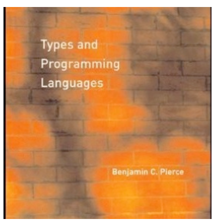
Introduction to Lambda Calculus

Lecture 5
CS 565



Lambda Calculus

So far, we've explored some simple but non-interesting languages

- language of arithmetic expressions
- IMP (arithmetic + while loops)

We now turn our attention to a simple but interesting language

- Turing complete (can express loops and recursion)
- Higher-order (functional objects are values)
- Interesting variable binding and scoping issues
- Foundation for many real-world programming languages
  Lisp, Scheme, ML, Haskell, ...

Intuition

Suppose we want to describe a function that adds three to any input:

- `plus3 x = succ (succ (succ x))`

- Read "*plus3 is a function which, when applied to any number x, yields the successor of the successor of the successor of x*"

- Note that the function which adds 3 to any number need not be named plus3; the name "plus3" is just a convenient shorthand for naming this function

```
(plus3 x) (succ 0) ≡
        ((λ x.(succ (succ (succ x)))) (succ 0))
```

Basics

There are two new primitive syntactic forms:

- `λ x.t`

  "The function which when given a value `v`, yields `t` with `v` substituted for `x` in `t`."

- `(t1 t2)`

  "the function `t1` applied to argument `t2`"

- Key point: functions are anonymous: they don't need to be named. For convenience we'll sometimes write:

```
plus3 ≡ λ x. (succ (succ (succ x)))
```

- but the naming is a metalanguage operation.

Abstractions

Consider the abstraction:

```
g ≡ λ f. (f (f (succ 0)))
```

The argument `f` is used in a function position (in a call).

We call `g` a higher-order function because it takes another function as an input.

Now, `(g plus3)`

```
= (λ f. (f (f (succ 0)))
  (λ x .(succ (succ (succ x))))
= ((λ x.(succ (succ (succ x)))
  ((λ x.(succ (succ (succ x))))(succ 0)))
= ((λ x. (succ (succ (succ x))))
  (succ (succ (succ (succ 0)))))
= (succ(succ(succ(succ(succ(succ(succ 0)))))))
```

Abstractions

Consider

```
double ≡ λ f. λ y. (f (f y))
```

The term yielded by applying double is another function
`(λ y. (f (f y))`

Thus, double is also a higher-order function because it returns a function when applied to an argument.

Example

```
(double plus3 0)

= ((λ f.λ y.(f (f y))) (λ x.(succ (succ (succ x)))) 0)

= ((λ y.((λ x.(succ (succ (succ x))))
    ((λ x. (succ (succ (succ x)))) y)))
   0)

= ((λ x. (succ (succ (succ x))))
   ((λ x. (succ (succ (succ x)))) 0))

= ((λ x. (succ (succ (succ x)))) (succ (succ (succ 0))))

= (succ (succ (succ (succ (succ (succ 0))))))
```

Key Issues

How do we perform substitution:

- how do we bind "free variables", the variables that are non-local in the function
- Think about the occurrences of `f` in

`λ y. (f (f y))`

How do we perform application:

- There may be several different application subterms within a larger term.
- How do we decide the order to perform applications?

Pure Lambda Calculus

The only value is a function

- Variables denote functions
- Functions always take functions as arguments
- Functions always return functions as results

Minimalist

- Can express essentially all modern programming constructs
- Can apply syntactic reasoning techniques (e.g. operational semantics) to understand behavior.

Scope

The λ abstraction `λ x.t` binds variable `x`.

The scope of the binding is `t`.

Occurrences of `x` that are not within the scope of an abstraction binding `x` are said to be free:

```
λ x. λ y.(x y z)
λ x.((λ y.z y) y)
```

Occurrences of `x` that are within the scope of an abstraction binding `x` are said to be bound by the abstraction.

Free Variables

Intuitively, the free variables of an exp are "non-local" variables

- Define `FV(M)` formally thus:

```
FV(x) = {x}
FV(M1 M2) = FV(M1)  ∪  FV(M2)
FV(λ x. M) = FV(M) — {x}
```

- Free variables become bound after substitution.
- But, if proper care is not taken, this leads to unexpected results:

```
(λx.λy. y x) y = λy. y y
```

- We say that term `M` is α-congruent to `N` if `N` results from `M` by a series of changes to bound variables:

`λx.(x y)` α-congruent to `λz.(z y)`
not α-congruent to `λy.(y y)`
`λx.x (λx.x)` α-congruent to `λx'.x'(λx.x)` and
α-congruent to `λx'.x'(λx''.x'')`

Substitution

`λx.M` α-congruent to `λy.M[y/x]` if `y` is not free or bound in `M`.

- Want to define substitution s.t. `(λx.N)M → [M/x]N`

Define this more precisely:

- Let `x` be a variable, and `M` and `N` expressions.
  Then `[M/x]N` is the expression `N'`:

  `N` is a variable: (case 1)

  `N = x` then `N' = M` (1.1)

  `N ≠ x` then `N' = N` (1.2)

  `N` is an application `(Y Z)`: (case 2)

  `N' = ([M/x]Y) ([M/x]Z)`

Substitution (cont)

N is an abstraction λy.Y (then [M/x]N is the expression N')
(case 3)

y = x then N' = N (3.1)

y ≠ x then:

x does not occur free in Y or if y does not occur free in M:

N' = λy.[M/x]Y (3.2.1)

x does occur free in Y and y does occur free in M:

N' = λz.[M/x]([z/y]Y) for fresh z (3.2.2)
First change bound variable y in Y to z, then perform substitution

Example

```
(λp.(λq.(λp.p( p q))(λr.(+ p r)))(+ p 4)) 2
[(+ p 4)/q]((λp.p(p q))(λr.(+ p r)))
([(+ p 4)/q](λp.p(p q)))([(+ p 4)/q](λr.(+ p r)))
                                                  (by case 2)
([(+ p 4)/q](λp.p(p q))) (λr.(+ p r))
            (by case 3.2.1 since q does not occur free in (+ p r))
(λa.[(+ p 4)/q]([a/p](p(p q)))) (λr.(+ p r))
                                                  (by case 3.3.2)
(λa.a (a (+ p 4))) (λr.(+ p r))
(λp. (λa.a (a (+ p 4)))(λr.(+ p r))) 2
```

Operational Semantics

Values:

$$\lambda\ x.\ t$$

Computation rule:

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

Congruence rules

$$\frac{t1 \rightarrow t1'}{(t1\ t2) \rightarrow (t1'\ t2)}$$

$$\frac{t2 \rightarrow t2'}{(v\ t2) \rightarrow (v\ t2')}$$

The computation rule is referred to as the β-substitution or β-conversion rule. $((\lambda\ x.\ t)t')$ is called a β-redex.

Evaluation Order

Outermost, leftmost redex first

Arguments to application are evaluated before application is performed

- Call-by-value
- “Strict”

Other orders do not evaluate arguments before application

- E.g. normal order
- “Lazy”

Example

```
(λx.x) ((λx.x) (λz.(λx.x) z))
id (id (λz.id z))
```

(with `id ≡ λx.x`)

Call-by-value (strict):

```
  id (id (λz. id z))
= id (λz. id z)
  (1st id would come 1st, but arg
   must be evaluated)
= λz. id z
```

Normal order (lazy):

```
  id (id (λz. id z))
= id (λz. id z)
= λz. id z
= λz.z
```

Multiple arguments

The λ calculus has no built-in support to handle multiple arguments.

However, we can interpret λ terms that when applied yield another λ term as effectively providing the same effect:

Example:

```
double ≡ λ f. λ x. (f (f x))
```

- We can think of double as a two-argument function.

Representing a multi-argument function in terms of single-argument higher-order functions is known as currying.

Programming Examples: Booleans

```
true ≡ λ t. λ f. t
false  ≡ λ t. λ f. f

(true v w) → ((λ t.λ f. t) v) w) →
             ((λ f. v) w) →
             v
(false v w) → ((λ t.λ f. f) v) w) →
              ((λ f. f) w) →
              w
```

Booleans (cont)

```
not ≡ λ b. b false true
```

The function that returns true if b is false, and false if b is true.

```
and ≡ λ b. λ c. b c false
```

The function that given two Boolean values (v and w) returns w if v is true and false if v is false. Thus, (and v w) yields true only if both v and w are true.

Pairs

We can encode common operations on pairs thus:

```
pair ≡ λf. λs. λb.b f s
fst  ≡ λp. p true
snd  ≡ λp. p false
```

Example:

```
fst (pair v w) →
fst ((λf. λs. λb.b f s)v w) →
fst ((λs. λb.b v s)w) →
(λp.p true)(λb.(b v w) →
(λb.b v w) true →
true v w →* v
```

Numbers (Church Numerals)

There are no explicit operations to manipulate numbers

Encode numbers with higher-order functions

```
zero ≡ λs. λz. z
one  ≡ λs. λz. s z
two  ≡ λs. λz. s (s z)
```

- read s as successor and z as zero

Numbers

```
succ ≡ λn. λs. λz.s (n s z)
```

A function that takes s and z and applies s repeatedly to z

```
plus ≡ λm. λn. λs. λz.m s(n s z)
```

takes two Church numerals and yields another Church numeral that given s and z applies s iterated n times to z and then applies s iterated my times to the result

Example

```
(plus one two succ zero) ⟶
(plus (λ s. λ z.(s z)) (λ s. λ z.(s (s z))) succ zero) ⟶
(λ s. λ z.((λ s. λ z.(s z)) s ((λ s. λ z.(s (s z))) s z) succ zero) →
(λ s. λ z.((λ s. λ z.(s z)) s ((λ s. λ z.(s (s z))) s z) succ zero) →
((λ s. λ z.(s z)) succ ((λ s. λ z.(s (s z))) succ zero)) ⟶
((λ s. λ z.(s z)) succ (succ ( succ zero))) ⟶
((λ s. λ z.(s z)) succ (succ ( succ zero))) ⟶
(succ (succ ( succ zero)))
```