Lecture 13: Programming in Pure Lambda Calculus

CS4400 Programming Languages

Readings: [TAPL], Chapter 5

Multiple Arguments

So far, we have looked at lambda abstractions which only take a single argument. However, unary functions are only a small part of our experience with programming. We use functions with multiple arguments all the time. How do we pass more than one argument to a lambda?

One approach would be to extend the calculus with a notion of tuples. Perhaps throw in some pattern matching, for good measure:

$$(\lambda(x, y).\ x\ y)\ (a, b)$$

However, this means that we are abandoning the very minimal core lambda calculus with all its simplicity. And we don't have to! As we know well by now, applying an abstraction simply replaces its bound variable with the argument that it's applied to, as in this trivial example:

$$(\lambda x.x\ y)\ b \longrightarrow (x\ y)[x := b] = (b\ y)$$

What happens if the abstraction actually just returns another abstraction.

$$(\lambda x.\ (\lambda y.\ x\ y))\ b \longrightarrow (\lambda y.\ x\ y)[x := b] = (\lambda y.\ b\ y)$$

Since neither of the bound variable of the inner abstraction (y) and the variable we are substituting for (x), nor the bound variable of the inner abstraction (y) and the term we are substituting (b) are in conflict, we simply substitute x for b *inside* the inner

abstraction. This yields an abstraction which can be applied to another argument. That is applying $(\lambda x.\ (\lambda y.\ x\ y))$ to b returned an abstraction which is "hungry" for another argument. We can now apply that abstraction to another argument:

$$(\lambda y.\ b\ y)\ a \longrightarrow (b\ y)[y := a] = b\ a$$

Let's do the same in one expression:

$$\begin{aligned}(((\lambda x.\ (\lambda y.\ x\ y))\ b)\ a &\longrightarrow ((\lambda y.\ x\ y)[x := b])\ a \\ &= (\lambda y.\ b\ y)\ a \\ &\longrightarrow (b\ y)[y := a] \\ &= (b\ a)\end{aligned}$$

We just applied an abstraction to two arguments. To make this a little easier to see, we can use left-associativity of application and the fact that the scope of a binder goes as far right as possible to rewrite the original expression as

$$(\lambda x.\ \lambda y.\ x\ y)\ b\ a$$

This technique is called *currying* (after Haskell Curry, although he was not the first one to come up with it). It is so common that, usually a short-hand is introduced for abstractions with more than one argument:

$$\begin{aligned}(\lambda x\ y.\ ...) &\equiv (\lambda x.\ \lambda y.\ ...) \\ (\lambda x\ y\ z.\ ...) &\equiv (\lambda x.\ \lambda y.\ \lambda z.\ ...) \\ &\text{etc.}\end{aligned}$$

If we allow arithmetic in our lambda expressions a nice example will be:

$$\begin{aligned}\left(\lambda x\ y.\ \frac{x+y}{y}\right)\ 4\ 2 &\longrightarrow \left(\lambda y.\ \frac{4+y}{y}\right)\ 2 \\ &\longrightarrow \frac{4+2}{2}\end{aligned}$$

Currying is used as the default for functions of multiple arguments by Haskell and OCaml (determined mostly by their standard libraries). On the other hand, Standard ML's library uses tuples as default.

Data types

We see that we can represent functions with multiple arguments in PLC. Surely, for representing other kinds of data (such as booleans, numbers, data structures), we need to introduce extensions and add these as primitive operations? Not really...

Booleans

Many types of values can be represented using *Church encodings*. Booleans are probably the simplest and most straightforward:

$$\begin{aligned} \mathsf{true} &= \lambda t\ f.\ t \qquad (= \lambda t.\ \lambda f.\ t) \\ \mathsf{false} &= \lambda t\ f.\ f \qquad (= \lambda t.\ \lambda f.\ f) \end{aligned}$$

What do these mean? The representation of true is a function that takes two arguments and returns the first one. On the other hand, false returns its second argument. To make sense of these, we need to put them to work and see how they work with boolean operations.

We start with the conditional: if-else. It should take three arguments and return its second one if the first one evaluates to true, and its third argument otherwise. That is we are looking for an expression:

$$\mathsf{if\text{-}then}\ \mathsf{true}\ x\ y \longrightarrow \ldots \longrightarrow x$$

and

$$\mathsf{if\text{-}then}\ \mathsf{false}\ x\ y \longrightarrow \ldots \longrightarrow y$$

Notice something?

$$\begin{aligned} \mathsf{true}\ x\ y &\longrightarrow x \\ \mathsf{false}\ x\ y &\longrightarrow y \end{aligned}$$

That means that all if-then needs to do is to apply its first argument to its second and third argument, since the boolean representation takes care of the selection itself:

$$\mathsf{if\text{-}then} = \lambda b\ t\ f.\ b\ t\ f$$

What about boolean operations?

Let's try to look at conjunction: and. We look for ??? to put in:

```
(λa b. ???) true true   --> ... --> true
(λa b. ???) true false  --> ... --> false
(λa b. ???) false true  --> ... --> false
(λa b. ???) false false --> ... --> false
```

First note that `true true x --> true` for any x, so it seems that `λa b. a b x` could work if we find an appropriate x:

```
(λa b. a b x) true true --> (λb. true b x) true --> true true x --> ... --> true
```

Now note that in all but the first case `and` should reduce to `false`. In the second case,

```
(λa b. a b x) true false --> ... --> true false x --> ... --> false
```

for any x, so that still works. Now, how can we get `false true x --> false`? By taking x to be `false`:

```
(λa b. a b false) false true --> ... --> false true false --> ... --> false
```

The final case also works:

```
(λa b. a b false) false false --> ... --> false false false --> ... --> false
```

Hence

$$\mathsf{and} = \lambda a\ b.\ a\ b\ \mathsf{false}$$

Another way of thinking about the definition of `and` is to define it terms of if-then-else. E.g., in Haskell,

```haskell
and :: Bool -> Bool -> Bool
and a b = if a then b else False
```

which just says that if the first argument is true then the result of `and` depends on the second one, and if its false the result will be false regardless of the second argument.

Based on this, we can express the and operation using `if-else`, which we defined above, and show that it is equivalent to the previous definition by simplifying it using normal order reduction:

```
and =   λa b. if-else a b false
    =   λa b. (λb t f. b t f) a b false
    --> λa b. (λt f. a t f) b false
    --> λa b. (λf. a b f) false
    --> λa b. a b false
```

Can you come up with a representation of or? not?

Pairs

Pairs can be encoded using an abstraction which "stores" its two arguments:

```
pair = λl r s. s l r
```

You can think of a `s` as a "chooser" function which either picks `l` or `r`. Selectors for the first and second element are then, respectively, defined as:

```
fst = λp. p (λl r. l)
snd = λp. p (λl r. r)
```

Take a look at the selector functions we pass to the pair representation. Are they familiar? (Hint: booleans)

Natural Numbers: Church Numerals

Natural numbers are Church-encoded as Church numerals:

```
zero = λs z. z
one = λs z. s z
two = λs z. s (s z)
three = λs z. s (s (s z))
...
```

A numeral for n can be understood as a function that takes some representation of a successor function and some representation of zero and applies the successor to zero n times.

How about operations on numerals? The successor of a numeral `λs z...` is computed by inserting one more application of `s` inside of the abstraction:

```
succ (λs z. z) --> ... --> λs z. s z
succ (λs z. s z) --> ... --> λs z. s (s z)
succ (λs z. s (s z)) --> ... --> λs z. s (s (s z))
...
```

We know that `succ` takes a numeral (which is an abstraction) and returns another numeral, which is again an abstraction:

```
succ = λn. (λs z. ...n...)
```

Taking λs z. z as an example input:

```
(λn. (λs z. ...n...)) (λs z. z)
  --> (λs z. ...(λs z. z)...))
  --> (λs z. s z)
```

We see that we need to apply an extra s *under* λs z.:

```
(λs z. s ...(λs z. z)...) --> ... --> (λs z. s z)
```

To do this we need to "open" the abstraction representing 0. This can be achieved by passing the outer s and z as arguments. We achieve what we wanted.

```
(λs z. s ...(λs z. z) s z...) --> (λs z. s ...z...) = (λs z. s z)
```

Working backwards, we arrive at our successor function:

```
(λs z. s z)
  <-- (λs z. s ((λs z. z) s z))
  <-- (λn. λs z. s (n s z)) (λs z. z)
  = succ (λs z. z)
```

Successor can be thus defined as:

```
succ = λn. (λs z. s (n s z)) = λn s z. s (n s z)
```

Once we have a successor operation, defining addition is quite simple if we keep in mind that a Church numeral m applies its first argument (s) to its second argument (z) m times:

```
plus = λm n. m succ n
```

Multiplication follows the same principle:

$$m * n = \underbrace{n + (...n}_{m \text{ times}} + 0)$$

Hence:

```
times = λm n. m (plus n) zero
```

We can define subtraction via a predecessor function, which is surprisingly more tricky than the successor function. For a numeral `λs z. s (s ... (s z))`, the predecessor should return a numeral with one less `s`. One way of defining a predecessor is via a function that "counts" the number of `s` applications in a numeral, but also remembers the previous count, that is, one less than the total number of applications of `s`:

```
pred = λn. snd (n (λp. pair (succ (fst p)) (fst p)) (pair zero zero))
```

Here the numeral `n` (of which we want to compute the predecessor) is applied to two arguments:

1. The function `(λp. pair (succ (fst p)) (fst p))`. This function takes a pair (bound to `p`) containing two numerals. It returns a pair containing the successor of the first element of `p`, together with its original value. That means, everytime the function is applied to a pair containing numerals `n` and `m`, it returns a pair with numerals corresponding to `n + 1` and `n` (`m` is discarded).
2. A pair containing two zeros: `(pair zero zero)`.

Finally, the second element of the pair is returned – which contains the count of `s` applications, except for the last one.

Here is an example. We let `f = (λp. pair (succ (fst p)) (fst p))`

```
pred three
    = (λn. snd (n f (pair zero zero))) (λs z. s (s (s z)))
  --> snd ((λs z. s (s (s z))) f (pair zero zero))
  --> snd ((λz. f (f (f z))) (pair zero zero))
  --> snd (f (f (f (pair zero zero))))
    = snd (f (f ((λp. pair (succ (fst p)) (fst p)) (pair zero zero))))
  --> snd (f (f (pair (succ (fst (pair zero zero))) (fst (pair zero zero)))))
  --> ...
  --> snd (f (f (pair (succ (fst zero)) (fst zero zero))))
  --> snd (f (f (pair (succ zero) zero)))
  --> ...
  --> snd (f (f (pair one zero)))
    = snd (f ((λp. pair (succ (fst p)) (fst p)) (pair one zero)))
  --> snd (f (pair (succ (fst (pair one zero))) (fst (pair one zero))))
  --> snd (f (pair (succ one) one))
  --> ...
  --> snd (f (pair two one))
  --> ...
  --> snd (pair (succ two) two)
  --> ...
  --> snd (pair three two)
  --> ...
  --> two
```

To subtract `n` from `m`, we need to take 1 away from `m` `n` times.

```
minus = λm n. n pred m
```

For completeness, an alternative predecessor definition is as follows (TODO: explain):

```
pred' = λn f x. n (λg h. h (g f)) (λu. x) (λu. u)
```

We can check if a variable is zero:

```
is-zero = λn.n (λx. false) true
```

We can define $\leq$

```
leq = λm n. is-zero (minus m n)
```

And we can define equality:

```
equal = λm n. and (leq m n) (leq n m)
```

Recursion

We have seen that we can define booleans, together with a conditional, and numbers, together with arithmetic operations in pure lambda calculus. However, to reach full Turing power, we lack one important ingredient: the ability to loop. To loop in a functional setting, we need the little brother of looping: self-reference.

To see that we can loop, let us look at a term, for which β-reduction never terminates in a normal form. This interesting term, called Ω, is defined as follows:

```
Ω = (λx. x x) (λx. x x)
```

We see that we have an abstraction which applies its argument to itself and which is applied to itself. How does reduction proceed?

```
(λx. x x) (λx. x x) --> (x x)[x := (λx. x x)]
  = (λx. x x) (λx. x x) --> (x x)[x := (λx. x x)]
  = (λx. x x) (λx. x x) --> (x x)[x := (λx. x x)]
  ...
```

Immediately after the first reduction step, we are back where we started! Well, we see we can loop forever (diverge), but how is this useful?

In a programming language like OCaml, we are used to defining recursive functions which refer to themselves inside of their body:

```
let rec fact = fun n ->
  if n = 0 then 1 else n * fact (n - 1)
```

How do we achieve this in lambda? While we have been freely using equations to define names for lambda expressions, these were just meta-definitions of names. That is, when we write

```
fact = λn. if-true (is-zero n) one (mult n (?fact? (pred n)))
```

we rely on our meta-language and our common understanding of it to replace any occurrence of `?fact?` with the right-hand side of the above equation, as many times as needed. But this is not beta-reduction, that is we are not defining a recursive function as an object in lambda calculus. To get there, we can think of a recursive definition as follows: "Assuming we have a function to call in the recursive case, we can complete the definition". In Haskell or OCaml, we can simply assume that we already have the function that we are defining. But what is really going on here, is that we can abstract the recursive call as an argument – which corresponds to saying "assuming we already have a function to call in the recursive case":

```
fact = λf. λn. if-true (is-zero n) one (mult n (f (pred n)))
```

Now factorial does not refer to itself anymore, we just need to give it a function to call in the else branch. Easy:

```
fact = (λf. λn. if-true (is-zero n) one (mult n (f (pred n)))) (λn. if-true (is-z
```

Wait, but now what about `f` in the second case? Ah, no problem:

```
fact = (λf. λn. if-true (is-zero n) one (mult n (f (pred n))))
          ((λf. λn. if-true (is-zero n) one (mult n (f (pred n))))
            (λn. if-true (is-zero n) one (mult n (f (pred n)))))
```

This apparently won't work... unless we have a way of supplying an argument for `f` as many times as it's needed. That is, a way to allow the function reference itself whenever it needs to. This is where *fixpoint combinators* come in.

In math, a fixed point of a function f is an input for which the function returns the input itself:

$$f(x) = x$$

If the above holds, we say that x is a fixed point of f. A fixpoint combinator (in general called fix) is an operation that computes the fixed point of a function. That is, it is a function for which the following equation holds:

```
fix f = f (fix f)
```

This equation just spells out that when a function is applied to its fixpoint, the fixpoint shall be returned. Let's use the above equation on itself, by replacing occurrences of `fix f` with the right-hand side:

```
fix f = f (fix f)
      = f (f (fix f))
      = f (f (f (fix f)))
      = ...
```

Now glance above: "If only we had a way of supplying an argument for `f` as many times as it's needed." Seems we are onto something. Let's replace `f` with our factorial:

```
fact = λf. λn. if-true (is-zero n) one (mult n (f (pred n)))

fix fact
  =   fact (fix fact)
  =   (λf. λn. if-true (is-zero n) one (mult n (f (pred n)))) (fix fact)
  --> (λn. if-true (is-zero n) one (mult n ((fix fact) (pred n))))
```

This looks promising. The problem? We haven't *defined* what `fix` is, we are just abusing our meta-notation again. In fact, there is more than one possible definition of `fix`. The simplest one is the *Y* combinator:

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

Notice how the structure is very similar to Ω above. We should check if it is a fixpoint combinator, that is, if it satisfies the fixpoint equation:

```
Y g = (λf. (λx. f (x x)) (λx. f (x x))) g
    = (λx. g (x x)) (λx. g (x x)))
    = g ((λx. g (x x)) (λx. g (x x)))
    = g ((λf. ((λx. f (x x)) (λx. f (x x)))) g)
    = g (Y g)
```

We have ourselves a fixpoint combinator. Let us try to use it to define our factorial function:

```
fact0 = (λf. λn. if-true (is-zero n) one (mult n (f (pred n))))
fact = Y fact0
```

What happens when we try to apply `fact` to a numeral?

```
fact three
  =   Y fact0 three
  =   (λf. (λx. f (x x)) (λx. f (x x))) fact0 three
  --> (λx. fact0 (x x)) (λx. fact0 (x x)) three
  --> fact0 ((λx. fact0 (x x)) (λx. fact0 (x x))) three
  =   fact0 (Y fact0) three
  --> (λn. if-true (is-zero n) one (mult n ((Y fact0) (pred n)))) three
  --> if-true (is-zero three) one (mult three ((Y fact0) (pred three)))
  --> ...
  --> mult three ((Y fact0) (pred three))
  =   mult three (fact0 (Y fact0) (pred three))
  --> ...
  --> mult three (fact0 (Y fact0) (if-true (is-zero (pred three)) one (mult (pred
  ...
  -->
```

However, the Y combinator is not universally applicable under any reduction strategy. Consider what happens with the Y combinator, if we apply the CBV strategy.

```
Y g =   (λf. (λx. f (x x)) (λx. f (x x))) g
    --> (λx. g (x x)) (λx. g (x x))
    --> g ((λx. g (x x)) (λx. g (x x)))
    --> g (g (λx. g (x x)) (λx. g (x x)))
    --> g (g (g (λx. g (x x)) (λx. g (x x))))
    --> ...
```

For CBV, we need the Z combinator:

```
λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

Let bindings

The last useful notation to introduce are let-bindings. We have already implemented them as part of our arithmetic expressions language – both as a substitution-based and environment-based evaluator. Let bindings can be introduced to pure lambda-calculus as *syntactic sugar* – a construct that is defined by translation to a combination of other constructs in the language. Introducing a let-binging corresponds to creating a λ-abstraction and immediately applying it to the bound expression:

```
let x = e1 in e2      ≡      (λx. e2) e1
```

We have to define `let` as syntactic sugar – we cannot write it as a function, the way we did for `if-then`, `add`, etc. Why is that the case?

We can also define a recursive version of let – called **`let rec`** in OCaml, **`letrec`** in Scheme:

```
let rec f = e1 in e2   ≡   let f = fix (λf. e1) in e2
                       ≡   (λf. e2) (fix (λf. e1))
```

Where `fix` is an appropriate fixpoint combinator (e.g., Y under CBN, Z under CBV and CBN).

Most languages also allow specifying function arguments to the left-hand side of the equal sign:

```
let f x y z = e1 in e2
let rec f x y z = e1 in e2
```

```
(define (f x y z) e1)
```

These can be translated as:

```
let f x y z ... = e1 in e2   ≡   let f = λx y z. e1 in e2
                             ≡   (λf. e2) (λx y z. e1)
```