Programming Languages

Lectures by Yoni Zohar
Summary by Ari Feiglin (`ari.feiglin@gmail.com`)

Contents

1 Untyped Lambda Calculus

Lambda calculus is a way of formalizing computations, it generalizes the concept of functions. A function in lambda calculus has the form $\lambda x.t$ and should be thought of a function $x \mapsto t(x)$, in a language like OCaml, this corresponds to a function definition of the form $\texttt{fun}\, x \to t$. It is built from syntax, and we then utilize semantics to give this syntax meaning.

1.0.1 Definition

Let V be an infinite set of variable symbols, then terms in lambda calculus are constructed recursively as follows:

(1) every variable is an term,

(2) if $x \in V$ is a variable and t is an term, then $\lambda x.t$ is an term,

(3) if t_1 and t_2 are terms, then so is $t_1 t_2$.

Notice that lambda calculus terms have the *unique reconstruction property*: every term t has one of the above forms, and such a form is *unique*. We can then construct functions on lambda terms via term recursion, as given by the following examples.

1.0.2 Definition

Given an term of the form $\lambda x.t$, every instance of x in the term t is called **bound**, and all other instances are **free**. Formally we can define the set of free variables in an term recursively as follows:

(1) for an term of the form x for a variable x, $\mathit{var}(x) = \{x\}$, $\mathit{free}(x) = \{x\}$, $\mathit{bnd}(x) = \varnothing$,

(2) for an term of the form $\lambda x.t$, $\mathit{var}(\lambda x.t) = \mathit{var}(t) \cup \{x\}$, $\mathit{free}(\lambda x.t) = \mathit{free}(t) \setminus \{x\}$, and $\mathit{bnd}(\lambda x.t) = \mathit{bnd}(t) \cup \{x\}$,

(3) for an term of the form $t_1 t_2$, $\mathit{var}(t_1 t_2) = \mathit{var}(t_1) \cup \mathit{var}(t_2)$, $\mathit{free}(t_1 t_2) = \mathit{free}(t_1) \cup \mathit{free}(t_2)$ and $\mathit{bnd}(t_1 t_2) = \mathit{bnd}(t_1) \cup \mathit{bnd}(t_2)$.

Alternatively, a **bound occurrence** of a variable x in t is an occurrence which occurs in t' where $\lambda x.t'$ is a subterm of t. A **free occurrence** is an occurrence which is not bound. Then $\mathit{free}(t)$ is the set of all variables which occur free in t, bndt is the set of all variables which occur bound in t.

So for example, let $t = (\lambda x.\lambda y.x)x\,z$, then $\mathit{var}(t) = \{x, y, z\}$, $\mathit{free}(t) = \{x, z\}$, $\mathit{bnd}(t) = \{x, y\}$. Here the x and y in $\lambda x.\lambda y.x$ are bound occurrences, and the x and z following it (in $x\,z$) are free. Notice that always $\mathit{var}(t) = \mathit{free}(t) \cup \mathit{bnd}(t)$, but as the above example shows, these two sets are not always disjoint. A proof of this union is done via term induction: prove it for $t = x$, then for $t = \lambda x.t'$, then finally for $t = t_1 t_2$.

(1) for $t = x$, $\mathit{var}(t) = \{x\}$, $\mathit{free}(t) = \{x\}$, and $\mathit{bnd}(t) = \varnothing$, so the union holds.

(2) for $t = \lambda x.t'$, $\mathit{var}(t) = \mathit{var}(t') \cup \{x\}$ which by induction is equal to $\mathit{free}(t') \cup \mathit{bnd}(t') \cup \{x\}$. Now $\mathit{free}(t) = \mathit{free}(t') \setminus \{x\}$, $\mathit{bnd}(t) = \mathit{bnd}(t') \cup \{x\}$ and so we see that $\mathit{free}(t) \cup \mathit{bnd}(t) = \mathit{var}(t)$ as required.

(3) for $t = t_1 t_2$, $\mathit{var}(t) = \mathit{var}(t_1) \cup \mathit{var}(t_2)$ which by induction is $\mathit{free}(t_1) \cup \mathit{free}(t_2) \cup \mathit{bnd}(t_1) \cup \mathit{bnd}(t_2) = \mathit{free}(t) \cup \mathit{bnd}(t)$.

1.0.3 Definition

An term without free variables is called a **combinator**. The **identity combinator** is the combinator $\mathsf{id} = \lambda x.x$.

Suppose we'd like to take a term t and substitute x with another term t'. For example, suppose t' is the variable z, then $\lambda\texttt{y.x}$ should become $\lambda\texttt{y.z}$. But then what should $\lambda\texttt{x.x}$ become? Surely not $\lambda\texttt{x.z}$, as that alters the entire interpretation of the function. So variables should be substituted only at free occurrences. But what about if t' were x and t was $\lambda\texttt{x.y}$, then substituting at y gives $\lambda\texttt{x.x}$, which once again changes the meaning of the function. So we should only substitute at free occurrences, if the λ-variable is not free in the term being substituted.

1.0.4 Definition

Let t, t' be terms and x a variable. Then $t[x \mapsto t']$ is the term obtained by substituting x with t' according to the following rules:

(1) $x[x \mapsto t'] = t'$,

(2) $y[x \mapsto t'] = y$ if y is a variable distinct from x,

(3) $(\lambda x.t)[x \mapsto t'] = \lambda x.t$,

(4) $(\lambda y.t)[x \mapsto t'] = \lambda y.(t[x \mapsto t'])$ if $y \neq x$ and $y \notin \mathit{free}(t')$,

(5) $(t_1\, t_2)[x \mapsto t'] = t_1[x \mapsto t']\, t_2[x \mapsto t']$.

But then what would the substitution $(\lambda y.x\, y)[x \mapsto y\, z]$ look like? Well y is free in the substituted term, so it doesn't match any of the above conditions. In such a case we take upon ourselves the following convention:

Convention

Terms that differ only in the named of bound variables are equivalent.

This means that we can view $\lambda y.x\, y$ as $\lambda w.x\, w$ and so the substitution becomes $\lambda w.y\, z\, w$.

1.0.5 Definition

A term of the form $(\lambda x.t)t'$ is called a **redex**. A term of the form $\lambda x.t$ is called a **abstraction**. We define the β **reduction** on terms which maps redexes to terms by $(\lambda x.t)t' \xrightarrow{\beta} t[x \mapsto t']$ where $t[x \mapsto t']$ is the term obtained by substituting t' at all the free occurrences of x.

For example, $(\lambda x.x)y \to y$, and

$$\big(\lambda x.(\lambda x.x)x\big)(u\, r) \to (\lambda x.x)(u\, r) = u\, r$$

When performing a β-reduction, we need to consider the order with which we perform the reduction. There are 4 ways:

(1) *Full β-reduction*, in which any redex can be reduced at any time. So at each step, we can arbitrarily choose a redex and reduce it. For example, take

$$\texttt{(}\lambda\texttt{x.x) ((}\lambda\texttt{x.x) (}\lambda\texttt{z.(}\lambda\texttt{x.x) z))}$$

which is just $\mathsf{id}(\mathsf{id}(\lambda z.\mathsf{id} z))$. This term contains three redexes:

$$\underline{\mathsf{id}(\mathsf{id}(\lambda z.\mathsf{id}\ z))},\quad \mathsf{id}(\underline{\mathsf{id}(\lambda z.\mathsf{id}\ z)}),\quad \mathsf{id}(\mathsf{id}(\lambda z.\underline{\mathsf{id}\ z}))$$

So we can choose for example to begin from the innermost redex and move outward:

$$\begin{aligned} & \mathsf{id}(\mathsf{id}(\lambda z.\underline{\mathsf{id} z})) \\ \to\ & \mathsf{id}(\underline{\mathsf{id}(\lambda z.z)}) \\ \to\ & \underline{\mathsf{id}(\lambda z.z)} \\ \to\ & \lambda z.z \end{aligned}$$

which cannot be reduced any more.

(2) *Normal order*, in which the leftmost outermost redex is reduced first. So using the same example as above:

$$\begin{aligned} & \underline{\mathsf{id}(\mathsf{id}(\lambda z.\mathsf{id} z))} \\ \to\ & \underline{\mathsf{id}(\lambda z.\mathsf{id} z)} \\ \to\ & \lambda z.\underline{\mathsf{id} z} \\ \to\ & \lambda z.z \end{aligned}$$

(3) *Call-by-name*, which is similar to normal order but it performs no reductions inside abstractions. Using the same example:

$$\begin{array}{rl} & \underline{\mathsf{id}(\mathsf{id}(\lambda \mathsf{z}.\mathsf{idz}))} \\ \to & \underline{\mathsf{id}(\lambda \mathsf{z}.\mathsf{idz})} \\ \to & \lambda \mathsf{z}.\mathsf{idz} \end{array}$$

(4) *Call-by-value*, which is the most commonly used in programming languages, like call-by-name, but a redex is reduced only when its right-hand side has already been reduced to a *value* (a term which cannot be reduced further, in this lambda calculus these are only abstractions).

$$\begin{array}{rl} & \mathsf{id}(\underline{\mathsf{id}(\lambda \mathsf{z}.\mathsf{idz})}) \\ \to & \underline{\mathsf{id}(\lambda \mathsf{z}.\mathsf{idz})} \\ \to & \lambda \mathsf{z}.\mathsf{idz} \end{array}$$

In this course we use call-by-value, since it is the most commonly used evaluation strategy.

Notice that in lambda calculus, all functions accept a single parameter as input. As in OCaml, to write a function which accepts multiple functions, we write one which accepts a single input and returns a function which also accepts a single input. So for example $f = \lambda x.\lambda y.x$ can then be called like $f\,u\,r$ and will return u after two β-reductions.

We now define booleans in lambda calculus (called Church booleans):

$$\mathsf{tru} = \lambda t.\lambda f.t, \qquad \mathsf{fls} = \lambda t.\lambda f.f$$

So tru accepts two arguments and returns the first, fls accepts two and returns the second. We now define

$$\mathtt{test} = \lambda b.\lambda m.\lambda n.\, b\, m\, n$$

So `test` accepts three arguments, the first b is a boolean (either tru or fls), and it applies it to the other two arguments. So for example

$$\mathtt{test}\,\mathsf{tru}\,v\,w = (\lambda b.\lambda m.\lambda n.\, b\,m\,n)\mathsf{tru}\,v\,w \to (\lambda m.\lambda n.\mathsf{tru}\,m\,n)v\,w \to (\lambda n.\mathsf{tru}\,v\,n)w \to \mathsf{tru}\,v\,w \to v$$

This doesn't do much, it just returns the first argument (after the boolean) if the boolean is true, and the second if it is false.

We can define a more interesting combinator

$$\mathsf{and} = \lambda b.\lambda c.b\,c\,\mathsf{fls}$$

Here b, c are booleans. Then if b is tru, $\mathtt{and}\,b\,c \to c$ after a β-reduction, and otherwise it will reduce to c. So if c is false, then $\mathtt{and}\,b\,c \to c = \mathsf{fls}$ and if c is true then it reduces to $c = \mathsf{tru}$, and if b is false then $\mathtt{and}\,b\,c \to b\,c\,\mathsf{fls} \to \mathsf{fls}$. So `and` functions as one would expect it to.

Utilizing booleans, we can encode pairs of values as terms:

$$\begin{aligned} \mathtt{pair} &= \lambda \mathsf{f}.\lambda \mathsf{s}.\lambda \mathsf{b}.\mathsf{b}\,\mathsf{f}\,\mathsf{s} \\ \mathtt{fst} &= \lambda \mathsf{p}.\mathsf{p}\,\mathsf{tru} \\ \mathtt{snd} &= \lambda \mathsf{p}.\mathsf{p}\,\mathsf{fls} \end{aligned}$$

Notice then that

$$\begin{array}{rll} & \mathtt{fst(pair\ v\ w)} & \\ = & \mathtt{fst(}\underline{\mathtt{(\lambda f.\lambda s.\lambda b.b\ f\ s)\ v}}\mathtt{\ w)} & \text{by definition} \\ \to & \mathtt{fst(}\underline{\mathtt{(\lambda s.\lambda b.b\ v\ s)\ w}}\mathtt{)} & \beta\text{-reduction on underlined redex} \\ \to & \mathtt{fst(}\underline{\mathtt{\lambda b.b\ v\ w}}\mathtt{)} & \beta\text{-reduction on underlined redex} \\ = & \underline{\mathtt{(\lambda p.p\ tru)(\lambda b.b\ v\ w)}} & \text{by definition} \\ \to & \underline{\mathtt{(\lambda b.b\ v\ w)tru}} & \beta\text{-reduction on underlined redex} \\ \to & \mathtt{tru\ v\ w} & \beta\text{-reduction on underlined redex} \\ \to & \mathtt{v} & \text{by definition of tru} \end{array}$$

In a similar manner we can show that `snd(pair v w)`$\to$`w`.

We now demonstrate how we can represent numbers in lambda calculus, via Church numerals:

$$\begin{aligned} \mathtt{c}_0 &= \mathtt{\lambda s.\lambda z.z} \\ \mathtt{c}_1 &= \mathtt{\lambda s.\lambda z.s\ z} \\ \mathtt{c}_2 &= \mathtt{\lambda s.\lambda z.s(s\ z)} \\ \mathtt{c}_3 &= \mathtt{\lambda s.\lambda z.s(s(s\ z))} \end{aligned}$$

etc.

In general if we write $\texttt{s}^n\ \texttt{z}$ for $\texttt{s(s(}\cdots\texttt{s z}\cdots\texttt{))}$ (n times), then $\texttt{c}_n = \lambda\texttt{s}.\lambda\texttt{z}.\texttt{s}^n\ \texttt{z}$. So each number n is represented by the combinator $\texttt{c}_n$ which accepts `s,z` and applies `s` n times to `z`. Notice that $\texttt{c}_0 = \textsf{fls}$, which is reminiscent of the fact that false and zero mean the same thing in many compiled languages.

Let us define

$$\texttt{scc=}\ \lambda\texttt{n}.\lambda\texttt{s}.\lambda\texttt{z}.\texttt{s(n s z)}$$

We see then that

$$\texttt{scc}\ \texttt{c}_n\ \texttt{z s} = \lambda\texttt{s}.\lambda\texttt{z}.\texttt{s(}\texttt{c}_n\ \texttt{s z) s z} = \texttt{s(s}^n\ \texttt{z)} = \texttt{s}^{n+1}\ \texttt{z} = \texttt{c}_{n+1}\ \texttt{z s}$$

so $\texttt{scc}\ \texttt{c}_n$ and $\texttt{c}_{n+1}$ are the same.

Similarly we can define

$$\texttt{plus=}\ \lambda\texttt{n}.\lambda\texttt{m}.\lambda\texttt{s}.\lambda\texttt{z}.\texttt{m s (n s z)}$$

so that `plusn m s z` will apply `s n s z` m times, resulting in $\texttt{s}^m\texttt{s}^n\texttt{z} = \texttt{s}^{n+m}\texttt{z}$ as desired. Similarly we define

$$\texttt{times=}\ \lambda\texttt{n}.\lambda\texttt{m}.\lambda\texttt{s}.\lambda\texttt{z}.\texttt{m (plus n)}\ \texttt{c}_0$$

so that `timesn m s z` will apply `plusn` m times to $\texttt{c}_0$, resulting in $n + n + \cdots + n + 0 = n \cdot m$. In a similar vein, we can define $\texttt{pow} = \lambda\texttt{n}.\lambda\texttt{m}.\lambda\texttt{s}.\lambda\texttt{z}.\texttt{m (times n)}\ \texttt{c}_1$, so that $\texttt{pow}\ \texttt{c}_n\ \texttt{c}_m$ is equal to $\texttt{c}_{n^m}$.

To test if a numeral is zero, we'd like to find a functions `ss` and `zz` such that applying `ss` one or more times to `zz` yields false, while not applying it at all yields true. That way when we do $\texttt{c}_n$ `ss zz`, it will result in $\textsf{tru}$ only if `ss` was never applied, meaning $n = 0$. Necessarily then `zz` must be `tru`, and have `ss` be the function which maps every input to $\textsf{fls}$. So we define

$$\texttt{iszro=}\ \lambda\texttt{n}.\texttt{n}\ (\lambda\texttt{x}.\textsf{fls})\ \texttt{tru}$$

To define the predecessor combinator, we must be a bit more clever than with the successor. One implementation is

$$\begin{aligned} \texttt{zz} &= \texttt{pair}\ \texttt{c}_0\ \texttt{c}_0 \\ \texttt{ss} &= \lambda\texttt{p}.\texttt{pair(snd p)(plus}\ \underset{\smile}{1}\ \texttt{(snd p))} \\ \texttt{prd} &= \lambda\texttt{m}.\texttt{fst(m ss zz)} \end{aligned}$$

The idea here is that applying `ss` to a (n, m) will result in $(m, m + 1)$. So starting from $(0, 0)$, you get $(0, 1)$ then $(1, 2)$ then $(3, 2)$ and so on. In general $\texttt{ss}^n\texttt{z} = (n, n - 1)$ for $n \geq 1$ and so the predecessor is just the second value.

Using the predecessor combinator we can define a subtraction combinator similar to addition:

$$\texttt{sub=}\ \lambda\texttt{m}.\lambda\texttt{n}.\texttt{m prdn}$$

Notice though that `sub` cannot give negative numbers, after all we didn't define negative numbers, so if $n \leq m$ then $\texttt{c}_n\texttt{-}\texttt{c}_m$ is just $\texttt{c}_0$. Thus we can define

$$\begin{aligned} \texttt{leq} &= \lambda\texttt{m}.\lambda\texttt{n}.\texttt{iszro(sub m n)} \\ \texttt{equal} &= \lambda\texttt{m}.\lambda\texttt{n}.\texttt{and(leq n m) (leq m n)} \end{aligned}$$

1.0.6 Definition

A term without a redex is called a **normal form**. The normal form of a term t is the normal form obtained through β reduction. A term without a normal form is called **divergent**.

For example, the normal form of $(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x})\texttt{y}$ can be reduced to $\lambda\texttt{y}.\texttt{y}$ which is its normal form. One example of a divergent combinator is

$$\texttt{omega=}\ (\lambda\texttt{x}.\texttt{x x})(\lambda\texttt{x}.\texttt{x x})$$

Since a single β reduction gives you back `omega`, which gives what is essentially an infinite loop. We can also define the following combinator

$$\texttt{fix=}\ \lambda\texttt{f}.(\lambda\texttt{x}.\ \texttt{f}(\lambda\texttt{y}.\ \texttt{x x y}))\ (\lambda\texttt{x}.\ \texttt{f}(\lambda\texttt{y}.\ \texttt{x x y}))$$

Suppose we'd like to write a function to compute factorials, which can be written as

```
if n=0 then 1
else n * factorial(n-1)
```

The idea is to unravel the function definition, to get something of the form

```
if n=0 then 1
else n * (if n-1=0 then 1
          else (n-1) * (if n-2=0 then 1
                        else (n-2) * ...))
```

Using Church numerals, we get

```
test (equal n c₀)
    c₁
    times n (test (equal (prd n) c₀)
            c₁
            times (prd n) (test (equal (prd (prd n)) c₀)
                           c₁
                           times (prd (prd n)) (...)))
```

Then we define

$$\begin{aligned} \mathtt{g} &= \lambda\mathtt{fct}.\lambda\mathtt{n}.\ \mathtt{test\ (equal\ n\ c_0)\ c_1\ (times\ n\ (fct\ (prd\ n)))} \\ \mathtt{factorial} &= \mathtt{fix\ g} \end{aligned}$$

Let us give an example run of `factorial` $\mathtt{c_3}$:

$$\begin{array}{lll} & \mathtt{factorial\ c_3} & \\ = & \mathtt{fix\ g\ c_3} & \\ \to & \mathtt{h\ h\ c_3} & \text{where } \mathtt{h{=}}\lambda\mathtt{x.g(}\lambda\mathtt{y.x\ x\ y)} \\ \to & \mathtt{g\ fct\ c_3} & \text{where } \mathtt{fct{=}}\lambda\mathtt{y.\ h\ h\ y} \\ \to & \big(\lambda\mathtt{n.\ test(equal\ n\ c_0)\ c_1\ (times\ n\ (fct\ (prd\ n)))}\big)\mathtt{c_3} & \\ \to & \mathtt{test(equal\ c_3\ c_0)\ c_1\ (times\ c_3\ (fct\ (prd\ c_3)))} & \\ \to & \mathtt{times\ c_3\ (fct\ (prd\ c_3))} & \\ \to & \mathtt{times\ c_3\ (fct\ c_2)} & \\ \to & \mathtt{times\ c_3\ (h\ h\ c_2)} & \\ \to & \mathtt{times\ c_3\ (g\ fct\ c_2)} & \text{similar to how } \mathtt{h\ h\ c_3} \text{ can be reduced to } \mathtt{g\ fct\ c_3} \\ \to & \mathtt{times\ c_3\ (times\ c_2\ (g\ fct\ c_1))} & \text{by the same process that we did for } \mathtt{c_3} \\ \to & \mathtt{times\ c_3\ (times\ c_2\ (times\ c_1\ (g\ fct\ c_0)))} & \\ \to & \mathtt{times\ c_3\ (times\ c_2\ (times\ c_1\ (test\ (equal\ c_0\ c_0)\ c_1\ ...)))} & \\ \to & \mathtt{times\ c_3\ (times\ c_2\ (times\ c_1\ c_1))} & \\ \to & \mathtt{c_6} & \end{array}$$

Let us prove that this works. Suppose we have a recurrence `r=`λ`x.`$\langle$`code with r`$\rangle$, let us use the notation $\langle$`r c`$\rangle$ to mean that within the recurrence, `r` is called on the value `c`. Let us define `g=`λ`r.`λ`x.`$\langle$`code with r`$\rangle$, which is like `r` but it accepts the function it should run on. So if we were to define `r`, then `r` and `g r` would be functionally the same. We claim then that `r=fix g` is a term which is equivalent to `r` (does the same thing). Let us reduce it a bit on some term `c`

$$\begin{array}{lll} & \mathtt{r\ c} & \\ = & \mathtt{fix\ g\ c} & \\ \to & \mathtt{h\ h\ c} & \text{where } \mathtt{h{=}}\lambda\mathtt{x.g(}\lambda\mathtt{y.x\ x\ y)} \\ \to & \mathtt{g\ r'\ c} & \text{where } \mathtt{r'{=}}\lambda\mathtt{y.h\ h\ y} \end{array}$$

Now we claim that `g r' c` gives the same result as `r c`, which we will prove on the number of recursive calls that `r c` makes. If we were to reduce this one more time, we'd get $\langle$`code with r'`$\rangle$ `c`, but since `r` makes no recursive calls on the input `c`, this functions the same as $\langle$`code with r`$\rangle$ `c`, which is `r c`. Now, suppose that on the first recursive call, the program calls `r' c'`, meaning for `r` it would call `r c'`. Now `r' c' = h h c' = g r' c'`, and by our inductive hypothesis `g r' c' = r c'`, so the code performs the same.

We can also define the *Y-combinator*:

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

Which can similarly perform recursion. Like `fix`, it is a *fixed-point* combinator, which is a combinator fix such that $f(\mathsf{fix} f) = \mathsf{fix} f$. Indeed:

$$\begin{array}{lll} & \mathsf{Y}\ \mathtt{g} & \\ = & \mathtt{(}\lambda\mathtt{f.(}\lambda\mathtt{x.f(x\ x))(}\lambda\mathtt{x.f(x\ x)))\ g} & \text{by definition} \\ \to & \mathtt{(}\lambda\mathtt{x.g(x\ x))(}\lambda\mathtt{x.g(x\ x))} & \text{by } \beta\text{-reduction} \\ \to & \mathtt{g((}\lambda\mathtt{x.g(x\ x))\ (}\lambda\mathtt{x.g(x\ x)))} & \text{by } \beta\text{-reduction} \\ = & \mathtt{g(}\mathsf{Y}\ \mathtt{g)} & \text{by the second equality} \end{array}$$

Though the final equality is only true up to β-reduction, meaning that `Y g` and `g(Y g)` both reduce to a similar term, not to one another.