Lecture 10 & 11: Introduction to Lambda Calculus

CS4400 Programming Languages

Readings: [TAPL] Ch. 5 (available online via our library)

Lambda Calculus

Lambda calculus is a theory of functions. What is a function? There are two basic views one can take when characterizing them:

1. Function as a graph
2. Function as a rules

Considering a function f as a graph is to consider it as a set of pairs – mappings between input and output values $(x, f(x))$. For example the square function on natural numbers $^2 : \mathbb{N} \to \mathbb{N}$ can be characterized as a set of pairs (n, n^2):

$$\{(0,0), (1,1), (2,4), (3,9), (4,16), (5,25), ...\}$$

Using a function as a graph is to find an output that corresponds to our input. The alternative view to take is to consider a function as rules – equations, which tell us how to compute the output of the function from its input. For example, the square function $^2 : \mathbb{N} \to \mathbb{N}$ is defined by the equation:

$$n^2 = n \times n$$

How do we use this function? We *substitute* an expression that looks like the left-hand side with the right-hand side, replacing the *argument* n with the expression and then computing the resulting expression. For example, our calculation might proceed as follows:

$$\begin{aligned} 4^{2^2} + 3^2 &= (4 \times 4)^2 + 3^2 \\ &= ((4 \times 4) \times (4 \times 4)) + 3^2 \\ &= ((4 \times 4) \times (4 \times 4)) + (3 \times 3) \\ &\ldots \\ &= 265 \end{aligned}$$

Or, as follows:

$$\begin{aligned} 4^{2^2} + 3^2 &= (4 \times 4)^2 + 3^2 \\ &= 16^2 + 3^2 \\ &= 16^2 + 9 \\ &= 256 + 9 \\ &\ldots \\ &= 265 \end{aligned}$$

In any case, the important thing to note is that we replace any occurrence of n^2 for any n using the defining equation. In general, if we define a function f by the equation $f(x) = E$, where E is some mathematical expression (potentially containing x), then we use (apply) this function by replacing any occurrence of $f(D)$ (where D is a mathematical expression) by $E[x := D]$, that is the expression E where all occurrences of x are replaced by D. This is called *substitution* of a variable x in an expression E for another expression D. E.g., if

$$f(x) = x + x$$

then:

$$\begin{aligned} f(20) + f(2 \times 3) &= (x + x)[x := 20] + (x + x)[x := 2 \times 3] \\ &= (20 + 20) + ((2 \times 3) + (2 \times 3)) \\ &\ldots \end{aligned}$$

The next question is, how important is the name of the function? We use names as mnemonics, so that we can say we can

1. say "let f be the function defined by the equation $f(x) = E$" (where E is an arbitrary mathematical expression), and
2. replace any occurrence of f applied to an argument with an instance of E where x is replaced with the argument expression.

We can do this without inventing names, by using functions as anonymous objects – just like we easily use numbers or strings or arrays. In mathematics an anonymous function will be written as $x \mapsto E$. For example, the square function is $x \mapsto x \times x$, the above function f is $x \mapsto x + x$.

The above exposition applies to programming too. Basically, all sensible "higher-level" programming languages allow us to define functions to abstract a computation by replacing a concrete expression with a variable – a placeholder. In Python we might write:

```
def square(x):
  return x * x
```

In C/C++:

```
int square(int x) {
  return x * x;
}
```

In Haskell:

```
square :: Integer -> Integer
square x = x * x
```

In Scheme:

```
(define (square x)
  (* x x))
```

In any programming language we operate with the rough understanding that whenever `square` is invoked with an argument, that application might as well be replaced with the body of the function with the argument variable replaced with the actual argument (either before or after evaluating the argument itself). More and more programming languages, particularly those which allow passing functions as arguments, allow creating functions without naming them – so called anonymous functions. Python and Scheme have `lambda`:

```
lambda x : x * x
```

```
(lambda (x) (* x x))
```

OCaml has `fun` or `function`:

```
fun x => x * x
```

Haskell has the backslash:

```
\x -> x * x
```

C++ has, well...

```
[](int x){ return x * x; }
```

As hinted by the Scheme and Python examples, Lambda calculus is the underlying theory behind these anonymous functions. In its pure form, it is exclusively concerned with what it means to apply an abstracted expression (as an anonymous function), to an argument. It studies this as a purely syntactic operation.

Where Python and Scheme have `lambda`, OCaml has `fun` and `function`, Lambda calculus has λ. That is an anonymous function with the formal parameter x is constructed using $\lambda x...$ We can write the squaring function in lambda notation as

$$\lambda x.\ x \times x$$

We say that this is a *lambda abstraction* that *binds* the variable x in $x \times x$. In other words, x is bound in $x \times x$. An application is written (similarly to Scheme, OCaml, or Haskell) by writing the function and argument next to each other (juxtaposition). For example, where in Scheme we could write

```
((lambda (x) (* x x)) 10)
```

and in Haskell

```
(\x -> x * x) 10
```

In lambda notation we write:

$$(\lambda x.\ x \times x)\ 10$$

As I mentioned before, Lambda calculus looks at the application of a function as a syntactic operation, in terms of *substitution*, as the process of replacing any occurrence of the abstracted variable with the actual argument. For the above, this is replacing any occurrence of x in $x \times x$ with 10:

$$\begin{aligned}(\lambda x.\ x \times x)\ 10 &= (x \times x)[x := 10] \\ &= 10 \times 10\end{aligned}$$

Another way of thinking about the bound variable x in the $\lambda x.\ x \times x$ as a placeholder or hole, where the argument "fits".

$$(\lambda\square.\ \square \times \square)\ 10 = \boxed{10} \times \boxed{10}$$

Pure Lambda Calculus

Here, we will look at the formal theory of pure Lambda Calculus. We will look at the syntax and a notion of computation.

Syntax

The basic syntax of the calculus is really simple:

```
<Lambda> ::= <Variable>
           | ( <Lambda> <Lambda> )
           | ( λ <Variable> . <Lambda> )
```

That is all there really is:[1]

- variable reference, e.g. x, y, z, a, $square$
- application, e.g., $(x\ y)$, $((\lambda x.\ x)\ (\lambda x.\ x))$
- lambda abstraction, e.g.,
  - $(\lambda x.\ x)$ – expressing the identity function
  - $(\lambda x.\ x\ x)$ – a function that applies its argument to itself

You might ask: what can we do with such a minuscule language? Turns out a lot. As proven by A.M. Turing, this pure version of Lambda calculus is equivalent in computational power to Turing Machines. That means we are able to build up a programming language out of these three constructs.

Syntax Conventions and Terminology Terminology: Take a lambda abstraction:

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

1. λx is a *binder* binding x
2. N is the *body* of the abstraction

To avoid writing too many parentheses, these conventions are usually taken for granted:

1. Outermost parentheses are usually dropped: `x x`, `λx. x`.
2. Application *associates to the left*. That is, `(((a b) c) d)` is the same as `((a b c) d)` is the same as `(a b c d)`, which is the same as `a b c d` (see previous rule).
3. Lambda abstraction bodies extend as far to the right as possible. That is, `(λa. (λb. ((a b) c)))` is the same as `λa. λb a b c`.

[1] Later, we will add extensions that make many things simpler and also allow us to build realistic programming languages.

Beta Reduction

Computation in pure lambda calculus is expressed in a single rule: the β-reduction rule:

$$(\lambda x.\ M)\ N \longrightarrow_\beta M[x := N]$$

The long arrow stands for "reduces to". On the left-hand side, we have an application of a lambda abstraction to an arbitrary term. On the right-hand side, we substitute the abstraction's bound variable with the argument. A term that matches the pattern on the left-hand side (that is, a lambda abstraction applied to something) is called a *redex*, short for *reducible expression*. For example:

- `(λx. x) a -->β a`
- `(λx. x x) (λy. y) -->β (λy. y) (λy. y)`
- the above reduces further: `(λy. y) (λy. y) -->β (λy. y)`
- not a redex: `(x x)`
- also not a redex: `x (λy. y)`
- also not a redex: `(λy. (λx. x) y)`, although it does contain a redex `((λx. x) y)`

Variables: Bound, Free. Closed Expressions

We have already mentioned the notion of a bound variable. A variable is said to be *bound* in an expression, if it appears under a λ-abstraction binding that particular variable. Or, in other words, it is bound if it appears in the scope of a binder. For example:

- `x` is bound in `(λx. x x)` – it appears in the scope of the binder `λx`
- both `x` and `y` are bound in `(λx. λy. x y)` – `x` appears in the scope of `λx`, `y` in the scope of `λy`
- `x` is *not* bound in `(λy. x y)`, but `y` is – `x` does not appear in the scope of any binder here, while `y` appears in the scope of `λy`

A *free* variable is one which appears in a position where it is not bound. For example:

- `x` is free in `x x`, in `λy. x y`, or in `(λy. y y) x`
- `x` is *not* free in `(λx. x x) (λx. x)`
- `x` is *both* bound and free in `(λx. x y) x`, while `y` is only free

As you can see above, a variable might be both bound and free in an a expression.

An expression which contains no free variables is *closed*, for example:

- `λx. x`
- `λx. λy. x y x`

A closed lambda expression is also called a *combinator*.

A variable is called *fresh* for an expression, if it does not appear free in that expression. For example, `x` is fresh for `y z` or `(λx. x x)`.

Names of Bound Variables Don't Matter

Intuitively, an identity function should be an identity function, no matter what we choose to name its bound variable. That is, `(λx. x)` should be considered the same as `(λy. y)` or `(λz. z)`. This is captured in the notion of alpha equivalence: two expressions are β-*equivalent*, if they only differ in the names of their bound variables. This also means, that we are free to β-*convert* any lambda term by consistently renaming bound variables. However, the new names must differ from free variables under the particular binder. We are thus free to convert `(λx. x)` to, e.g., `(λa. a)`; `(λy. z y)` to `(λx. z x)`, but not to `(λz. z z)`.

Substitution

We were happy to use substitution in an informal manner up until now:

> $M[x := N]$ means replacing occurrences of the variable x in the expression M with the expression N.

Here we want to pin it down. For that, we will need to consider the difference between bound and free variables. Let's try to start with a naive definition of substitution.

Naive Substitution There are three syntactic forms, we need to consider each form:

Variable: `x[y := N] = ?`

Application: `(M1 M2)[y := N] = ?`

Abstraction: `(λx. M)[y := N] = ?`

Variables are straightforward: we either find the variable to be substituted or we find a different one:

1. `y[y := N] = N`
2. `x[y := N] = x` if `x` $\neq$ `y`

Application is also relatively simple – we simply substitute in both left-hand and right-hand side:

3. `(M1 M2)[y := N] = (M1[y := N] M2[y := N])`

Now, for a lambda abstraction we need to consider the variables involved. We certainly don't want to override the bound variable of a function:

4. `(λy. M)[y := N] = (λy. M)`

The remaining case seems simple enough too:

5. `(λx. M)[y := N] = (λx. M[y := N])` if $x \neq y$

If we test this substitution everything seems to be okay:

```
(x x)[x := (λy. y)] = (x[x := (λy. y)] x[x := (λy. y)])
                    = (λy. y) (λy. y)

((λx. x y) x)[x := (λy. y)] = (λx. x y)[x := (λy. y)] x[x := (λy. y)]
                            = (λx. x y) (λy. y)
```

However, what happens if the expression that we are substituting contains the bound variable?

```
(λy. x)[x := y] = λy. y
```

We see that in this case, we have just "captured" variable `y` and changed its status from free to bound. This changes the meaning of a variable – whereas the original meaning of `y` was given by the context of the left-hand side expression, now it is given by the binder `λy`. In particular, we changed a constant function—which, after the substitution should return a free `y`, no matter what argument it is applied to—to an identity function, that just returns whatever its argument is.

From this we see that we need substitution to behave differently when there the expression that we are trying to substitute, contains free variables that clash with variables bound by a lambda-abstraction.

Safe Substitution To fix this we can restrict when the last case of our substitution applies:

5. `(λx. M)[y := N] = (λx. M[y := N])` if $x \neq y$ and if `x` is not free in `N`

Now our substitution is "safe". However, this turns it into a partial function – it is left undefined for cases where the bound variable `x` appears free in `N`. To go around this, we can make use of alpha-conversion: we consistently rename the bound variable `x` to one that doesn't clash with `y` or the free variables in `N` or `M`. Only then do we perform the actual substitution of `y`.

5. `(λx. M)[y := N] = (λx'. M[x := x'][y := N])` if `x` $\neq$ `y` and `x'` is fresh for `y`, `N` and `M`

Now substitution is a total function again. For an implementation, we just need to know how to pick a fresh variable. Notice how we replace the bound variable `x` with `x'` and also rename any ocurrence of `x` to `x'` in the body `M`. Since `x'` is chosen so that it does not appear free in `M` or `N`, we are avoiding any potential clashes.

Reduction Strategies

Beta reduction tells us how to reduce a redex. The missing piece of the puzzle is how to decide where to look for a redex and apply the beta-reduction rule. This is given by reduction strategies.

(The following text is taken, with minor modifications, from *Types and Programming Languages*)

Full Beta-Reduction Under this strategy, any redex may be reduced at any time. At each step we pick some redex, anywhere inside the term we are evaluating, and reduce it. For example, consider the term:

```
(λa. a) ((λb. b) (λz. (λc. c) z))
```

This term contains three redexes:

- `(λa. a) ((λb. b) (λz. (λc. c) z))`
- `(λb. b) (λz. (λc. c) z)`
- `(λc. c) z`

Under full beta-reduction, we might choose, for example, to begin with the innermost redex, then do the one in the middle, then the outermost:

```
(λa. a) ((λb. b) (λz. (λc. c) z))
  --> (λa. a) ((λb. b) (λz. z))
  --> (λa. a) (λz. z)
  --> λz. z
```

`λz. z` cannot be reduced any further and is a *normal form*.

Note, that under full beta-reduction, each reduction step can have more than possible one result, depending on which redex is chosen.

Normal Order Under normal order, the leftmost, outermost redex is always reduced first. Our example would be reduced as follows:

```
(λa. a) ((λb. b) (λz. (λc. c) z))
  --> (λb. b) (λz. (λc. c) z)
  --> λz. (λc. c) z
  --> λz. z
```

Again, `λz. z` is the normal form and cannot be reduced further.

Because each redex is chosen in a deterministic manner, each reduction step has one possible result – reduction thus becomes a (partial) function.

Call by Name Call by name puts more restrictions on which redexes are fair game, and disallows reductions inside abstractions. For our example, we perform the same reduction steps as normal form, but stop short of "going under" the last abstraction.

```
(λa. a) ((λb. b) (λz. (λc. c) z))
  --> (λb. b) (λz. (λc. c) z)
  --> λz. (λc. c) z
```

Haskell uses an optimization of call by name, called call by need or lazy evaluation. Under call by name based strategies, arguments are only evaluated if they are needed.

Call by Value Under call by value, only outermost redexes are reduced and each redex is only reduce after its right-hand side has been fully reduced to a normal form.

```
(λa. a) ((λb. b) (λz. (λc. c) z))
  --> (λa. a) (λz. (λc. c) z)
  --> λz. (λc. c) z
```

Evaluation strategies based on call by value are used by the majority of languages: an argument expression is evaluated to a value before it is passed into the function as an argument. Such a strategy is also called *strict*, because it strictly evaluates all arguments, regardless of whether they are used.