Turns out, we've alredy encoded enough to build lists at a higher level! Just use bools and pairs!

nil = mkpair false false

cons = λ h. λ t. mkpair true (mkpair h t)

is-empty = fst

head = λ l. fst (snd l)

tail = λ l. snd (snd l)

Note: tail nil does weird stuff, but so does following null → next in C / Java etc...
(kinda how list work at asm lvl)

Doing Good! bools, pairs, list ☺

NUMBERS "Church Numerals"

$0 = \lambda s. \lambda z. z$

$1 = \lambda s. \lambda z. s z$

$2 = \lambda s. \lambda z. s (s z)$

$3 = \lambda s. \lambda z. s (s (s z))$

...

- Numbers encoded wit 2-arg funcs
- "i" composes its first arg w/ itself i times, starting w/ its second arg
- "s" stands for successor
- "z" stands for zero
- implement arithmetic by cleverly passing in right thing for s and z

EG:
$\lambda n.\lambda s.\lambda z.\ n\ s\ z$

SUCC = $\lambda n.\ \lambda.s\ \lambda z.\ s\ (n\ s\ z)$

- take a "number" & return a number that applies its first arg one more time

$\text{plus} = \lambda n. \lambda m. \lambda s. \lambda z. \; n \; s \; (m \; s \; z)$

- ▷ take two numbers and use the 2^{nd} as the "zero" for the first
- ▷ result = a function which applies its 1^{st} arg $n+m$ times to its 2^{nd} arg

$\text{times} = \lambda n. \lambda m. \; m \; (\text{plus} \; n) \; \text{zero}$

- ▷ take two numbers and use "plus n" as the "succ" in the first.
- ▷ (adds n) m times to zero

$\text{is\_zero} = \lambda n.\ n\ (\lambda x.\ \text{false})\ \text{true}$

▷ think about it

- We can also do pred, minus, div, is_equal, etc...
- Notice we've done everything w/out loops!

LOOPS (well ... recursion)

OK, almost there, but how do we repeat actions?
Can we write divergent progs? Yes!
How about useful ones? ... let's see...
(turns out doing this clearly in full detail
takes time ... sketch only here)

- write func that takes f and calls that in place of recursion:

[redacted]

  - e.g. if we had a few more features:

$$\lambda f.\ \lambda x.\ \text{if } (x = 0) \text{ then } 1 \text{ else } (x * f(x-1))$$

- now if we could just pass in the func itself, we'd be good...
- next apply "fix" to get a recursive func.

$$\text{fix}\ (\lambda f.\ \lambda x.\ \text{if } (x = 0) \text{ then } 1 \text{ else } (x * f(x-1)))$$

- Fix "passes a function itself"
  - unroll once, further as needed

- full details tricky...

$$Y = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ (x\ x)\ y))$$
$$\qquad (\lambda x.\ f\ (\lambda y.\ (x\ x)\ y))$$

$$f_! \triangleq \lambda f.\ \lambda p.\ \lambda n.\ (p\ n)\ n\ (f_{\cancel{!}}\ p\ (n+1))$$

$$Y\ f_! = (\lambda x.\ f_!\ (\lambda y.\ (x\ x)\ y))$$
$$\qquad (\lambda x.\ f_!\ (\lambda y.\ (x\ x)\ y))$$

$$= f_!\ (\lambda y.\ ((\lambda x.\ f_!\ (\lambda y.\ (x\ x)\ y))$$
$$\qquad (\lambda x.\ f_!\ (\lambda y.\ (x\ x)\ y))))$$

$$= f_!\ (Y\ f_!)$$
$$= \lambda p.\ \lambda n.\ (p\ n)\ n\ ((Y\ f_!)\ p\ (n+1))$$

With fix we can do all sorts of stuff easier:

- implement numbers as lists of bools (0 - nil
  ... binary!)
- append
- ... everything!

Any problems w/ CBV?

- what about:

$$\text{if true } (\lambda x.\, x)\ ((\lambda x.\, x\ x)(\lambda x.\, x\ x))$$

- may need "thunk"

$$\text{if true } (\lambda x.\, x)\ (\underbrace{\lambda z.}_{\text{delay eval}}\ ((\lambda x.\, x\ x)(\lambda x.\, x\ x)))$$

We can even do lets!

$$\text{"let } x = e_1 \text{ in } e_2\text{"} \approx (\lambda x.\ e_2)\ e_1$$

Turing Complete? Heck yes!

Next: more reduction strategies, types

Reduction Strategies, Substitution

Whew! We've done a lot!

λ-calculus **syntax**:

$$e ::= \lambda x.e \mid e\ e \mid x$$
$$v ::= \lambda x.e$$

CBV, λ-to-λ semantics:

$$\boxed{e \to e'}$$

$$\frac{}{(\lambda x.e)v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

Last time we saw the first axiom as:

$$\frac{e[v/x] = e'}{(\lambda x.e)\,v \longrightarrow e'}$$

- ▷ more common, but...
- ▷ the verbose version will be handy later

Other Reduction Strategies

What if we tweak semantics to throw out order?

$\boxed{e \rightarrow e'}$

$$\frac{}{(\lambda x.e)e' \longrightarrow e[e'/x]} \qquad \frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2} \qquad \frac{e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1 e_2'} \qquad \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'}$$

This is sort of weird for a PL to do,
but provides some advantages:

- optimizations / partial evaluation
- flexibility in equivalence proofs

The order in which you evaluate expressions is called a "reduction strategy."

<u>Amazing fact</u> (Church-Rosser Theorem)

In our order-less semantics,
if $e \xrightarrow{*} e_1$ and $e \xrightarrow{*} e_2$,
then $\exists\ e_3$ s.t. $e_1 \xrightarrow{*} e_3$ and $e_2 \xrightarrow{*} e_3$.

What does this mean?

- "no strategy gets painted into a corner"
- we're never irredeamable if want to get a v

- can't pick a "wrong way" to go about eval

In general, any rewriting system with this property is said to have the "Church-Rosser property".

How would you prove this? ...

Equivalence via Rewriting

Let's add a couple more rules. They're often super convenient.

1. Replace $\lambda x. e$ with $\lambda y. e'$ where e' is e with all "free" x replaced by y. (assume y not already used in e)

$$\frac{}{\lambda x. e \longrightarrow \lambda y. e[y/x]}$$

2. Replace $\lambda x. e\ x$ with e if x does not occur "free" in e.

$$\frac{\text{"}x \text{ not free in } e\text{"}}{\lambda x. e\ x \longrightarrow e}$$

~ if e then true else false $\eqsim$ e

~ List.map (fun x → f x) l $\eqsim$ List.map f l

* Careful of side effects / non-term in CBV!

That's all folks!

With all these rules, plus ability to run them "backward" (rewrite right side to left), we can show anything that can be shown.

Astonishing:

Under natural denotational semantics (basically treat lambdas as functions), e and e' denote the same thing iff this rewriting can show $e \xrightarrow{*} e'$

- So our rules are sound, meaning the respect the semantics

- Our rules are complete, meaning we never have to add more rules to show a true equivalence.

(Good b/c natural denotational semantics for λ-calc isn't so convenient or natural!)

(Need set D isomorphic to $D \rightarrow D$!)

So: to decide if expressions equiv, just search via rules!

Just gave algo for prog equivalence ?! No!

- can't tell when to stop search

Other popular semantics

Seen: "full reduction", l-to-r CBV

Claim: w/out assignment (mutation), I/O, exceptions, … (in short: effects) you cannot distinguish l-to-r CBV from r-to-l CBV.

▷ How would you prove?

- Rember? we did equiv proofs for semantics! (couple lectures back)

"call by name" (CBN)

* even smaller than CBV!

$$\boxed{e \longrightarrow e'}$$

$$\frac{}{(\lambda x.e)e' \longrightarrow e[e'/x]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$$

Note [illegible] diverge strictly less often than CBV

Why? Only evals on demand.

But may take more steps.

Why? Re-evaluates args!

If our ~~code~~ lang has no effects, then order only matters for "performance" (# of steps) and termination.

Imagine if OCaml "if" was CBV!

```
let rec f n =
  if n < 1 then
     1
  else
     n * f (n-1)
```

Another Strategy

"Call By Need"

- also "lazy" evaluation
- only eval an arg the first time it's used, then remember & reuse result
    - like CBN, but doesn't re-eval args

Best of Both Worlds?

- for pure code, asymptotically no slower than CBV
- side effects get tricky

Haskell uses Call-By-Need
(Haskell programmers are lazy)

example:

```
x = factorial 20
y = x + x
```

example:

```
ones = 1 : ones
foo = take 10 ones
```

Also: roll your own control flow

One little detail

- still haven't nailed down substitution
  - used in the rule for call
- how hard can it be?
  - surprisingly, quite subtle
  - sort of where we hid all the complexity...

Informally: $e[e'/x]$ "replaces each x in e w/ e'"

Examples:

$$x[(\lambda y.\, y)/x] = \lambda y.\, y$$

$$(\lambda y.\, y\ x)[(\lambda z.z)/x] = \lambda y.\, y\ (\lambda z.\, z)$$

$$(x\ x)[(\lambda x.\, x\ x)/x] = (\lambda x.\, x\ x)(\lambda x.\, x\ x)$$

Let's take a crack at this...

$$\boxed{e_1[e_2/x] = e_3}$$

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y}$$

$$\frac{e_1[e/x] = e_1'}{(\lambda y . e_1)[e/x] = \lambda y . e_1'} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

"Recursively replace every x leaf with e."

<u>Wrong</u> for nested functions, if the inner function binds same var as outer func. (shadows).

consider: $(\lambda x . \lambda x . x)\ 42$

yikes!

OK, let's try again...

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{e_1[e/x] = e_1' \quad \boxed{y \neq x}}{(\lambda y.\, e_1)[e/x] = \lambda y.\, e_1'}$$

stop early

$$\boxed{\frac{}{(\lambda x.\, e_1)[e/x] = \lambda x.\, e_1}} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

don't overwrite

- Respect shadowing: stop when you hit binder for var you're substituting.
- Still wrong! If a function body e uses an "outer" y, these rules will <u>capture</u> it.

$(\lambda x.\lambda y.x)(\lambda z.y) \overset{\ddot{\frown}}{\longrightarrow} \lambda y.(\lambda z.y)$

$$\not\equiv$$

$(\lambda a.\lambda b.a)(\lambda z.y) \rightarrow \lambda b.(\lambda z.y)$

(doesn't happen in CBV/CBN if there are no free vars, but can arise under full reduction)

Free Variables

Need to know what's bound:

$FV(x) = \{x\}$

$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

$FV(\lambda x.e) = FV(e) - \{x\}$

Now.

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{e_1[e/x] = e_1' \quad y \neq x \quad y \notin FV(e)}{(\lambda y.e_1)[e/x] = \lambda y.e_1'}$$

$$\frac{}{(\lambda x.e_1)[e/x] = \lambda x.e_1} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

OK... but could get stuck (no rule applies)

Implicit Renaming

- our defn only partial if y "accidentally" used as a binder
  - whole point was we don't care how local vars are named!

- To handle this, we allow implicit renaming of a binding & all its occurrences

- By renaming, the $y \neq x$ rule can always apply
  - we can lose x shadowing rule

- Generally, never distinguish between terms based on how their vars are named
  - Key design principle
  - different ASTs considered the same!

Assume implicit systematic renaming, then

$$\boxed{e_1[e_2/x] = e_3}$$

$$\frac{}{x\,[e/x] = e} \qquad \frac{y \neq x}{y\,[e/x] = y} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

$$\frac{e_1[e/x] = e_1' \quad y \neq x \quad y \notin FV(e)}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'}$$

Notoriously annoying problem in PL
- google "capture avoiding substitution"

Can implement w/ this verbose rule too:

$$\frac{z \neq x \quad z \notin FV(e_1) \quad z \notin FV(e) \quad e_1[z/y] = e_1' \quad e_1'[e/x] = e_1''}{(\lambda y . e_1)[e/x] = \lambda z . e_1''}$$

- always find some z
  - global counter in compiler

Jargon

- implicit renaming called "α-conversion"

- $(\lambda x.e_1)\,e_2 \rightarrow e_1[e_2/x]$ called "β-reduction"

- $\lambda x.e\,x \rightarrow e$ called "η-reduction"
    - reverse calle "η-expansion"
    - delays eval in CBV
    -