Thursday Week 2 - Lecture 4

Ben Lain, Logan Poole

October 19, 2019

1 Lambda calculus

Today, we'll be continuing from tuesdays discussion of a very simple language that only deals with functions (lambda calculus). We remind you that lambda-calculus terms (i.e. programs) are defined as follows:

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & \mid & \lambda x.M & \text{abstraction} \\ & \mid & M\ M & \text{application} \end{array}$$

where x is a base case, and $\lambda x.M$ and MM are the recursive cases. $\lambda x.M$ stands for an anonymous function; this concept is also present in languages like Ruby, even though they don't use the symbol lambda.

Semantics: Axioms and Rules

In order to fully understand a language, we need to understand its semantics. We explain λ-calculus semantics in terms of a simple model based on rewriting or simplifying programs. We have two axioms:

(α)

$$\lambda x.M = \lambda z.[z/x]M \qquad z \text{ a fresh new variable}$$

(β)

$$(\lambda x.M)N = [N/x]M$$

The β axiom explains what to do when you have to evaluate a function application: you replace the formal parameter x with the actual parameter N. The α axiom is about renaming: you can rename a variable with a variable that has not already been used.

Remark 1. Remember that in Assignment 1, in addition to the axiom

$$n + m \to +(n, m)$$

we had inference rules that were telling us how to apply the axiom:

$$\frac{e_1 \to e_1{}'}{e_1 + e_2 \to e_1{}' + e_2} \qquad \frac{e_2 \to e_2'}{m + e_2 \to m + e_2'}$$

These rules were capturing a left-to-right evaluation. For example, giving the expression $(2+3)+(9+1)$, we were first executing $(2+3)$ to 5, then evaluating $(9+1)$ to 10 and then performing $5+10$. If instead we wanted a right-to-left evaluation we would have the following inference rules:

$$\frac{e_2 \to {e_2}'}{e_1 + e_2 \to e_1 + e_2'} \qquad \frac{e_1 \to e_1'}{e_1 + m \to e_1' + m}$$

With these inference rules, there is no ambiguity on which operation to execute. If instead, we just give the axiom then it would mean that you can apply the axiom in any order you want. So for example both

$$(2+3)+(9+1) \to 5+(9+1) \qquad \text{and} \qquad (2+3)+(9+1) \to (2+3)+10$$

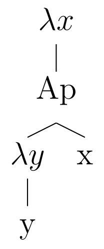
would be legal applications of the axiom. Analogously, for λ-calculus we are not specifying where to apply the axioms, so for example both rewritings:

$$(\lambda x.x)((\lambda z.z)w) \to (\lambda z.z)w \qquad \text{and} \qquad (\lambda x.x)w$$

are possible. The interesting result is that the order of application of the axioms does not impact the final result of the program.

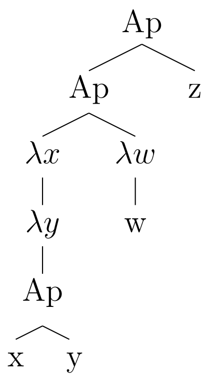
Example 2.

- Let's look at the program $\lambda x.(\lambda y.y)x$, and we'll see that it is transformed into $\lambda x.x$ by application of axiom β. The abstract syntax tree of $\lambda x.(\lambda y.y)x$ is:



  Note that the β axiom does not apply to the top of the program but just below the λ-node.

- Consider $(\lambda x.\lambda y.x\ y)(\lambda w.w)z$ whose abstract syntax tree is[1]



[1]Remember that application is left associative. Thus, $M\ M\ M$ is read as $(M\ M)\ M$.

As before there is no way we can apply the β-axiom to the whole program, since the left child of the the Ap-node is not labelled with a λ. However, we can apply the rewriting on the left child, as follows:

$$(\lambda x.\lambda y.xy)(\lambda w.\lambda w)z \rightarrow (\lambda y.(\lambda w.w)\ y)z$$

Note that we can represent the above rewriting as an optimization of JavaScript programs :

```
(function(x){return (function (y) {return x(y);});})
    (function(w){return w;})
     (z)   -->
     (function (y){return (function (w){return w;}) (y);}) (z)
```

- Consider $(\lambda y.(\lambda w.w)\ y)\ z$ further. It can be reduced in two ways depending on where we apply the β-axiom, as seen below.

$$(\lambda y.(\lambda w.w)\ y)\ z \rightarrow (\lambda y.[y/w]y)\ z = (\lambda y.y)\ z \rightarrow z$$
$$(\lambda y.(\lambda w.w)\ y)\ z \rightarrow [z/y]((\lambda w.w)\ y) = (\lambda w.w)\ z \rightarrow z$$

We can note, though, that both ways lead to the same answer z. In general, in lambda calculus this will be true. We should always obtain the same answer regardless of in which order we apply the axioms.

- Let's consider one last example of reduction:

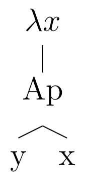
$$(\lambda x.x)((\lambda w.w)z) \rightarrow (\lambda w.w)\ z \rightarrow z$$
$$(\lambda x.x)((\lambda w.w)z) \rightarrow (\lambda x.x)\ z \rightarrow z$$

2 Free Variable Capture

So far we've been fairly informal with respect to substitution. The last lecture ended with an issue: We started with two programs that were the same, worked with both of them, and wound up with two different programs. Consider $(\lambda x.\lambda y.x)\ y$ and $(\lambda x.\lambda z.x)\ y$. By axiom α, they should both be the same, but if we naively apply our notion of substitution as below, we have a problem.

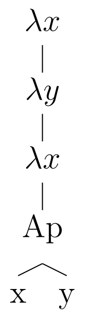
$$(\lambda x.\lambda y.x)\ \underline{y} \rightarrow [y/x]\lambda y.x = \lambda y.y$$
$$(\lambda x.\lambda z.x)\ \underline{y} \rightarrow [y/x]\lambda z.x = \lambda z.y$$

It turns out that the first rewriting is the wrong one. The underlined occurrence of y is called a *free variable.* A variable if called 'free' if there is no declaration of it. Basically, we acted as though there's a relationship between that occurrence of y and the y defined as a formal parameter in $\lambda y.x$, when there wasn't one. We can understand this by examining the tree of a lambda expression. Consider $\lambda x.y\ x$ and its abstract syntax tree below.



If you follow a path from y up the tree, we can't find a declaration of it, in other words a node labelled with λy. On the other hand, if we follow a path for x, we do find such a declaration, λx. So, x is bound and y is free.

But, now consider the tree below. Which definition of x should x correspond to?



We say that it is bound by the first definition you find when traveling up the tree from x. This is similar to the way definitions work in programming languages like Java.

Knowing the difference between free and bound variable, now we can say that the problem with the rewriting

$$(\lambda x.\lambda y.x)\ y \rightarrow \lambda y.y$$

is that a variable which was free becomes bound. By allowing this phenomena to occur leads to a language being dynamically scoped, as Lisp. More on the difference between static and dynamic scope will be covered when we study the runt-time system. For now, let's define the notion of substitution.

3 Substitution

Let us start with the easy cases:

$$\begin{aligned}
[z/x]x &= z \\
[z/x]y &= y \\
[z/x]MN &= ([z/x]M)([z/x]N)
\end{aligned}$$

What would be the result of the following substitution:

$$[N/x]\lambda x.x$$

If we accept the α-axiom we accept the fact that $\lambda x.x$ is the same as $\lambda z.z$ so let's try asking a different question, what is the result fo the following substitution:

$$[N/x]\lambda z.z$$

Since we do not see any occurrences of x in $\lambda z.z.$ then we should say

$$[N/x]\lambda z.z = \lambda z.z$$

Since operations on equal terms produce equal results we do not have any other choice than saying that the result of our first question is:

$$[N/x]\lambda x.x = \lambda x.x$$

So we add one more clause to our formal definition of substitution:

$$[N/x]\lambda x.M = \lambda x.M$$

What should we do for the following substitution:

$$[y\ x/z]\lambda w.z$$

We should propagate the substitution as we did for application:

$$[y\ x/z]\lambda w.z = \lambda w.y\ x$$

But what about the following one:

$$[y\ x/z]\lambda x.z = \lambda x.y\ x$$

We know that the above is not correct because we encounter the free variable capture. We express the fact that it is correct to propagate the substitution under a lambda if we satisfy a proviso as follows:

$$[N/x]\lambda w.M = \lambda w.[N/x]M. \qquad w \text{ does not occur free in } N$$

What if w occur free in N what should we do? We apply renaming! As for example in order to do the substitution $[y\ x/z]\lambda y.z\ y$, since y occur free in $y\ x$ we rename first as:

$$[y\ x/z]\lambda y.z\ y = [y\ x/z]\lambda w.z\ w = \lambda w.y\ x\ w$$

We put all the clauses together as follows:

$$\begin{array}{lcll}
[N/x]x & = & N & \\
[N/x]y & = & y & \\
[N/x]MN & = & ([z/x]M)([z/x]N) & \\
[N/x]\lambda x.M & = & \lambda x.M & \\
[N/x]\lambda w.M & = & \lambda w.[N/x]M & \quad w \notin FV(N)
\end{array}$$

John McCarthy, the inventor of Lisp, implemented the notion of substitution wrong. He did not apply the renaming step, and that led to Lisp being dynamically scoped.

Remark 3. Note that we can't just rename our free variables with the name of other free variables in the same program! You might change the result of your program. Imagine having two global variables you're dealing with in a function in java, and swapping their names inside the function! You could change the output. Consider the following code:

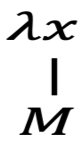


Figure 1: Pattern for $\lambda x.M$

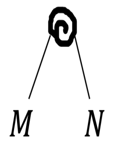


Figure 2: Pattern for MN

```
int y = 2;
int z = 0;
int w = 5;
int f(int x) { return x + y; }
```

Notice that variable `y` is free inside function `f`. We have:

`f(10)` evaluates to `12`

But if we rename `y` to `z` we obtain:

```
int f(int x) { return x + z; }
```

and:

`f(10)` evaluates to `10`

Free Variables

We want to compute the following function:

$$FV : \lambda\text{-term} \rightarrow \text{free variables}$$

We definite FV by cases on the structure of a term:

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M) - \{x\} \\ FV(MN) & = & FV(M) \bigcup FV(N) \end{array}$$

We refer to x, $\lambda x.M$ and MM as patterns. The above definition should be read as follows: when we call the function FV with a term which is just a variable follow the first case; when the term starts with a λ (i.e. the abstract syntax tree follow the pattern displayed in Figure 1) follow the second clause, and when the term is an application (i.e. the abstract syntax tree follow the pattern displayed in Figure 2) follow the third clause.

Example 4. $FV(x(\lambda x.x)) = FV(x) \cup FV(\lambda x.x) = \{x\} \cup (FV(x) - \{x\}) = \{x\} \cup (\{x\} - \{x\}) = \{x\} \cup \emptyset = \{x\}$

Example 5. $FV(\lambda x.z) = FV(z) - \{x\} = \{z\} - \{x\} = \{z\}$

Example 6. $FV((\lambda z.z)w) = FV(\lambda z.z) \bigcup FV(w) = (FV(z) - \{z\}) \bigcup FV(w) = (\{z\} - \{z\}) \bigcup \{w\} = \emptyset \bigcup \{w\} = \{w\}$

Exercise: Given a term M, define the function BV which gives the set of bound variables occurring in M.

Example 7. Follow some more examples of α and β-reductions.

- $\lambda x.xw = \lambda z.[z/x]xw = \lambda z.zw$
- $\lambda x.xz \neq \lambda z.[z/x]xz = \lambda z.zz$
  This is an example of free variable capture - it is not a correct application of α axiom
- $(\lambda x.(\lambda x.x)x)4 \to [4/x](\lambda x.x)x = (\lambda x.x)4$
- $(\lambda x.(\lambda z.z)x)4 \to [4/x]((\lambda z.z)x) = (\lambda z.z)4$
- $(\lambda x.\lambda y.xy)y \to [y/x]\lambda y.xy = \lambda y'.yy'$
  Notice how $[y/x]\lambda y.xy$ does not return $\lambda y.yy$, but instead we first apply a renaming step.