CS 421 – Spring 2007
Lecture Notes Set 26: Lambda Calculus

Elsa L. Gunter[1]
Transcribed by: Pooja Mathur

Corresponding to Slides: `10-lambda-intro (slides 1-18)`
Made available: April 2, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

2 Motivation (slides 5-6, 14)

So you must be wondering...what is this λ-calculus...and why do I have to learn it? Well, λ-calculus is the essence of application, that is, applying a function to an argument and doing the computation involved in that application. Basically, it is trying to get down to the essence of doing computation viewed as applying a function to an argument.

Now, programs can be viewed as functions that go from some input, the initial state and values, to some output, the final state and final values. Looking at things this way, we can use λ-calculus as a way of formalizing the functions and their computations.

There are two types of λ-calculus, typed and untyped. Typed is very similar to OCaml. If you have a function that takes an α and returns a β, then when you try to use the program, you better be giving the function something of type α and you will undoubtedly get back something of type β. The untyped version doesn't care about any of this. So anywhere that you would usually have a type error would not matter when using an untyped language.

We want to be able to write sequential programs in terms of λ-calculus to study the theoretical sides of it. OCaml is very close to λ-calculus. For example, in OCaml we have fun x $\rightarrow$ exp. In λ-calculus it is translated to λ x.exp. Another example is let x = e_1 in e_2, in λ-calculus it is translated to (λ x. e_2)e_1.

3 Untyped λ-Calculus

3.1 Three kinds of expressions (slides 7-8)

There are exactly three kinds of expressions used in λ-Calculus. First are the variables, the names of arguments. This is like: x, y, z, etc. Using the BNF grammar style we have been using lately, we can describe this a little more formally. Here we are just stating the different ways we can represent expressions

<expression> $\rightarrow$ <variable>
| <abstraction>
| <application>
| (<expression>)

Next is what is called the λ abstraction. Think of this like saying fun .. $\rightarrow$ .. So if you had the variable x and then the expression e, we could write $\lambda\, x.e$ which has the meaning $fun\ x \rightarrow e$. Formally, we can say that expressions have the following format.

<abstraction> $\rightarrow$ λ <variable> . <expression>

[1] © 2007, Share and Enjoy

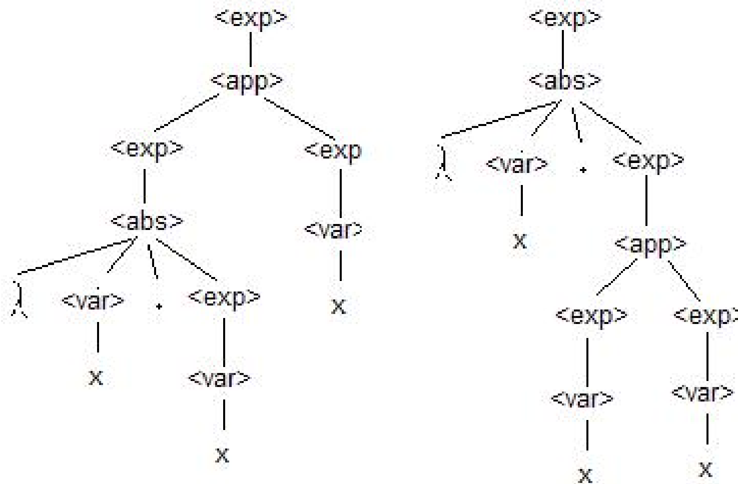
Lastly, there is application. Here, you write the function applied to the argument. And that's all you can do. And in BNF grammar terms, we have:

<application> → <expression> <expression>

Note that this last rule makes this grammar ambiguous.

Looks at the follow example. $\lambda x.xx$ How can we parse this.

We can do a couple different things, but what we want is the right tree.



To clear up the ambiguity, we need to define the scope of the λ. We need to say the λ has as broad a scope as possible. With that, the grammar will be disambiguated.

3.2 Terminology (slide 9)

An occurrence is the location of a subterm, a variable in a tree. Then there is also a free occurrence, which is an occurrence of a term that is not bound to a specific λ variable. Then there are bound occurrences, which are variables that are bound to the λ variable. It is the variable hanging below the λ in the tree. A variable is said to be free if it has a free occurrence. Variable binding simply means that there is a binding of a variable x in the expression part of the λ term. And finally, the scope of a binding lasts for all of the occurrences in a λ expression.

3.3 Example (slide 10)

Now, let's play a little game called label that occurrence and scope!!

Here is the whole expression: (λ x. λ y. y (λ x. x y)) x

We will label all the λ's 1 through 3 and the associated arguments in the same fashion. Then all of the variables a through d.

$$(\lambda_1\ x_1.\ \lambda_2\ y_2.\ y_a\ (\lambda_3\ x_3.\ x_b\ x_c))\ x_d$$

Let's start:

Question: Is y_a free or bound? Answer: Bound

Question: y_a is bound by what? Answer: $\lambda_2\ y_2$

Question: Is x_b free or bound? Answer: Bound

Question: x_b is bound by what? Answer: $\lambda_3\ x_3$

Question: Is y_c free or bound? Answer: Bound

Question: y_c is bound by what? Answer: $\lambda_2\ y_2$

Question: Is x_d free or bound? Answer: Free

Question: Why? Answer: The scope of $\lambda_1\ x_1$ only extends to the end of the parentheses. So, when we see that x_d it means that this is being used like in the application rule and so is not bound by $\lambda_1\ x_1$.

3.4 What now? (slide 11)

Now that we see what such terms look like, what is this all good for? How can we use it? Well, basically, this helps a lot when it comes to substitution. λ-calculus, when you have an application, you take your argument and apply to all the places that variable fits. You substitute it in.

So if you had (λ x. $_1$) e_2. It turns into e_1[e_2 / x].

That is, anytime you saw an x in your expression e_1, you replace it with e_2. One thing that we have to look out for, when we do this is, if e_2 had free variables in it, they would still have to be free after substituting them in for x. This is called avoiding free variable capture.

3.5 Power and Problems (slides 12-13)

Using this very simple idea, it is powerful enough to be able to express all computations, with just this. We just right down λ's and variables and apply them to each other. We don't need anything more than that.

But since it is so simple, problems arise as well. Some of the problems are that since we want to express sequential computation, we need to have an order of evaluation. Other issues are that you need to find a way to express basic data types. Also, one has to deal with representing recursion. Lastly, constants, and if_then_else statements are syntactic sugar and those need to be represented here as well.

With pure λ-calculus, there is no notion of type. So applying a function to itself is ok, when generally, if a function returns something different than it takes in, this would be a problem. So, applications of functions change, because of type restrictions. But as soon as you start adding types to your λ-calculus, you immediately start losing power. And it loses it's turing complete-ness.

4 α Conversion

4.1 Basics (slides 15-16)

So, let's look at another example first. Recall this: λ_1 x_1. λ_2 y_2. y_a (λ_3 x_3. x_b x_c)) x_d. If instead of that we had: (λ_1 z_1. λ_2 y_2. y_a (λ_3 x_3. z_b x_c)) x_d, yes, the meaning has changed, but disregard that for a moment. The z_b now binds to the z_1 in the beginning. When we try to substitute the x_d in for the z's, we get the same expression we started with and now the z_b, that is now an x_b is being bound to the x_3. Which is different than what we were trying to do.

To fix this, we need to change the name of x_3. Changing the names of variables is what α conversion does. But it also makes sure we aren't stepping over the toes of any free variables as well.

α conversion is done by doing a naive substitution after we have checked that it is ok to do that substitution. So for example, we can convert λ x.exp to λ y. (exp [y/x]) (that is, replace all the x's in exp with y) after doing to checks. First we need to make sure y is not a free variable, because after throwing it into the λ expression it will become bound. Second, you need to make sure that there are no free occurences of x in exp, because if there are, they will become bound to y after the substitution is done.

So for example, if you had λ x. x y, you can't substitute in y for x. Since y is a free variable in the λ expression, replacing the x's with y's will cause the free y to be bound. Another example is λ x. λ y. x y. You cannot replace the x's with y here either. Even though y is not a free variable here, replacing the x's will cause both variable y's to be bound to the second λ when the first of the variable y's should be bound to the first λ y. However, if you had parentheses: λ x. (λ y. y) x, the λ y scope is only for the inside of the parentheses, so you can replace the x's with y's and everything will still be ok.

4.2 Congruence (slide 17)

Now, this is the notion that two terms are congruent if they are equivalent to each other. This basically means that if you have two expression, e_1 and e_2 and they are congruent (the symbol for which we use the ˜) then, (e e_1) ˜(e e_2) and (e_1e) ˜(e_2e). Which also means that λ x. e_1 ˜λ x. e_2.