CS 314 Lecture 7

Lambda calculus

February 12, 2019

Lambda calculus

Our substitution rule for function application is formally called β-reduction.

We'll need one other rule (the alligator's color changing rule).

Some common functions:

- $\lambda x.x$
- $\lambda x.y$
- $\lambda xy.x$
- $\lambda xy.y$

Some common functions:

- $\lambda x.x$ (id)
- $\lambda x.y$ (const y)
- $\lambda xy.x$ (first)
- $\lambda xy.y$ (second)

α-equivalence

These are the same:

- $\lambda x.x$
- $\lambda y.y$
- $\lambda z.z$

Free and bound variables

In the expression $\lambda x.xy$, we say the x in the body is *bound* (by the enclosing λ), but y is free.

Free and bound variables

Recall the three types of lambda terms:

- variables – x
- abstractions – $\lambda x.x$
- applications – $(\lambda x.x)y$

Free and bound variables

Recall the three types of lambda terms:

- variables – x
- abstractions – $(\lambda x.M)$
- applications – $(M\ N)$

Bound variables

BV denotes the bound variables of a lambda term:

- $BV\ x = \{\}$
- $BV\ (\lambda x.M) = (BV\ M) \cup \{x\}$
- $BV\ (M\ N) = (BV\ M) \cup (BV\ N)$

Free variables

FV denotes the free variables of a lambda term:

- $FV\ x = \{x\}$
- $FV\ (\lambda x.M) = (FV\ M) - \{x\}$
- $FV\ (M\ N) = (FV\ M) \cup (FV\ N)$

If $(FV\ M) = \{\}$, M is called *closed*. M is also called a *combinator*.

Free and bound variables

Note that FV and BV may not be disjoint!

In $x(\lambda xy.x)$, the first x is free, but the remaining x and y are bound.

Induction

Theorem: All lambda terms have balanced parentheses.

Recall the rules for building lambda terms:

- A variable is a lambda term.
- If M is a lambda term, then $(\lambda x.M)$ is a lambda term.
- If M and N are lambda terms, then $(M\ N)$ is a lambda term.

Induction

Theorem: All lambda terms have balanced parentheses.

Induction:

- Variables have balanced parentheses (none).
- If M is balanced, then $(\lambda x.M)$ is balanced (adds one left, one right).
- If M and N are lambda terms, then $(M\ N)$ is a lambda term (adds one left, one right).

Equivalence

$$\lambda x.x = \lambda y.y$$

$$\lambda xyz.abc = \lambda mno.abc$$

$$\lambda x.\lambda y.xy = \lambda a.\lambda b.ab$$

Variable capture

$$\begin{aligned}(\lambda x.\lambda y.xy)yz &= (\lambda y.yy)z \\ &= zz\end{aligned}$$

But...

$$\begin{aligned}(\lambda a.\lambda b.ab)yz &= (\lambda b.yb)z \\ &= yz\end{aligned}$$

Variable capture

This is called variable capture: a variable that was free becomes bound.

$$(\lambda x.\lambda y.xy)yz \Rightarrow (\lambda y.yy)z$$

- The x in $\lambda y.xy$ is free (although bound in $\lambda x.\lambda y.xy$)
- But both ys in $\lambda y.yy$ are bound.

α-conversion

We can rename xs in $\lambda x.M$ with y, as long as y is not already a free variable in the body:

$\lambda x.M \equiv \lambda y.M[x := y]$, where $y \notin FV\ M$

- $\lambda x.xx = \lambda y.yy$
- $\lambda x.xy \neq \lambda y.yy$

η-reduction

One last rule (η-reduction):

Given an expression of the form $\lambda x.fx$, we can replace this with f.

```
double mySin(double x)
{
    return sin(x);
}
```

Normal form

We say a lambda term is in *normal form* when it can't be reduced any further.

Does every lambda term have a normal form?

Normal form

Does every lambda term have a normal form? Consider this expression:

$$(\lambda x.xx)(\lambda x.xx)$$

- Replace every x in xx with the argument $(\lambda x.xx)$
- We get $(\lambda x.xx)(\lambda x.xx)$
- We can do function application!
- ...

Normal form

Does every expression that contains $(\lambda x.xx)(\lambda x.xx)$ inevitably loop?

Normal form

Does every expression that contains $(\lambda x.xx)(\lambda x.xx)$ inevitably loop?

Consider this expression:

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$$

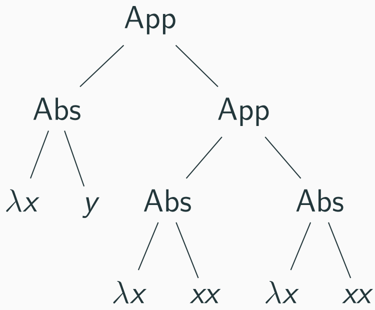
Evaluation order

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$$

- Applicative order: evaluate the argument to a normal form first
- Normal order: evaluate the top-most element first

Evaluation order

Parsing $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$:



Evaluation order

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$$

- Applicative order: infinite loops!
- Normal order: returns y

Evaluation order

In general, if an expression has a normal form, normal order evaluation will reach it. But it can be inefficient:

$$(\lambda x.xx)((\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b))$$

- Applicative order: evaluate argument first
- Normal order: do function application first

Evaluation order

Applicative:

- $(\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b)$
- $(\lambda ab.b)(\lambda w.w)(\lambda m.m)$
- $\lambda m.m$
- Then only do application: $(\lambda x.xx)(\lambda m.m)$
- $(\lambda m.m)(\lambda m.m)$
- $\lambda m.m$

Evaluation order

Normal:

- Do application: $(\lambda x.xx)((\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b))$
- $((\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b))((\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b))$
- Reduce function: $(\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b) \Rightarrow ... \Rightarrow \lambda m.m$
- Do application: $(\lambda m.m)((\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b))$
- $(\lambda xyz.zxy)(\lambda w.w)(\lambda m.m)(\lambda ab.b)$
- $(\lambda ab.b)(\lambda w.w)(\lambda m.m)$
- $\lambda m.m$

Church booleans

We can do more than just symbol manipulation with lambda calculus.

Let's define boolean values in terms of lambda expressions:

- $\lambda x.\lambda y.x \equiv \text{true}$
- $\lambda x.\lambda y.y \equiv \text{false}$

Boolean functions

- not
- and
- or

Boolean functions

not is a function that negates its argument:

x	not x
true	false
false	true

Boolean functions

not is a function that negates its argument:

```
not(x) = x ? false : true
```

another way to write this:

```
not x = if x
          then false
          else true
```

Boolean functions

not is a function that negates its argument:

x	not x
$\lambda xy.x$	false
$\lambda xy.y$	true

not: $\lambda p....$

Boolean functions

not is a function that negates its argument:

x	not x
$\lambda xy.x$	false
$\lambda xy.y$	true

not: $\lambda p.(p\ ?\ ?)$

Boolean functions

not is a function that negates its argument:

x	not x
$\lambda xy.x$	false
$\lambda xy.y$	true

not: $\lambda p.(p \text{ false true})$

Boolean functions

not is a function that negates its argument:

not false:

- ($\lambda p.p$ false true)(false)
- ($\lambda p.p$ false true)($\lambda xy.y$)
- ($\lambda xy.y$) false true
- true

not true:

- ($\lambda p.p$ false true)(true)
- ($\lambda p.p$ false true)($\lambda xy.x$)
- ($\lambda xy.x$) false true
- false

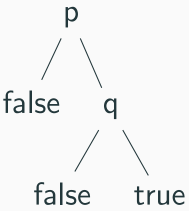
Boolean functions

How about and?

p	q	p and q
true	true	true
true	false	false
false	true	false
false	false	false

Boolean functions

As a decision tree, with false being the left branch, and true being the right:



Boolean functions

How about and?

```
and p q = if p
          then if q
               then true
               else false
          else false
```

Boolean functions

How about and?

```
and p q = if p
          then q
          else false
```

Boolean functions

and: $\lambda p.\lambda q.(p\ ?\ ?)$

Boolean functions

and: $\lambda p.\lambda q.(p\ q\ \text{false})$

Church numerals

- zero: $\lambda f.\lambda x.x$
- one: $\lambda f.\lambda x.fx$
- two: $\lambda f.\lambda x.f(fx)$

Church numerals

The successor function, succ: $\lambda n.\lambda f.\lambda x.f(nfx)$

- succ zero
- $(\lambda nfx.f(nfx))(\lambda fx.x)$
- $(\lambda fx.f((\lambda fx.x)fx))$
- $(\lambda fx.fx)$
- one

Church numerals

pred: $\lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$

- pred one
- $(\lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u))(\lambda fx.fx)$
- $(\lambda fx.(\lambda fx.fx)(\lambda gh.h(gf)))(\lambda u.x)(\lambda u.u)$
- $(\lambda fx.(\lambda x.(\lambda gh.h(gf))x))(\lambda u.x)(\lambda u.u)$
- $(\lambda fx.(\lambda x.\lambda h.h(xf)))(\lambda u.x)(\lambda u.u)$
- $(\lambda fx.(\lambda h.h((\lambda u.x)f)))(\lambda u.u)$
- $(\lambda fx.(\lambda h.hx))(\lambda u.u)$
- $\lambda fx.(\lambda u.u)x$
- $\lambda fx.x$