numerals.* We can now write some primitives for church numerals. Let's start with a function for testing if a number is zero.

$$isZero := \lambda x_n.x_n(\lambda x.fls)\ tru$$

As examples, let's apply $isZero$ to c_0, c_1 and c_2

$$\begin{array}{rcl} isZero\ c_0 & = & (\lambda x_n.x_n\ (\lambda x.fls)\ tru)\ (\lambda s.\lambda z.z) \\ & \rightarrow_\beta & (\lambda s.\lambda z.z)\ (\lambda x.fls)\ tru \\ & \rightarrow_\beta & (\lambda z.z)\ tru \\ & \rightarrow_\beta & tru \\ isZero\ c_2 & = & (\lambda x_n.x_n\ (\lambda x.fls)\ tru)\ (\lambda s.\lambda z.s\ z)) \\ & \rightarrow_\beta & (\lambda s.\lambda z.s\ z)(\lambda x.fls)\ tru \\ & \rightarrow_\beta & (\lambda z.(\lambda x.fls)\ z))\ tru \\ & \rightarrow_\beta & (\lambda x.fls)\ tru) \\ & \rightarrow_\beta & fls \\ isZero\ c_2 & = & (\lambda x_n.x_n\ (\lambda x.fls)\ tru)\ (\lambda s.\lambda z.s\ (s\ z)) \\ & \rightarrow_\beta & (\lambda s.\lambda z.s\ (sz))(\lambda x.fls)\ tru \\ & \rightarrow_\beta & (\lambda z.(\lambda x.fls)\ ((\lambda x.fls)\ z))\ tru \\ & \rightarrow_\beta & (\lambda x.fls)\ ((\lambda x.fls)\ tru) \\ & \rightarrow_\beta & (\lambda x.fls)\ fls \\ & \rightarrow_\beta & fls \end{array}$$

Let's define some arithmetic operations.

$$\begin{array}{rcl} succ & := & \lambda x_n.\lambda s \lambda z.x_n s\ (s\ z) \\ plus & := & \lambda m.\lambda n.\lambda s.\lambda z.ms(nsz) \\ times & := & \lambda m.\lambda n.m(plusn)c_0 \end{array}$$

Exercise: Write the subtraction operation for church numerals.

7.1 Recursion

In lambda calculus, we can only write *anonymous functions*, *i.e.*, function that do not have names. This is limited because it seems to prohibit us from writing recursive functions. If we can't name a function how can we call it within the body of that function?

It turns out that recursion is not a problem. Why? Although lambda calculus does not allow us to name functions, it allows us to duplicate terms. Consider for example the term $\omega = \lambda x.x\ x$ and the application $\omega\ t = (\lambda x\ x)\ t$. By beta reduction we have $\omega\ t = t\ t$. We now have two copies of the term t. Based on this intuition, it is possible to construct a *fix-point* combinator that gives us the "recursive form" of an anonymous function.

Suppose we want to write a recursive function `g`. How can we write such a function? Here is an example `λ f. λ x. if x <= 0 then 1 else x*f (x-1)` . Now if we can get a hold on the recursive form of `g`, denoted `G`, then we can call `g` with `G`. This is often referred to as *open recursion*. This is the idea behind our approach. We call the recursive version `G` the fix-point of `g`, denoted `fix g`. There are several ways we can come up with a *fix-point combinator*, a combinator that when applied to `g` gives us its fix points. They are all similar to the term ω.

Let's develop the intuition a bit further. Suppose we are given the `g`. To apply `g` to itself twice, we want to have a way of making two copies of `g`, and retain the ability to make further copies. Can we use ω? One problem with ω is that loses its ability to replicate after one application. How about $\omega\ \omega$, *i.e.*, the term `Ω = (λ x. x x) (λ x. x x)`. This combinator reproduces itself! Let's now stick a function argument `f` into Ω, `λ f. (λ x. f (x x)) (λ x. f (x x))`, and let's call this the `Y` combinator. Now note that `Y g` $\rightarrow_\beta$ `g (Y g)`. This suffices to give us recursion under the restriction that we delay evaluation of `Y g` until after `g`. This is exactly what the call by name evaluation would do. The `Y` combinator thus suffices as a fix-point combinator in the call-by name setting.

Does this work in the call-by value setting? Not quite. Because, the in the call by value setting `g (Y g)` $\rightarrow_\beta$ `g ( g (Y g))` and the evaluation diverges. We need a different combinator. This new combinator is very similar to the `Y` combinator—it just suspends evaluation of the fix point operator until it is applied by `g`. This is called the `Z` combinator. Here it is: `Z = λ f. (λ x. f (λ y. x x y)) (λ x. f (λ y. x x y))`.

8 Nameless Representation of Terms

Last class we finished the discussion of untyped lambda calculus. Defining the calculus was easy. Programming with lambdas can be a bit tricky but it was fun. We even showed that we can write recursive function without having the ability to name functions themselves.

In this class, we will talk about de Bruijn indices. This is a neat trick that allows us to do substitution without doing lambda conversions.

8.1 de Bruijn Indices

Lambda conversions require coming up with fresh names on the fly. This can be quite difficult, so people come up with ideas to get around this problem. In

this class, we will talk about a particular idea due to de Bruijn (pronounced "de brown").

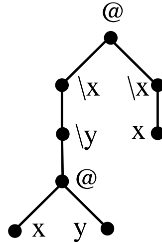

Figure 1: The ASTs for (λ x. λ y. x y) (λ x. x) with pointers.

Consider the term `(λ x. λ y. x y) (λ x.x)` and its abstract syntax tree (see Figure 1).

Can we represent the same AST without using variable names? Yes, the variable names do nothing but refer to the lambda where that variable is bound. So we can simply represent them explicitly via pointers.

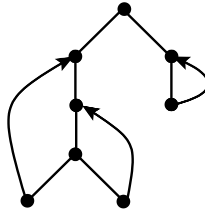

Figure 2: The ASTs for (λ x. λ y. x y) (λ x.x) with pointers.

Figure 2 shows such a representation. Note that when drawing the ASTs for lambda term with pointers, the labels are redundant. The degree of a node determines the kind of a term (application has degree two, lambda abstraction has degree one, and the leaves are variables). This is the key idea behind de Bruijn indices.

It is going to be difficult to work with AST's. Can we find a textual representation for de Bruijn's AST's?

How about representing pointers as the distance they travel? For example, the AST in Figure 2 can be written as `(λ. λ. 1 0) (λ. 0)`. We measure the distance as the number of lambdas (starting at zero) from the leaf corresponding to the variable to the binding lambda. If given the textual representation based on distances, we can construct the AST, then this approach is sound. But is it?

The answer is yes. The reason is that the pointers can only go from one node to a node that is on the path from that node to the root. So distances, uniquely identify the binding location. But why is that the pointers can only go up along the path from the node to the root? Because a node can only refer

to a variable bound by a lambda above it. So the representation based on the distances works. The distances are known as *de Bruijn* indices.

Let's develop the idea that a node can refer only to a variable that is bound on the path to the root. Let's denote the *(naming) context*, denoted Γ of a node (*i.e.*, a term) as a relation that maps variables to their de Bruijn indices. We can write Γ as a sequence $\Gamma = x_n x_{n-1} \ldots x_0$ to represent the relation $\{(x_n, n), (x_{n-1}, n-1), \ldots (x_0, 0)$. In other words, the de Bruijn index of a variable is determined by its position in the sequence. Note that there can be multiple instance of the same variable, *e.g.* $\Gamma = xyzxy$.

How can we construct the naming context for each node in an AST?

Consider some node in the tree and suppose we have the current context, Γ, for that node. We can construct the context for its subtree(s) as follows. If the root node is not a lambda then the same context is passed to the subtree(s). If the root is a lambda then we pass Γ, x to the subtrees. This extends Γ with x, which is given an index of 0, and it shifts the indices of all other variables by one, which is required because they are now further away. To construct the naming context for each node, we apply this idea starting at the root node. Note that the subtrees can be traversed in parallel, so we need not specify the ordering.

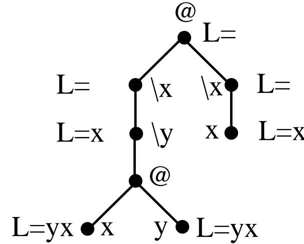

Figure 3: The ASTs for (λ x. λ y. x y) (λ x. x) with naming contexes.

For example, Figure 3 shows the naming context for each node (or term) in the term `(λ x. λ y. x y) (λ x. x)`.

Using the naming contexts, we can transform a term into its de Bruijn representation. This is relatively straightforward. Just replace each variable node with the smallest index of the variable in the context.

Now that we know what is going on, let's write it out. Let $dB(\Gamma, \cdot)$ be a function that transforms a lambda term to a de-Bruijn term under the context lambda.

$$
\begin{aligned}
dB(\Gamma, x) &= \Gamma_x \\
dB(\Gamma, \lambda x.t) &= \lambda.dB((\Gamma, x), t) \\
dB(t_1\ t_2) &= dB(t_1)\ dB(t_2)
\end{aligned}
$$

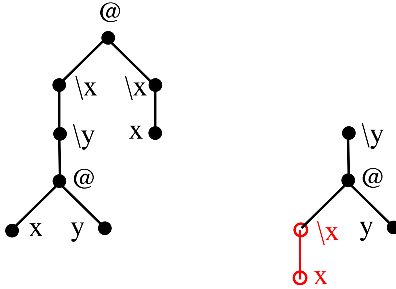

Figure 4: $(\lambda x.\lambda y.x\ y)\ (\lambda x.x)\ \rightarrow_\beta\ \lambda y.(\lambda x.x)\ y$ illustrated.

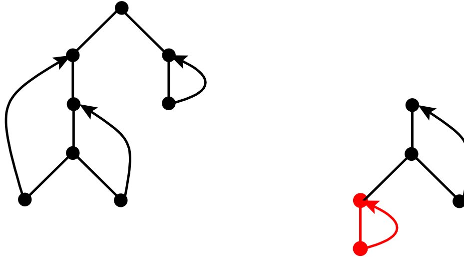

Figure 5: $(\lambda.\lambda.1\ 0)(\lambda.0)\ \rightarrow_\beta\ \lambda.(\lambda.0)\ 0$ illustrated

8.2 Shifting and Substitution

We can visualize beta reduction as an operation on AST's as substituting the right subtree of a degree two node into the left subtree at the leaves specified by the variable name being bound. For example, Figure 4 show a substitution. Since the term being substituted has not free variables, we need not work about alpha conversion.

How does substitution work with de Bruijn indices? It is very similar, except that we don't need to worry about renaming whatsoever. It just works. This example is somewhat simple because the term being substituted term does not have any free variables.

What happens with free variables? Nothing interesting, they just keep pointing to the same node that they used to (no binding location that the subtree being substituted can disappear). Figure 6 shows an example. The deleted elements are shown dashed.

Let's write out a formula for substitution. To get some intuition, consider the last example. The reduction, $(\lambda.(\lambda.\lambda.1\ 0)(\lambda.1)\ \rightarrow_\beta\ \lambda(\lambda\lambda.2\ 0)$, transforms an expression into something totally unrecognizable. Let's see what is going on.

Imagine yourself substituting a tree T in place another leaf node. Remember that during substitution the pointers keep pointing to the same binding locations. But then their length may change—we may have to stretch them. Which pointers stretch?

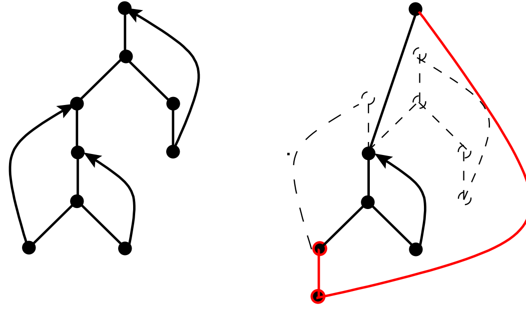

Figure 6: $\lambda.(\lambda.\lambda.10)\ (\lambda.1)\ \rightarrow_\beta\ \lambda.(\lambda.\lambda.2\ 0)$ illustrated

Only the pointers that point outside T (free variables) stretch. Let t be the term for T, to reflect the stretching, we will have to shift t. Let's define a d-place shift of a term t, denoted $\uparrow^d (t)$, as a term where all the free terms are shifted by d (increased by d). But how do we identify which variables are free? The key insight is that the bound variables always constitute the smallest indices in a term (because they are closest). We can thus carry along a parameter called *cutoff* that is set the smallest index a free variable may have. Here is the definition:

$$\begin{aligned}
\uparrow^d_c (i) &= i \quad \text{if } i < c \\
\uparrow^d_c (i) &= i + d \quad \text{if } i \geq c \\
\uparrow^d_c (\lambda.t) &= \lambda.\uparrow^d_{c+1} (t) \\
\uparrow^d_c (t_1\ t_2) &= (\uparrow^d_c (t_1))\ (\uparrow^d_c (t_2))
\end{aligned}$$

We will write $\uparrow^i (t)$ to mean $\uparrow^i_0 (t)$.

Examples:

1. $\uparrow^2 (\lambda.\lambda.1(0\ 2)) =$
2. $\uparrow^2 (\lambda.01(\lambda.0\ 1\ 2)) =$

We can now define substitution.

$$\begin{aligned}
[t/j]i &= t \ \text{ if } i = j \\
[t/j]i &= i \ \text{ if } i \neq j \\
[t/j]\lambda.t' &= \lambda.[\uparrow^1 (t)/j+1]t' \\
[t/j]t_1\ t_2 &= [t/j]t_1\ [t/j]t_2
\end{aligned}$$

Based on substitution, we define beta reduction as

$$(\lambda.t_1)t_2 = \uparrow^{-1} ([\uparrow^1 (t_2)/0]t_1)$$

9 Homework Exercise

1. Give the call by name and call by value operational semantics for lambda calculus and show that they are deterministic. That is for each semantics show that if $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$.

2. Define g := λ `f.` λ `x. if x <= 0 then 1 else x*f (x-1) .` Apply beta reduction on $g(Z\ g)$ until g is unrolled a few times to have the `if` term to be called two times. Explain $(Z\ g)$ is the fixpoint of g in no more than five lines.

3. Give the de Bruijn representation for the term tru, fls, and $pair$ and write out each step of the reduction $isZero\ c_2 \rightarrow^*_\beta fls$ in the Bruijn notation.