CS704: Lecture 3
Lambda Calculus: Substitution, Reduction, and Normal Forms

Thomas Reps
[Based on notes taken by Tycho Andersen on January 25, 2010]

Abstract

This lecture concerns the rules for computing on terms in the λ-calculus. It discusses substitution, reduction rules, and normal forms.

1 Introduction and Review

The λ-calculus model of computation was invented by Alonzo Church in the 1930's. In today's terms, it can be thought of as a (very!) stripped-down functional programming language. It is a *declarative* (as opposed to *imperative*) programming language: there is no notion of assignment. In fact, all computation is performed by rewriting one λ-term to another. Roughly, a computation halts (it if does) when no more rewriting is possible. Despite the stripped-down nature of the language, λ-calculus is Turing-complete. That is, any problem that can be formulated as a Turing-machine computation can be formulated as a λ-calculus computation, and vice versa.

Last time we gave a deliberately imprecise rewriting rule (the "naive" rule of β-reduction):

$$(\lambda x.M)N \rightarrow_\beta M', \text{ where } M' \text{ is } M \text{ with all occurrences of } x \text{ in } M \text{ replaced by } N.$$

In English, given a function applied to an argument, rewrite it to the function body (which is $x+1$ in the example above) with all occurrences of the formal (x above) replaced with the argument (3 above).

2 How Does λ-Calculus Stack Up as a Computational System

To see how the λ-calculus stacks up as a computational system, let's ask (and answer) a few rhetorical questions.

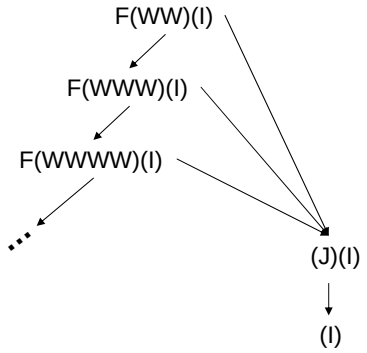
Does every λ-term have a normal form? No, there exist λ-terms that do not have a normal form. In particular, consider the following two examples:

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \ldots$$

$$(\lambda x.xxx)(\lambda x.xxx) \rightarrow_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow_\beta \ldots$$

In the first example, at each step the term rewrites to itself, creating an infinite loop. In the second example, an extra copy of $(\lambda x.xxx)$ is introduced at each step, so the term grows in size with each "reduction" step. (Thus, reduction is a misnomer, because the *size* of the λ-term that results from a "reduction" step does not necessarily decrease.)

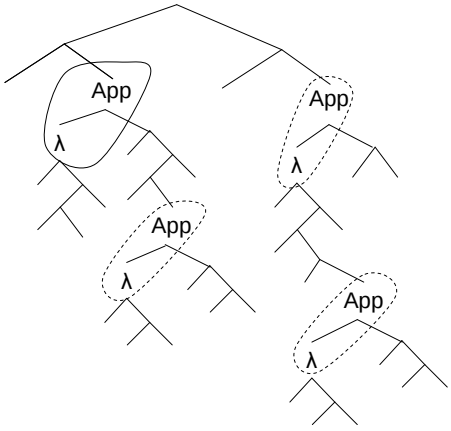
If a λ-term has a normal form, does every choice of reductions lead to that normal form? No, consider the following λ-terms $W \stackrel{\text{def}}{=} (\lambda x.xxx)(\lambda x.xxx)$, $F \stackrel{\text{def}}{=} \lambda x.\lambda y.y$, $I \stackrel{\text{def}}{=} \lambda x.x$, $J \stackrel{\text{def}}{=} \lambda y.y$, and the transitions possible from the λ-term $F(WW)I$.



If one always chooses to perform a reduction step that rewrites an occurrence of WW, the reduction sequence never terminates. However, at each stage each λ-term in the chain on the left-hand side can, in two reduction steps, be reduced to a normal form, as shown in the diagram.

The example relates to the programming-language notion of *strictness*: a function is *strict* in a given argument position i if whenever the i^{th} actual parameter diverges, the function diverges. An abstraction in λ-calculus, therefore, is *not* strict in its (one) argument.

If a λ-term has a normal form, is there a reduction strategy that will always find it? Yes, by repeatedly performing the leftmost-outermost reduction. The leftmost-outermost β-redex can be found by simply doing a pre-order traversal of the abstract-syntax tree until the first β-redex is encountered. Below is a diagram that shows a λ-term that has a total of four redexes, two of which are outermost redexes. The one circled with a solid line is the leftmost-outermost redex.



Applied λ-calculi. Other languages can be considered as "sugared" (or "applied") λ-calculi [1]. In "pure" λ-calculus, there are no primitive data values or functions, but applied λ-calculi can be created by adding

- constant symbols (e.g., 0, 1, 2, ...; 'a', 'b', ...; "", "a", "aa", ...)
- constant functions (e.g., "+", "*", ...)

(as well as possibly other elements).

An example of a term in an applied λ-calculus is $\lambda x.x+1$. Again, to compute one would repeatedly find occurrences of an application of a λ-term (say, $\lambda x.x + a$) to an argument (which is itself also a λ-term). Computation is performed by repeatedly performing a rewriting step on the current λ-term until one is left with a term that cannot be rewritten any further, e.g.,

$$(\lambda x.x + 1)3 = 3 + 1 = 4$$

Is the fact that λ-calculus has only one-argument functions a limitation? No, multi-argument functions can be built up by using nested abstractions. For instance, $\lambda x.\lambda y.x$ is an example of a "two-argument" function built by nesting $\lambda y.x$ inside of $(\lambda x. \ldots)$. $\lambda x.\lambda y.x$ is a sort of "first" function. That is, it takes two actual parameters and returns the first of them, ignoring the value of the second actual parameter.

λ-calculus allows familiar functions to arise in unfamiliar ways. In an applied λ-calculus, the "add" function is $\lambda x.\lambda y.x+y$. This "two-argument" function can be used to form more specific functions, such as the "add three" function, by rewriting:

$$(\lambda x.\lambda y.x + y)3 = \lambda y.3 + y$$

The example illustrates that in λ-calculus the add function is a *higher-order function*: it takes an actual parameter and gives back a function as the result. (Another kind of higher-order function takes a function as an actual parameter. In λ-calculus, all functions are higher-order in this sense.)

Rewriting in the λ-calculus provides interesting ways of manipulating programs. Suppose that instead of $\lambda y.3 + y$, we wanted to obtain $\lambda y.y + 4$ from the add function $\lambda x.\lambda y.x + y$. How can we obtain this? Note that

$$(\lambda x.\lambda y.x + y)4 = \lambda y.4 + y,$$

which is different from the desired result.

We can create the desired term by using a *combinator* (i.e., a λ-term with no free (unbound) variables). If we apply the combinator $\lambda f.\lambda a.\lambda b.fba$ to the add function, and then apply the resulting λ-term to 4 we have:

$$\begin{array}{rcl} ((\lambda f.\lambda a.\lambda b.fba)(\lambda x.\lambda y.x + y))4 & \rightarrow & (\lambda a.\lambda b.(\lambda x.\lambda y.x + y)ba)4 \\ & \rightarrow & (\lambda a.\lambda b.(\lambda y.b + y)a)4 \\ & \rightarrow & (\lambda a.\lambda b.b + a)4 \\ & \rightarrow & \lambda b.b + 4 \end{array}$$

Remark. Combinators will play a role in the next lecture when we introduce *encodings* that (i) simulate data of various kinds, (ii) operations on that data, and (iii) various programming constructs (such as conditional expressions and recursion). □

Reasoning about programs. Rewriting (reduction, substitution of equals for equals) gives us a tool for reasoning about programs. Such reasoning is difficult for other languages if either (i) it has notions of assignment, destructive updating, etc., or (ii) all you have is a "definition" of the language based on what is produced by a compiler.

Recursion. The λ-term $\lambda x.x$ represents the identity function, and hence is roughly equivalent to the (unnamed) C function defined by

```
datum ?(datum x) {
   return x;
}
```

What corresponds to

```
int fact(int x) {
   return(x == 0 ?  1 :  x * fact(x-1));
}
```

The answer is that there is an analog of recursive definitions in the λ-calculus, but in pure λ-calculus the problem is that functional expressions do not have permanent names (as `fact` does above). That feature is key for recursion to work in C: in the body of `fact`, the name `fact` is used to make the recursive call.

In λ-calculus, a combinator will let us solve the problem of not having permanent names. (In essence, the combinator's bound variables essentially "issue" temporary names.)

3 A Precise Definition of β-Reduction

First, some terminology:

- Scoping: With which enclosing occurrence of a λ is each variable associated? (In λ-calculus, one rarely talks of scopes, but in conventional programming-language terms, each occurrence of $(\lambda x.\ldots)$ creates a scope and the λx part is like a declaration of x that is added for that scope.)
- Bound variable: A variable that is associated with an enclosing λ.
- Free variable: A variable that is *not* associated with an enclosing λ.

Note that it is important to be clear about which (sub-)term is being referred to. For instance, consider a λ-term of the form $P \stackrel{\text{def}}{=} \lambda x.M$, and assume that none of the abstractions inside of M are of the form $(\lambda x.\ldots)$. The occurrences of variable x in P are *bound variables in* P. However, if we are just considering M alone, the occurrences of x in M are *free variables in* M.

Definition 3.1 (Free and bound variables) *The free and bound variables of a λ-term are defined in terms of the structure of the term, as follows:*

1. *In an expression of the form x (i.e., a variable),*

- *x is free*
- *there are no bound variables*

2. *In an expression of the form $\lambda x.M$:*
   - *every x in M is bound*
   - *for every other variable y, if y is free in M, then it is free in $\lambda x.M$; if y is bound in M, then it is bound in $\lambda x.M$.*

   (Note that a variable y can be both free and bound in M: some occurrences of y can be free in M, while if M has a sub-term of the form $(\lambda y \ldots)$, other occurrences of y can be bound in M.
3. *In an expression of the form MN: this is the same as in 2, but for both M and N.*

Alternatively, the free variables can be defined inductively as follows:

$$\begin{array}{rcl} FV(x) & \stackrel{\text{def}}{=} & \{x\} \\ FV(\lambda x.M) & \stackrel{\text{def}}{=} & FV(M) - \{x\} \\ FV(M\ N) & \stackrel{\text{def}}{=} & FV(M) \cup FV(N) \end{array}$$

□

Next, we introduce the correct β-reduction rule. The naive β-reduction rule has some problems because it does not respect scoping.

- Sometimes we do not want to replace *all* the occurrences of the formal in the function body. In the example below, the variable x is overloaded, and replacing all occurrences yields the wrong answer:

$$\begin{aligned} (\lambda x.x + ((\lambda x.x + 1)(3)))2 &= 2 + ((\lambda x.2 + 1)(3)) \\ &= 2 + 2 + 1 \\ &= 5 \end{aligned} \qquad \text{(Bad!)}$$

  To avoid this problem, we will use a rewriting rule called α-reduction, which will be used to rename formal parameters. Below is the correct evaluation.

$$\begin{aligned} (\lambda x.x + ((\lambda x.x + 1)(3)))2 &= 2 + ((\lambda z.z + 1)(3)) \\ &= 2 + (3 + 1) \\ &= 6 \end{aligned} \qquad \text{(Ok!)}$$

- The naive β-reduction rule permits a variable that is free in an argument to become bound. Consider $\lambda x.\lambda y.x$, which is a two-argument function that returns the first argument and ignores the second. If the variable names are not overloaded (i.e., the actual parameter(s) do not have the same name(s) as a formal parameter(s) in a nested function(s)), we can successfully evaluate this function using the naive rule:

$$((\lambda x.\lambda y.x)w)z = (\lambda y.w)z = w$$

  However, if variable names are overloaded, we may obtain the wrong answer.

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

α-reduction is applied to abstractions. In $\lambda x.M$ it replaces all *free* occurrences of x in M with some other identifier that is not in M. Some examples:

$$\lambda x.(\lambda y.x) \rightarrow_\alpha \lambda z.\lambda y.\lambda z$$
$$\lambda x.(\lambda x.x) \rightarrow_\alpha \lambda z.\lambda x.\lambda x$$

Formally, the α-reduction and β-reduction rules are defined in terms of the operation of *substitution* defined in Defn. 3.2.

Remark. I use several kinds of equality-like symbols:

- $P \stackrel{\text{def}}{=} Q$ means that P is defined to be the λ-term Q.
- $P \equiv Q$ means that P and Q are identical. Thus, $P \not\equiv Q$ means that that there is some difference between P and Q.

□

Definition 3.2 (Substitution) *The substitution of a term N for all free occurrences of a variable x in M, denoted by $M[N/x]$,*[1] *is defined inductively as follows, according to the structure of M:*

1. $x[N/x] \stackrel{\text{def}}{=} N$
2. $y[N/x] \stackrel{\text{def}}{=} y$ *iff* $y \not\equiv x$
3. $MP[N/x] \stackrel{\text{def}}{=} (M[N/x]P[N/x])$
4. $(\lambda x.M)[N/x] \stackrel{\text{def}}{=} \lambda x.M$
5. $(\lambda y.M)[N/x] \stackrel{\text{def}}{=} \lambda z.((M[z/y])[N/x])$, *where* $y \not\equiv x$ *and* z *is fresh*

□

Definition 3.3 (Reduction Rules) *The rules of α-reduction and β-reduction are defined as follows:*

α-reduction: $\lambda x.M \rightarrow_\alpha \lambda y.M[y/x]$
β-reduction: $(\lambda x.M)N \rightarrow_\beta M[N/x]$.

□

References

[1] P. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[1] $M[N/x]$ is properly read as "M with N substituted for all free occurrences of x"; however, it is often shortened to "M with N in for x".