CS 152 Abstractions and Programming Paradigms
Lecture on Lambda Expressions

Rushikesh K Joshi

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

The Lambda Calculus

- Lambda calculus is a calculus with its core features of function definition (**abstraction**) and function **application** by variable substitution.
- The untyped lambda calculus was invented by Church years ago. The basic syntax is extremely tiny.
- Anything that is computable can be expressed in lambda calculus, and it is equivalent to 'Turing Machine' (you will learn this in another course)
- It's interesting to see how one can explain the computational structures that you see in programming languages in terms of expressions in lambda calculus (lambda expressions)

Example: Abstraction and Application

- **Definition, i.e. Abstraction**:
  f (x) = x * x
- **Application, compute it by substitution**:
  f 3
  = 3 * 3
  = 9

But in the basic lambda calculus, we have only symbols and no predefined numbers..Let's see what this means..

Abstractions in Lambda Calculus

- **A function with 1 argument**:
  $\lambda x.x$
  It takes one value as input argument, and it returns the same value as its result
- Application and its reductuon sequence:
  $(\lambda x.x) \;\; m \quad \rightarrow$ m
  On the left hand side we have the function definition, which is applied to a value. This application reduces to a value after substitution.

Abstractions in Lambda Calculus

- **A function with 2 arguments**: $\lambda x.\lambda y.x\ y$
- It has an outer body with one argument, the outer body returns a function, again a function with one argument
- The inner function applies x to y, x has been sourced in from the outer function
- All lambda abstraction accept only argument. In the manner shown above, we can define functions which accept two arguments. This method is called **Currying**.

Abstractions in Lambda Calculus

- **A function with 2 arguments**: $\lambda x.\lambda y.x\ y$
- An application and its reduction sequence: $(\lambda x.\lambda y.x\ y)(\lambda x.x)p \rightarrow (\lambda y.(\lambda x.x)y)p \rightarrow (\lambda y.y)p \rightarrow p$
- The substitution process is called Beta Reduction
- If we cannot reduce an expression further, we are said to have reached the **normal form**, else we have a **redex**, a reducible expression

We are Almost Done about the basic Lambda Calculus (omitting some more details)

Where is the rest of the world?

i.e. Numbers, Arithmetic Expressions, Logical Operators, Booleans Lists...?

All that can be build in terms of just what we have done so far... Let's see how..

Church Numerals

The core concept behind Church's solution is this: A number *n* can be thought of as a function that accepts another function *f* and applies *f* to another argument *z* exactly as many times as the value of number *n*. So,

C0 = $\lambda f.\lambda z.z$
C1 = $\lambda f.\lambda z.f\ z$
C2 = $\lambda f.\lambda z.f\ (f\ z)$
C3 = $\lambda f.\lambda z.f\ (f\ (f\ z))$
C4 = $\lambda f.\lambda z.f\ (f\ (f\ (f\ z)))$ and so on

Successor Function on a Numeral

$successor = \lambda m.\lambda f.\lambda z.f\ (m\ f\ z)$

- How does it work?
- Function f is applied m times to z. To the result, f is applied once more by the outer application.

Adding Numerals

$$add = \lambda m.\lambda n.\lambda f.\lambda z.m\ f\ (n\ f\ z)$$

- How does it work?
- f is applied n times to z. To the result produced, f is applied m times.
- Effectively, f is being applied $(m + n)$ times to z
- That behavior is consistent with our representation of natural number $(m + n)$.

Multiplying Numerals

$mult = \lambda m.\lambda n.\lambda f.\lambda z.m\ (n\ f)\ z.$

- How does it work?
- $n\ f$ is applied m times to z.
- If it is applied once, it will result in $n\ f\ z$, i.e. n applications of f to z.
- So overall, we obtain $(m * n)$ successive applications of f to initial value z, which represents natural number $(m * n)$.

Church Booleans

$$\mathit{true} = \lambda x.\lambda y.x$$

$$\mathit{false} = \lambda x.\lambda y.y$$

If Then Else

$$ifThenElse = \lambda l.\lambda m.\lambda n.l\ m\ n.$$

Logical Operators

$$\begin{aligned}
and &= \lambda b_1.\lambda b2.b1\ b2\ false \\
or &= \lambda b_1.\lambda b2.b1\ true\ b2 \\
isfalse &= \lambda b.b\ false\ true \\
istrue &= \lambda b.b\ true\ false \\
not &= isfalse \\
xor &= \lambda b_1.\lambda b_2.b1(not\ b2)b2
\end{aligned}$$

Logical Operators

Verify these by evaluating the following applications:

- and true true
- and false true
- or true true
- or false true
- xor true true

The Pair Abstraction

(*pair u v*) constructs a pair

(*first pair u v*) should return the first element in this pair, which is *u*.

(*second pair u v*) should return the second element in this pair, which *u*.

Solution to the Pair Abstraction

pair = $\lambda x.\lambda y.\lambda f.f\ x\ y$.

first = $\lambda p.p\ \mathit{true}$.

second = $\lambda p.p\ \mathit{false}$.

Now try the following exercises:
(1) first pair u v (2) second pair (pair u v) (pair x y)