
CS 421 – Spring 2007

Lecture Notes Set 28:

Evaluations and Data Types in λ Calculus

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 10-lambda-intro (slides 29-end) 11-lambda-data (slides 1-6)
Made available: April 6, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

2 Lazy Evaluation

2.1 The idea

After learning the basics of reductions and conversions, we are in a position to use λ calculus to describe evaluation methods. So with lazy evaluation, you hold off evaluating your argument until you actually know you need it.

So an `if_then_else`, for example, can be described as taking in three arguments. But you don't want to evaluate all three at once. It will never evaluate one of those arguments. That is, it starts off with the *if* part, and with the result from that, we know should we go to the *then* part, or the *else* part, but we don't do both.

2.2 The basic procedure (slide 29)

So, you can think of a term, when given, as a variable (in which case there is nothing you can do with it), or an application. So the way you can break down your terms is by viewing it as a sequence of applications fown to a final argument. And when we look at the sequence of applications. We want to look at the left most one. So in OCaml terms, if you had a function f that took three arguments, which would look like $f\ x\ y\ z$. So by saying the left most application means we are going to focus on the $(f\ x)$ part.

So we are just going to do a step of β reduction. However, we are going to do it without reducing x in any way. Then we continue to reduce the left most application until there isn't anything left to reduce.

2.3 Example (slide 30)

Let's evaluate $(\lambda z. (\lambda x. x))((\lambda y. y\ y)(\lambda y. y\ y))$ with lazy evaluation.

Note that if we ever had to start with the right side, we learned last time, that this would simply reduce to itself forever and ever. So we would be stuck in an infinite loop. But with lazy evaluation, we don't look at the second argument.

So, to start, we will reduce the left-most application. What happens here is that our function says give me two arguments and I'll throw the first one away and then return the second argument. So after the β reduction, we get a function that takes an argument and then gives it right back.

¹© 2007, Share and Enjoy

Our function is the $(\lambda z. (\lambda x. x))$ part. It has a formal parameter of z . The argument of z is $((\lambda y. y y)(\lambda y. y y))$. When we go to substitute it in for z , we end up doing nothing since there are no z 's. So, since we never needed it, we only cared about the x 's it just went away. Even though we had an argument that failed to terminate, we were still ok. So our final answer is $(\lambda x. x)$.

2.4 Example 2 (slides 33-44)

Now, let's look at $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$.

Our function is $(\lambda x. x x)$. Our argument is $((\lambda y. y y) (\lambda z. z))$.

With lazy evaluation, first we evaluate our function. It's already reduced so our next step here is to do the substitution. We replace both x 's with the argument and get: $((\lambda y. y y) (\lambda z. z))((\lambda y. y y) (\lambda z. z))$.

Next evaluate our function (the left-most application), which is now the first $((\lambda y. y y) (\lambda z. z))$. So, we apply the $(\lambda z. z)$ to the $(\lambda y. y y)$ and get $((\lambda z. z)(\lambda z. z))((\lambda y. y y) (\lambda z. z))$.

Now we evaluate the left most application again, which is applying the identity function to the identity function so we now have: $(\lambda z. z)((\lambda y. y y) (\lambda z. z))$

At this point, the function is the $(\lambda z. z)$ and the left-most application is applying the $((\lambda y. y y) (\lambda z. z))$ to the $(\lambda z. z)$.

So we do that and since it's the identity function we just get back the argument, $((\lambda y. y y) (\lambda z. z))$.

Then, again, we do the substitution to get $(\lambda z. z)(\lambda z. z)$.

After that, we do the next application and get $(\lambda z. z)$.

We can't do any more with this, so we're done. $(\lambda z. z)$ is our final answer.

Here are all the steps:

$$\begin{aligned} &(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \rightarrow \\ &((\lambda y. y y) (\lambda z. z))((\lambda y. y y) (\lambda z. z)) \rightarrow \\ &((\lambda z. z)(\lambda z. z))((\lambda y. y y) (\lambda z. z)) \rightarrow \\ &(\lambda z. z)((\lambda y. y y) (\lambda z. z)) \rightarrow \\ &((\lambda y. y y) (\lambda z. z)) \rightarrow \\ &(\lambda z. z)(\lambda z. z) \rightarrow \\ &(\lambda z. z) \end{aligned}$$

Note that each time we were evaluating the left-most application. If it seems like we skipped a step it was just because a term was already reduced as far as it could so we moved on.

3 Eager Evaluation

3.1 The idea (slide 31)

The idea of eager evaluation is what we have seen, it's what you know. Here we are just stating it in a formal language. We start out with looking at the top most application. So with the $f x y z$ example. The top most application is the z being applied to the $(f x y)$. First you evaluate the $(f x y)$. This is done recursively, so it boils down to first evaluate the f , and then the f applied to x and then the $(f x)$ applied to y , etc. Then evaluate the z by itself. And then reduce the whole term.

So, to evaluate this function you begin by evaluating f separately from x , then do a reduction between the two. Then you evaluate the term you just reduced the $(f x)$ and then evaluate the y and then reduce. This means at each iterations you're doing an evaluation, evaluation, reduction.

3.2 Example (slide 32)

Let's try evaluating $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$ with eager evaluation. This is the same example from before. There is only one function, the $(\lambda z. (\lambda x. x))$, and one argument, the $((\lambda y. y y) (\lambda y. y y))$. So we evaluate function. That's done. So we evaluate the argument next. But we already discussed this. It infinitely evaluates to itself, so, we never end up getting to the reduction stage. This means that eager evaluation cannot reduce this term. Here, having an argument that fails to terminate means that our overall evaluation will fail to terminate.

3.3 Example 2 (slide 45)

Now, let's look at $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$.

Our function is $(\lambda x. x x)$. Our argument is $((\lambda y. y y) (\lambda z. z))$.

With eager evaluation, first we evaluate our function. It's already reduced so our next step here is to evaluate the first argument.

We take our argument, $((\lambda y. y y) (\lambda z. z))$ and we start with the substitution. So we get: $((\lambda z. z)(\lambda z. z))$

Now we apply the $(\lambda z. z)$ to $(\lambda z. z)$. And doing that gets us: $(\lambda z. z)$

We can't reduce this anymore so now we can do the reduction between the function and the argument, $(\lambda x. x x)(\lambda z. z)$.

So, doing the reduction gets us the term $((\lambda z. z)(\lambda z. z))$. Since both parts are already reduced to their simplest forms we go straight to the substitution part and end up with $(\lambda z. z)$.

We can't do any more with this, so we're done. $(\lambda z. z)$ is our final answer.

Here are all the steps:

$$\begin{aligned} &(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \rightarrow \\ &(\lambda x. x x)((\lambda z. z)(\lambda z. z)) \rightarrow \\ &(\lambda x. x x)(\lambda z. z) \rightarrow \\ &(\lambda z. z)(\lambda z. z) \rightarrow \\ &(\lambda z. z) \end{aligned}$$

Note that each time we were following the evaluate, evaluate, reduce scheme. If it seems like we skipped a step it was just because a term was already reduced as far as it could so we moved on. You can easily go back through the example and find where we "skipped" a part because something was already reduced.

4 Data Structures

4.1 What we want next (new slide set: slide 2)

So, we want to be able to model the kinds of data structures and types that we have seen in OCaml. So want to be able to do represent things like booleans, pairs, and lists. So let's say we have an enumeration type τ with n constructors, C_1, C_2, \dots, C_n . The n things are just the different things the type can be. So if the type was the days of the week, $n = 7$.

To represent a constructor C_i with λ calculus, it would look like: $\lambda x_1. \lambda x_2 \dots \lambda x_n. x_i$. Think of this as a match statement of sorts. We want to return the i^{th} value, so we apply the correct case for the i^{th} constructor. In other words, you're being given n different things. Each thing is labeled 1 through n . When we search for a specific thing, we simply return it.

4.2 Simple cases: Booleans (slide 3)

Let's try to represent booleans. So, when you have the data type of boolean, there are two possibilities for what the boolean can be, true or false (which means $n = 2$ in this case). Like described above, we have that a our value $C_i = \lambda x_1. \lambda x_2. x_i$. That is our C_1 we can say is true and C_2 is false. So to represent true we have $C_1 = \lambda x_1. \lambda x_2. x_1$. And similarly for false, we have $C_2 = \lambda x_1. \lambda x_2. x_2$.

You can take this and apply it to `if_then_else` statements, we can describe x_1 as the then case and x_2 as the else case. Then we can say that C_1 will return the then case while C_2 returns the else case. We never have to look at the case we don't return.

Side note on notation, we will write $\lambda x_1 x_2 \dots x_n. e$ as the shorthand for $\lambda x_1. \lambda x_2 \dots \lambda x_n. e$. Also, we will denote $(\dots(e_1 e_2)\dots e_n)$ as $e_1 e_2 \dots e_n$.

4.3 How to use this (slide 4)

Like we said earlier, to do something with this, we are going to use match statements, or at least something similar. We were saying that we want to match each C_i with it's x_i . That is:

match c with $C_1 \rightarrow x_1 \mid C_2 \rightarrow x_2 \mid \dots \mid C_n \rightarrow x_n$

Well, with λ notation, we can rewrite this as:

$\lambda x_1 \dots x_n c. c x_1 \dots x_n$

This line is saying, we have all of our different cases, and our particular constructor c , then on the other side of the decimal, you let your constructor tell you which one of those cases it meant. That is, the match statement is saying, give me a case for each constructor. Then give me the constructor. With that information, the constructor itself will tell you with case is the right one.

4.4 If then else (slide 5)

So, for example, with the `if.then_else` statement, we give the then case as x_1 and x_2 as the else case, then give the boolean you got from the if part and that will return the then or else case accordingly.

If we had the statement `if b then c else d` then we can translate it to:

`if.then_else b x_1 x_2 = b x_1 x_2`

This is equivalent to:

$\lambda b x_1 x_2. b x_1 x_2$

Through match statements, you get that b with true goes to x_1 and false goes to x_2 . That looks like:

`if.then_else b x_1 x_2 = match b with True \rightarrow $x_1 \mid$ False \rightarrow x_2 =
 $(\lambda x_1 x_2 b. b x_1 x_2) x_1 x_2 b$`

This last line is basically saying that you have your parts the result from the *if* case, the b , the *then* case and the *else* case, and we are going to take those parts and apply it to our match statement and see what it returns. It will either return the *then* case or the *else* case.

4.5 Not (slide 6)

To do *not*, we do a simple case analysis with the `if.then_else` statement. We basically check to see if b was originally true, if it was, like normal, we go to the *then* case, but the *then* case returns false. And similarly with the *else* case, we get there if b was false, so we return the opposite value, which is true.

We, used $b c d$ as part of the example above, and we will reuse it. Our `if.then_else` will look like $(\lambda b c d. b c d)$. The arguments we are passing in are, b (the boolean from the *if* part), $(\lambda x y. y)$ to represent false for the *then* case then then $(\lambda x y. x)$ to represent true for the *else* case.

With that, we can say that:

`not \equiv $\lambda b. b (\lambda x y. y) (\lambda x y. x)$`