Introduction to the Lambda Calculus

See Chapters 3, 4, and 5 for relevant material.

The abstract syntax for the pure lambda calculus is dirt simple:

$$x \in \mathsf{Var}$$
$$e \in \mathsf{Exp} ::= x \mid \lambda x.e \mid e_1\ e_2$$

The (small-step) operational semantics is also dirt simple:

$$(\lambda x.e_1)\ e_2 \rightarrow e_1[e_2/x]$$

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

where $e_1[e_2/x]$ denotes capture-avoiding substitution of the term e_2 for the free occurrences of x within the term e_1. This is formalized as follows:

$$\begin{array}{rcll} x[e_2/x] & = & e_2 & \\ y[e_2/x] & = & y & (y! = x) \\ (\lambda x.e)[e_2/x] & = & \lambda x.e & \\ (\lambda y.e)[e_2/x] & = & \lambda y.(e[e_2/x]) & (y \notin \mathsf{FV}(e_2)) \\ (e\ e')[e_2/x] & = & (e[e_2/x])\ (e'[e_2/x]) & \end{array}$$

Note that $\mathsf{FV}(e)$ stands for the set of free variables in an expression and is defined as:

$$\begin{array}{rcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(e_1\ e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\ \mathsf{FV}(\lambda x.e) & = & \mathsf{FV}(e) \setminus \{x\} \end{array}$$

These rules are meant to capture a high-level model of functions and function calls. Intuitively, we substitute the actual parameter (e_2) for the formal parameter (x) within the body of the function and then run the function.

First, a basic principle of static scoping is that variable names shouldn't matter as far as the semantics of a program are concerned. Both $\lambda x.x$ and $\lambda y.y$ behave the same (they are terms that represent the identity function) so we would like to treat them as equivalent.

This gives rise to the notion of α-equivalence and α-equivalent classes of terms. We can define:

$$x \stackrel{\alpha}{=} x$$

$$\frac{e_1 \stackrel{\alpha}{=} e_1' \qquad e_2 \stackrel{\alpha}{=} e_2'}{e_1\ e_2 \stackrel{\alpha}{=} e_1'\ e_2'}$$

$$\frac{e_1[z/x] \stackrel{\alpha}{=} e_2[z/y]}{\lambda x.e_1 \stackrel{\alpha}{=} \lambda y.e_2} \qquad (z \notin \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2))$$

and say that two terms e_1 and e_2 are α-equivalent if $e_1 \stackrel{\alpha}{=} e_2$. We write $[e]$ for the set of terms that are alpha-equivalent to e (i.e., $\{e' \mid e' \stackrel{\alpha}{=} e\}$).

When we define the semantics of programs, we're going to technically be using alpha-equivalence classes over terms instead of the terms themselves. This means that, by default, alpha-equivalent terms will end up having the same meaning. It also means that we can use the "bound variable convention":

> When you are writing down a lambda-calculus expression, you can always pick a term in the α-equivalence class that avoids naming conflicts with bound variables.

For instance, in the substitution rule above, we have a clause:

$$(\lambda y.e)[e2/x] = \lambda y.(e[e2/x]) \qquad (y \notin \mathsf{FV}(e_2))$$

Suppose $e_2 = y$. Then we can't do the substitution because of the side condition. However, we can first pick out the α-equivalent term $(\lambda z.e[z/y])$ where z is a fresh variable and then do the substitution.

A few technicalities regarding scope and variables: First, note that when we have:

$$(\lambda x.e)[e2/x]$$

we don't push the substitution in. The reason is that the inner binding for x shadows the outer binding that we were substituting. For instance, if we start with:

$$\lambda x.(\lambda x.x)$$

and apply that to e_2, we'll end up trying to reduce $(\lambda x.x)[e_2/x]$ which should evaluate to just $\lambda x.x$.

Encodings in the Pure Lambda Calculus

An easy encoding is:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \stackrel{\Delta}{=} (\lambda x.e_2)e_1$$

Note that $(\lambda x.e_2)\ e_1 \rightarrow e_2[e_1/x]$ which is the intended behavior of a let expression.

Another easy encoding is a multi-argument function.

$$\lambda[x_1, x_2, \ldots, x_n].e \stackrel{\Delta}{=} \lambda x_1.\lambda x_2. \ldots \lambda x_n.e$$

To encode a data value, we think about how the data values are used instead of how they are built. For instance, booleans are used in conditionals. We want to define some functions for true, false, and if such that

$\mathsf{if}\ e\ e_1\ e_2 \rightarrow e_1$	when e evaluates to true
$\mathsf{if}\ e\ e_1\ e_2 \rightarrow e_2$	when e evaluates to false

We can simplify the problem by defining if as:

$$\mathsf{if}\ e\ e_1\ e_2 \stackrel{\Delta}{=} e\ e_1\ e_2$$

That is:

$$\mathsf{if} \stackrel{\Delta}{=} \lambda\mathsf{test}.\lambda\mathsf{then}.\lambda\mathsf{else}.\mathsf{test\ then\ else}$$

Then we can take true and false to be:

$$\mathsf{true} \triangleq \lambda \mathsf{then}.\lambda \mathsf{else}.\mathsf{then}$$

$$\mathsf{false} \triangleq \lambda \mathsf{then}.\lambda \mathsf{else}.\mathsf{else}$$

Then you can verify that:

$$\mathsf{if\ true}\ e_1\ e_2 \rightarrow^* e_1$$

$$\mathsf{if\ false}\ e_1\ e_2 \rightarrow^* e_2$$

We can define numbers as follows:

$$\begin{array}{rcl} 0 & \triangleq & \lambda f.\lambda x.x \\ 1 & \triangleq & \lambda f.\lambda x.(f\ x) \\ 2 & \triangleq & \lambda f.\lambda x.(f\ (f\ x)) \\ 3 & \triangleq & \lambda f.\lambda x.(f\ (f\ (f\ x))) \\ & \vdots & \\ n & \triangleq & \lambda f.\lambda x.f^n\ x \end{array}$$

Note that a number n takes a function f and some argument x and applies the function to the argument n times.

$$\begin{array}{rcl} \mathsf{inc} & \triangleq & \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \\ \mathsf{plus} & \triangleq & \lambda n.\lambda m.n\ inc\ m \\ \mathsf{times} & \triangleq & \lambda n.\lambda m.n\ (plus\ m)\ 0 \end{array}$$

We can define pairs (2-tuples) as:

$$\mathsf{pair} \triangleq \lambda x.\lambda y.\lambda f.f\ x\ y$$

So, $\mathsf{pair}\ 0\ 1 \rightarrow^* \lambda f.f\ 0\ 1$.

$$\mathsf{first} \triangleq \lambda p.(p\ (\lambda x.\lambda y.x))\mathsf{second} \triangleq \lambda p.(p\ (\lambda x.\lambda y.y))$$

So,

$$\begin{array}{rcl} \mathsf{first}\ (\mathsf{pair}\ 0\ 1) & \rightarrow^* & 0 \\ \mathsf{second}\ (\mathsf{pair}\ 0\ 1) & \rightarrow^* & 1 \end{array}$$

You can define lists, trees, predecessor, subtraction, tests for zero, and just about anything else you can imagine in the pure lambda calculus.

You can also encode loops. Here's the analog of "$\mathsf{while\ true\ do\ skip}$":

$$\mathsf{forever} \triangleq (\lambda x.x\ x)(\lambda x.x\ x)$$

Note that $\mathsf{forever} \rightarrow \mathsf{forever} \rightarrow \mathsf{forever} \rightarrow \ldots$.

A more interesting combinator is Y:

$$\mathsf{Y} \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Note that for any function f:

$$\mathsf{Y}\ f \rightarrow f\ (Y\ f)$$

So, for instance, we can define:

$$\mathsf{factbody} \stackrel{\Delta}{=} (\lambda f.\lambda x.\mathsf{if}\ (\mathsf{eq}\ x\ 0)\ 1\ (\mathsf{times}\ x\ (f\ (\mathsf{dec}\ x\ 1))))$$

$$\mathsf{fact} \stackrel{\Delta}{=} \mathsf{Y}\ \mathsf{factbody}$$

Then note that

$$\begin{array}{rcl} \mathsf{Y\ factbody} & \rightarrow & \mathsf{factbody\ (Y\ factbody)} = \mathsf{factbody\ fact} \\ & \rightarrow & \lambda x.\ \mathsf{if}\ (\mathsf{eq}\ x\ 0)\ 1\ (\mathsf{times}\ x\ (\mathsf{fact}\ (\mathsf{dec}\ x\ 1))) \end{array}$$

So, if you have an ML function:

```
let fun f(x) = e
in
   e'
end
```

we can encode this as:

$$(\lambda f.e')\ (\mathsf{Y}\ (\lambda f.\lambda x.e))$$

Consider:

$$\begin{array}{c} \mathsf{inc}\ 1 = (\lambda n.\lambda g.\lambda x.g\ (n\ g\ x))\ (\lambda f.\lambda x.(f\ x)) \\ \rightarrow \lambda g.\lambda x.g\ ((\lambda f.\lambda x.(f\ x))\ g\ x) \end{array}$$

and then we're stuck. So the result of inc 1 is

$$\lambda g.\lambda x.g\ ((\lambda f.\lambda x.(f\ x))\ g\ x)$$

which is not 2:

$$\lambda g.\lambda x.g\ (g\ x)$$

But I claim that these two functions behave the same in all contexts. That is, (inc 1) and 2, when given the same arguments, produce equivalent results. So we're morally justified in saying that (inc 1) is equal to 2, even though the evaluation rules don't justify this directly.

The problem is that, for any given function, there are lots and lots of ways to write down that function as a lambda term. It would be nice, when we're reasoning about programs, if we could somehow collapse all of the functions down so that, if they behave the same, then we could get them to line up syntactically. This would allow us to mechanize the comparison of two programs to see if they are equal.

Let me define the following inference rules:

$$\frac{e \stackrel{\alpha}{=} e'}{e \stackrel{\beta}{=} e'}$$

$$\frac{e_1 \stackrel{\beta}{=} e_2}{e_2 \stackrel{\beta}{=} e_1}$$

$$\frac{e_1 \stackrel{\beta}{=} e_2 \qquad e_2 \stackrel{\beta}{=} e_3}{e_1 \stackrel{\beta}{=} e_3}$$

The first rule says that if two expressions are α-equivalent, then we also consider them to be semantically (β) equivalent. The second and third rules say that $\stackrel{\beta}{=}$ is an equivalence relation in that it is symmetric and transitive. (α-equivalence gives us reflexivity.)

$$(\lambda x.e_1)\ e_2 \stackrel{\beta}{=} e_1[e_2/x]$$

This rule says that a function applied to an expression can be considered equivalent to the substitution of that expression for the formal parameter in the body of the function – just like the evaluation rule.

$$\frac{e_1 \stackrel{\beta}{=} e_1' \qquad e_2 \stackrel{\beta}{=} e_2'}{e_1\ e_2 \stackrel{\beta}{=} e_1'\ e_2'}$$

$$\frac{e \stackrel{\beta}{=} e'}{\lambda x.e \stackrel{\beta}{=} \lambda x.e'}$$

These two rules say that we can replace beta-equivalent expressions within sub-expressions and still get out equivalent expressions.

Finally, there's one more rule that's possible:

$$\lambda x.(e\ x) \stackrel{\beta}{=} e \qquad (x \notin \mathsf{FV}(e))$$

This is called the "η- (eta-)rule". It reflects the fact that if x is not in e, then $\lambda x.(e\ x)\ e' \stackrel{\beta}{=} e\ e'$. That is, when we apply $\lambda x.(e\ x)$ to an argument, and we apply e to that same argument, we get out the same result.

Neat fact: if $\vdash e_1 \stackrel{\beta}{=} e_2$, then for all e, $e_1 e \stackrel{\beta}{=} e_2\ e$. This just says that $\stackrel{\beta}{=}$ captures our notion of equivalence.

Unfortunately, we can't turn this into a decision procedure for reasoning about the equivalence of programs since, as we've seen, the lambda calculus is Turing complete. Still, β-equivalence lets us easily reason about expressions in an algebraic style.

For instance, we can now prove that $(\mathsf{inc}\ 1) \stackrel{\beta}{=} 2$ (exercise) using these rules.

It's much, much easier to use these rules than to construct some denotational semantics and try to prove that two expressions are equivalent. In fact, there's a lot of literature in the functional world on writing naïve, but easy to understand programs as your specification. Then, you derive an efficient implementation by using these algebraic laws to get something that is provably equivalent to your specification.

Call-by-Value vs. Call-by-Name

Note that the pure lambda calculus does not evaluate arguments before calling a function:

$$(\lambda x.e_1)\ e_2 \rightarrow e_1[e_2/x]$$

This was extremely useful for implementing if as a function:

$$\mathsf{if} \stackrel{\Delta}{=} \lambda b.\lambda t.\lambda e.b\ t\ e$$

because when we write "$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$" we do *not* want to evaluate e_1 or e_2 until we've evaluated e. Otherwise, in a recursive function, we'd loop forever.

We say that the pure lambda calculus is call-by-name. The terminology comes from Algol and related languages.

In contrast, ML, C, C++, Java, etc. are call-by-value languages. In a call-by-value language, the evaluation rules are roughly:

$$(\lambda x.e_1)v \rightarrow e_1[v/x]$$

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v\ e_2 \rightarrow v\ e_2'}$$

where v ranges over values. In the case of the lambda calculus, the only values we have are functions ($\lambda x.e$). Note that the rules effectively force you to evaluate in a left-to-right, innermost-to-outermost fashion, and force you to evaluate the argument before you call the function.

Call-by-value (CBV) doesn't work well when we decide to define something like if, because it eagerly evaluates the "then" and "else" clauses. For instance, if you define in ML:

```
val True = fn x => fn y => x
val False = fn x => fn y => y
val If = fn x => fn y => fn z => x y z
```

and then write:

```
If True (print "Hello") (print "Goodbye")
```

then you'll get both "Hello" and "Goodbye" printed. In a call-by-name language, you'd only get "Hello" which is what we normally intend.

Call-by-name (CBN) isn't always better though. Consider:

$$\mathsf{double} = \mathsf{fn}\ x\ \ =>\ \ x + x$$

If we did something like:

$$\mathsf{double}\ (\mathsf{big\text{-}computation\text{-}that\text{-}takes\text{-}2\text{-}weeks})$$

then in CBN, this would reduce to:

$$(\mathsf{big\text{-}computation\text{-}that\text{-}takes\text{-}2\text{-}weeks}) + (\mathsf{big\text{-}computation\text{-}that\text{-}takes\text{-}2\text{-}weeks})$$

and you'd end up computing for 4 weeks (plus a little `:-)`). In call-by-value, you'd first reduce (big-computation-that-takes-2 to its value (say 3) and then call couble:

$$\mathsf{double}\ (\mathsf{big\text{-}computation\text{-}that\text{-}takes\text{-}2\text{-}weeks}) \rightarrow \mathsf{double}\ 3 \rightarrow 3 + 3$$

which only takes 2 weeks. So, in principle, if we ignore efficiency, CBN is better because we don't compute something if we don't need it. But in practice, for most things, CBV is better because we only compute things at most once.

Another option is call-by-need (a.k.a. Lazy) evaluation. This requires a more complicated model so we won't formalize it here. But the basic idea is to insert a level of indirection and a flag and only evaluate things once. Here's how we might encode a lazy computation in ML:

```
datatype 'a E = Value of 'a | Computation of unit -> 'a
type 'a thunk = ('a E) ref

fun read_thunk r =
   case (!r) of
     Value v => v
   | Computation f =>
     let val v = f()
     in
         r := (Value v);
         v
     end

fun new_thunk f = ref (Computation f)
```

I can simulate something like "if e_1 then e_2 else e_3" by writing:

```
fun If (e1:'a thunk->'a thunk->'a thunk) (e2:'a thunk) (e3: 'a thunk) =
     e1 e2 e3

fun True (x:'a thunk) (y:'a thunk) = x
fun False (x:'a thunk) (y:'a thunk) = y
```

Now, I could write:

```
read_thunk(If True (new_thunk (fn () => print "Hello"))
           (new_thunk (fn () => print "Goodbye")))
```

This will evaluate to unit and print "Hello". If I write:

```
fun double (x: int thunk) =
    (read_thunk x) + (read_thunk x)
```

then I can write:

```
double (new_thunk (fn () => big-computation-that-takes-2-weeks))
```

and I'll only do the computation once. But note that I could also write:

```
fun forget (x: int thunk) = 3
```

and

```
forget (new_thunk (fn () => big-computation-that-takes-2-weeks))
```

will run in constant time (i.e., won't take 2 weeks).