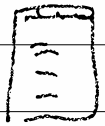
1-1

Goal: Implement a VM

What does a program mean?

"1+1" means "2"

"word.c" means 

a semantics — english (human)
— math (universal)
— code

$\forall x,y \in \mathbb{N}.\ x + y = y + x$

BNF or CFG

$\mathcal{J}_0$: e := v | (+ e e) | (* e e)

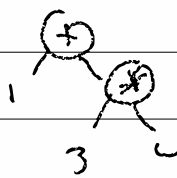
V := number

"(+ 1 (* 3 4))" $\in \mathcal{J}_0.e$

"(+ 1))"

e := v or  or 

V := number



1-2/  = new Add(new Num(1),

class E {} new Mult(

class Add : E {} new Num(3),

class Num : E {} new Num(4))

class Mult : E {}

```
Add::Add(E*l, E*r) {
    this.l = l; this.r = r; }
```

(+ 1 (* 3 4))

Semantics : meaning of programs

interpreter : a program in ~~the~~ language M that tells you the semantics of language O

virtual machine : a fast interpreter we like

interp (u big step semantics) : e → v

interp v = v

interp (+ e_1 e_2) = (interp e_1) $+_v$ (interp e_2)

$+_v$ (Num n_1) (Num n_2) = Num ($n_1 + n_2$)

1-3/ E^* ← Num* ≈ V*

```
virtual E::interp() = 0;
// interp v = v
E* Num::interp() { return this; }
// interp (+ e1 e2) = (interp e1) +v (interp e2)
E* Add::interp() {
Num* n1 = this.l.interp();
Num* n2 = this.r.interp();
return new Num(n1.n + n2.n); }
```

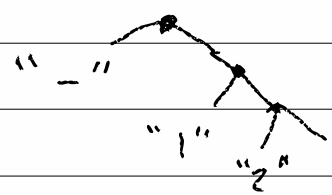
(- e1) = (* -1 e1)

(- e1 e2) = (+ e1 (- e2))

desugar (expander) [compiler] : string tree
string expr
S-expr
Sexpr

Sexpr = string or pair of sexpr or empty

⟹ J_0

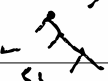


= (- 1 2)

1 - 2

2-1/ Sexpr J_0

desugar $(- e_1) = (* -1 \text{ (desugar } e_1))$



$(- e_1\ e_2) = (+ (d\ e_1)\ (d\ (- e_2)))$

$(+) = 0$

$(+ e_1\ .\ more) = (+ (d\ e_1)\ (d\ (+\ .\ more)))$

$(*) = 1$

$(* e_1\ .\ more) = (* (d\ e_1)\ (d\ (*\ .\ more)))$

J_1 e := v | (if e e e) | (e e ...)

v := number | bool | prim

prim := + | * | / | - | ≤ | < | = | > | ≥

interp v = v

interp (if e_c e_t e_e) =

c = interp e_c

e_k = if c e_t else e_f

return interp e_k

interp (e_p e_0 ... e_n) =

p = interp e_p (must be a prim)

v_0 ... v_n = interp v_0 ... interp v_n

ret $\delta(p, v_0 \ldots v_n)$

2-2/

```
vs delta (prim p, list vs) =
  if (p == ADD)
    return new Num(vs[0].n + vs[1].n)
  if (p == LT)
    return new Bool(vs[0].n < vs[1].n)
```

---

big-step has a big problem : $e \Rightarrow v$

- it is partial
- it says nothing about "in between"
- very un-math-like way of proving
- inefficient / unhelpful for implementations

small step : $e \to e$

step (if true e_t e_f) = e_t

step (if false e_t e_f) = e_f

step (p v_0 ... v_n) = $\delta(p, v_0 \ldots v_n)$

if step $e_c = e_c'$ then sstep (if e_c e_t e_f) = (if e_c' e_t e_f)

if step $e_i = e_i'$ then step (e_0 .. e_i e_{i+1} ... e_n) = (e_0 .. e_i' e_{i+1} ... e_n)

2-3)

$(1+1) + (2+3) \overset{\text{subeo}}{=} 2 + (2+3) \overset{\text{step}}{=} 2 + 5 \overset{\text{step}}{=} 7$

$= (1+1) + 5$

A context, C ::= hole

| if C e e

| if e C e

| if e e C

| (e ... C e ...)

plug : C × e ⇒ e

plug hole e = e

plug (if C e_1 e_2) e_p = (if e_p e_1 e_2)

plug (e_b ... C e_a ...) e_p = (e_b ... (plug C e_p) e_a ...)

plug (+ (2+3)) (1+1) = (1+1) + (2+3)

C 2 = 2 + (2+3)

parse : e ⇒ C × e

2-4/

$e \quad \Rightarrow \quad e$

step $C[\text{if true } e_t \; e_f] = C[e_t]$

step $C[\text{if false } e_t \; e_f] = C[e_f]$

step $C[p \; v_0 \ldots v_n] = C[\delta(p, v_0 \ldots v_n)]$

find_redex : $e \Rightarrow C \times e$

↑ redex

reducible expression

When do two programs mean the same thing?

"1 + 2" "2 + 1"

"4 billion + 1 + 2" "2 + 1 + 4 billion"

$\forall C =$ [redacted]

"quicksort" "mergesort" "heapsort"

"insertionsort"

$\forall l.$ "mergesort l" = "hs l"

$C = ([\text{redacted}] \; L)$

$\forall C. \; C[\underset{\text{interp}}{x}] = C[\underset{\text{interp}}{y}]$ — observational equivalence

time e = e's ans x secs

3-1/ step : e ⇒ e ⇒⇒⇒ v

C[if true e+ ef] → C[et]

.... + ... * ... − .... (1+1) .... * ... +

C := hole | if C e e | if e C e | if e e C |
(e ... C e ...)

evaluation context, E

E := hole | if E e e | (v ... E e ...)
Ehole Eif Eapp

E[if true e+ ec] → E[e+]
E[if false e+ ef] ⇒ E[ef]
E[(p v0 ... vn)] → E[δ(p, v0 ... vn)]

find-redex : e ⇒ (E, e) or #false

find-redex v = false

fr (if e0 e1 e2) = case (fr e0) with
#false ⇒ (hole, (if e0 e1 e2))
(E, e0') ⇒ (if E e1 e2, e0')

3-2/

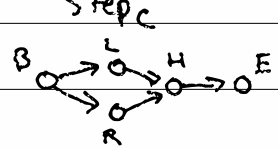
fr (v ... e_0 e_1 ... e_n) =

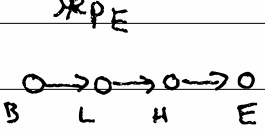
(E, e_0') = fr e_0

((v ... E e_1 ... e_n) , e_0')

new EApp(v, E, e_1 .. e_n)

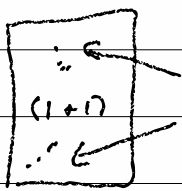
fr (v ...) = (hole, (v ....))





The Standard Reduction theorem

Alonzo Church ((Church-Rosser) Rosser



E = (if (* 7 8 (+ 1 2 (* ....

..... (if ... hole ..)^(434) ...))

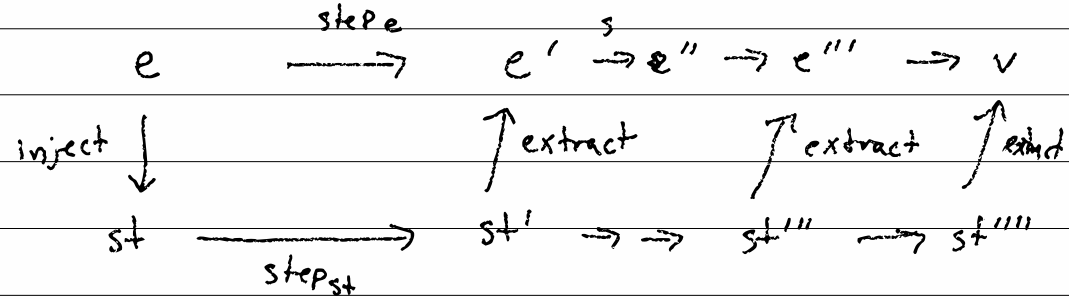
E[(1+1)] --> E [2]

2n time, in space

2n time, in space

3-3/ $step_E : e \Rightarrow e$

machine semantics



CC0 $st = \langle e, E \rangle$

inject $e = \langle e, hole \rangle$

extract $\langle e, E \rangle = E[e]$

done? $\langle v, hole \rangle = true$

1 $\langle if\ e_c\ e_t\ e_f, E \rangle \Rightarrow \langle e_c, E[if\ hole\ e_t\ e_c] \rangle$

2 $\langle true, E[if\ hole\ e_t\ e_c] \rangle \Rightarrow \langle e_t, E \rangle$

3 $\langle false, E[if\ hole\ e_t\ e_f] \rangle \Rightarrow \langle e_f, E \rangle$

4 $\langle e_0\ e_1 \cdots e_n, E \rangle \Rightarrow \langle e_0, E[(hole\ e_1 \cdots e_n)] \rangle$

5 $\langle v, E[v_0 \cdots hole\ e_0\ e_1 \cdots] \rangle \Rightarrow \langle e_0, E[(v_0 \cdots v\ hole\ e_1 \cdots)] \rangle$

6 $\langle v_n, E[p\ v_0 \cdots hole] \rangle \Rightarrow \langle \delta(p, v_0 \cdots v_n), E \rangle$

$(y_n, z_n) \longrightarrow (l_n, l_n)$

3-4/

HL: (test '(+ 1 (* 2 (if true 3 4))) 7)

test (new Sexpr (new Atom("+"),
new Se(new A("1"),
....
),
new Num (7))

test (se, ex-val) ==
e = desugar se
big-step eval (e) = actual-big-val
if (abv != ev) { error }
small-step eval (e) = ac - sm - v
(if (asv != ev) { error }
cc-eval e = accv
if (accv != ev) { error }
ll-eval e = allv
if (allv != ev) { error }

3-5/

low level--eval e =

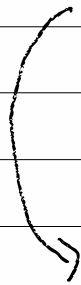
print e as C constructors into "x.c"

compile "x.c" into "x.bin"

run "x.bin" and capture output

parse output

return value



cc x.c ll.c -o x.bin

4-1) $CC_0 \rightarrow CK_0$

$st = \langle e, k \rangle$

$k = kret$ // hole

continuation / kontinuation / stack k

$kif\ e\ e\ k$ // if E e e

$kapp\ (v \ldots)\ (e \ldots)\ k$ // (v ... E e ...)

$A \rightarrow kapp\ (v \ldots)\ (e \ldots)\ B$

$\Longleftrightarrow \quad B[(v \ldots\ A\ e \ldots)]$

$inject\ e = \langle e, kret \rangle$

$extract\ \langle e, k \rangle = kintoE(k)[e]$

$done?\ \langle v, kret \rangle = true$

$\langle if\ e_c\ e_t\ e_f, k \rangle \mapsto \langle e_c, kif(e_t, e_f, k) \rangle$

$\langle true, kif(e_t, e_f, k) \rangle \mapsto \langle e_t, k \rangle$

$\langle false, kif(e_t, e_f, k) \rangle \mapsto \langle e_f, k \rangle$

$\langle (e_0\ e_1 \ldots), k \rangle \mapsto \langle e_0, kapp((), (e_1 \ldots), k) \rangle$

$\langle v, kapp(v_0 \ldots, e_0\ e_1 \ldots, k) \rangle \mapsto \langle e_0, kapp(v_0 \ldots v, e_1 \ldots, k) \rangle$

$\langle v_n, kapp(p\ v_0 \ldots, (), k) \rangle \mapsto \langle \delta(p, v_0 \ldots v_n), k \rangle$

$\delta(SUB, (v_1\ v_0)) = v_0 - v_1$

(3 4 5)

5-1/ $J_1 \to J_2$

$e ::= v \mid (e\ e \ldots) \mid (\text{if } e\ e\ e) \mid x$

$x ::=$ variable names

$v ::= b \mid f$ ← new

$b ::=$ number | bool | prim

$f ::=$ function names

$p ::= (\text{program } (d \ldots)\ e)$

$d ::= (\text{define } (f\ x \ldots)\ e)$

```
(program (define (add1 x) (+ 1 x))
         (add1 5))
```

$E ::= \text{hole} \mid (\text{if } E\ e\ e) \mid (v \ldots E\ e \ldots)$

J_1 : step : $e \to e$

J_2 : step : $\Delta \times e \to e$ (with $(f \mapsto d)$)

$$E[(f\ v_0 \ldots)] \mapsto E[e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n]]$$

where $\Delta(f) = (\text{define } (f\ x_0 \ldots x_n)\ e)$

S-2/ $e[x \leftarrow v]$ means look inside of e, find all the x_s and replace with v

$x[x \leftarrow v] = v$

$y[x \leftarrow v] = y \quad (y \in \overset{\text{vari}}{X})$

$u[x \leftarrow v] = u \quad (u \in \text{values})$

$(\text{if } e_1\ e_2\ e_3)[x \leftarrow v] = (\text{if } e_1[x \leftarrow v]\ e_2[x \leftarrow v]\ e_3[x \leftarrow v])$

$(e_0 \ldots e_n)[x \leftarrow v] = (e_0[x \leftarrow v] \ldots e_n[x \leftarrow v])$

$f(x) = 7x + 2x^2 + 1$

$f(5) = 7 \cdot 5 + 2 \cdot 5^2 + 1$

```
(define (f x y) (+ (* x 2) (* -1 y)))
(define (g x) (f x x)
(+ 5 (g 10) (f 9 (g 1)))          = e
(  "  (f 10 10)   "   )
(  "  (+ 10·2 -10)  "   )
(+ 5     10        "   )
(+ 5     10     (f 9 1))
(                (+ 9·2 -1))
(+ 5     10        17)
```

32

5-3/ $CK_0 \Rightarrow CK_1$ $st = \langle \Delta, e, k \rangle$

$(1+N)$

$\langle \Delta, v_n, kapp((f\ v_0 \ldots), c), k \rangle$

$\mapsto \langle \Delta, e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n], k \rangle$ ^{N}

where $\Delta(f) = (define\ (f\ x_0 \ldots x_n)\ e)$

$\langle \Delta, (if\ e_c\ e_t\ e_f), k \rangle \mapsto \langle \Delta, e_c, kif(e_t, e_f, k) \rangle$

(if (x > 10 mi) (error) (+ 1 (f x)))

(define (f x) (f x))

(f 0)

$\mapsto$ (f 0) $\overset{1}{\mapsto}$ (f 0) $\overset{2}{\mapsto}$

"Jay"

proper fucton call implementation

"Guido"

tail-call optimization

6-1/ CK_1 is linear-time, so it's not fast !!
and unrealistic because syntax is available
at run-time

goal: machine with constant function calls

$\mapsto \quad \langle \Delta, e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n], k \rangle$

"x" $\quad f(x) = 2x^2 + 5x + 7$

$\langle f(5), \rho \rangle \mapsto \langle 2x^2 + 5x + 7, [x \leftarrow 5] \rangle$

$\langle 2 \cdot 5^2 + 5x + 7, [x \leftarrow 5] \rangle$

$\langle 50 + 5 \cdot 5 + 7, [x \leftarrow 5] \rangle$

$\langle 82, [x \leftarrow 5] \rangle \quad = 82$

6-2) $CK_1 \Rightarrow CEK_0$

$st := \langle \Delta, e, env, k \rangle$

$env := mt \mid env[x \leftarrow v]$

Jay's envs: $e := make\text{-}v\text{-}empty \mid make\text{-}v\text{-}cons(make\text{-}v\text{-}cons(x, v), env)$

$inject^{\Delta}\, e = \langle \Delta, e, mt, knet \rangle$

$extract\ \langle \Delta, e, env, k \rangle = E(k)[e[env\ replace]]$

$done?\ \langle \Delta, v, env, knet \rangle$

!!! ↓ WRONG ↓ !!! (look ahead ?pgs)

$\langle \Delta, x, mt, k \rangle \mapsto error$

$\langle \Delta, x, env[x \leftarrow v], k \rangle \mapsto \langle \Delta, v, \overset{env[x \leftarrow v]}{\underset{or}{mt}}, k \rangle$

$\langle \Delta, x, env[y \leftarrow v], k \rangle \mapsto \langle \Delta, x, env, k \rangle$

$\langle \Delta, (if\ e_t\ e_s\ e_f), env, k \rangle \mapsto$

$\langle \Delta, e_c, env, kif(e_t, e_f, k) \rangle$

$\langle \Delta, true, env, kif(e_t, e_f, k) \rangle \mapsto \langle \Delta, e_t, env, k \rangle$

$\langle \Delta, false, env, kif(e_t, e_f, k) \rangle \mapsto \langle \Delta, e_f, env, k \rangle$

$\langle \Delta, (e_0\ e_1 \ldots), env, k \rangle \mapsto \langle \Delta, e_0, env, kapp((), (e_1 \ldots), k) \rangle$

$\langle \Delta, v_n, env, kapp((v_0 \ldots), (e_0\ e_1 \ldots), k) \rangle$

$\mapsto \langle \Delta, e_0, env, kapp((v_0 \ldots v_n), (e_1 \ldots), k) \rangle$

$\langle \Delta, v_n, env, kapp((p\ v_0 \ldots), (), k) \rangle \mapsto \langle \Delta, \delta(p, (v_0 \ldots v_n)), env, k \rangle$

$\langle \Delta, v_n, env, kapp((f\ v_0 \ldots), (), k) \rangle \mapsto$ let (define (f x_0 ..) e) = $\Delta[f]$

$\langle \Delta, e, env[x_0 \mapsto v_0] \ldots, k \rangle$

6-3)
```
(define (f x) (+ x y))
(define (g y) (f 5))
(g 10)
```

$(g\ 10) \rightarrow (f\ 5) \rightarrow (+\ 5\ y)$

$\langle (g\ 10), \emptyset \rangle \mapsto \langle (f\ 5), mt[y \leftarrow 10] \rangle$
$\mapsto \langle (+\ x\ y), mt[y \leftarrow 10][x \leftarrow 5] \rangle$
$\mapsto \langle (+\ 5\ y), \text{''} \rangle$
$\mapsto \langle (+\ 5\ 10), \text{''} \rangle$
$\mapsto \langle 15, \text{''} \rangle$

WRONG:

$\langle \Delta, (\text{if } e_c\ e_t\ e_f), env; k \rangle$
$\mapsto \langle \Delta, e_c, env; kif(e_t, e_f, k) \rangle$
$\langle \Delta, true, env'; kif(e_t, e_f, k) \rangle$
$\mapsto \langle \Delta, e_t, env'; k \rangle$

```
(define (g y) true)
(define (f y) (if (g 10) y 6))
(f 6)
```

Dynamic Scope

6-4/ $k := \text{kret} \mid \text{kif}(e, e, env, k)$

$\mid \text{kapp}((v \ldots), (e \ldots), env, k)$

RIGHT

$\langle \Delta, x, env, k \rangle \mapsto \langle \Delta, env(x), mt, k \rangle$

$\langle \Delta, \text{if } e_c \ e_t \ e_f, env, k \rangle \mapsto \langle \Delta, e_c, env, \text{kif}(e_t, e_f, env, k) \rangle$

$\langle \Delta, \text{true}, \overset{\text{NEW}}{env'}, \text{kif}(e_t, e_f, \overset{\text{OLD}}{env}, k) \rangle \mapsto \langle \Delta, e_t, \overset{\text{OLD}}{env}, k \rangle$

$\langle \Delta, \text{false}, \_\_, \text{kif}(e_t, e_f, env, k) \rangle \mapsto \langle \Delta, e_f, env, k \rangle$

$\langle \Delta, (e_0 \ e_1 \ldots), env, k \rangle \mapsto \langle \Delta, e_0, env, \text{kapp}(\ (), (e_1 \ldots), env, k) \rangle$

$\langle \Delta, v_n, \_\_, \text{kapp}((v_0 \ldots), (e_0 \ e_1 \ldots), env, k) \rangle$

$\mapsto \langle \Delta, e_0, env, \text{kapp}((v_0 \ldots v_n), (e_1 \ldots), env, k) \rangle$

$\langle \Delta, v_n, \_\_, \text{kapp}((p \ v_0 \ldots), (\ ), \_\_, k) \rangle$

$\mapsto \langle \Delta, \delta(p, (v_0 \ldots v_n)), mt, k \rangle$

$\langle \Delta, v_n, \_\_, \text{kapp}((f \ v_0 \ldots), (\ ), \_\_, k) \rangle$

$\mapsto \langle \Delta, e, mt[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n], k \rangle$

where $\Delta(f) = (\text{define } (f \ x_0 \ldots x_n) \ e)$

6-5) $J_2 \Rightarrow J_3$

$e := v \mid (e\ e \ldots) \mid (\text{if } e\ e\ e) \mid x$

$v := b \mid (\lambda\ (x \ldots)\ e)$

$x :=$ variable names

$b :=$ number | bool | prim

$E[((\lambda\ (x_0 \ldots x_n)\ e)\ v_0 \ldots v_n)]$

$\mapsto E[\ e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n]]$

capture-avoiding

$((\lambda\ (x)\ x)\ 5) \mapsto 5$

$(((\lambda\ (x)\ (\lambda\ (y)\ x))\ 5)\ 6) \mapsto$

$((\lambda\ (y)\ 5)\ 6) \mapsto 5$

$(((\lambda\ (f)\ (\lambda\ (y)\ (f\ 5)))\ (\lambda\ (x)\ y))\ 6)\ 7) \mapsto$

$(((\lambda\ (y)\ (\lambda\ (x)\ y))\ 6)\ 7) \mapsto$

$((\lambda\ (x)\ 6)\ 7) \mapsto 6$

6-6/

$\langle v_n, \overset{new}{env}, kapp((\lambda\ (x_0 \cdots x_n)\ e)\ v_0 \cdots), (), \overset{old}{env}, k\rangle$
$\mapsto \langle e, mt[x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n], k\rangle$

$\langle((\lambda x.\lambda y.x)\ 5)\ 6, \emptyset, kret\rangle$
$\langle \lambda x.\lambda y.x, \emptyset, kapp((), (5), \emptyset, kret)\rangle$
$\langle 5, \emptyset, kapp((\lambda x.\lambda y.x), (), \emptyset, kapp((), (6), \emptyset, kret))\rangle$
$\langle \lambda y.x, \emptyset[x \leftarrow 5], kapp((), (6), \emptyset, kret)\rangle$
$\langle 6, \emptyset, kapp((\lambda y.x), (), \emptyset, kret)\rangle$
$\langle x, \emptyset[y \leftarrow 6], kret\rangle$ (old)
$\langle x, \emptyset[x \leftarrow 5][y \leftarrow 6], kret\rangle$
$\langle 5, mt, kret\rangle \mapsto 5$

$(((\lambda x.\lambda y.x)\ 5)\ 6) \mapsto (\lambda y.5)\ 6 \mapsto 5$

CEK

old $v ::=$ num | bool | prim | $(\lambda\ (x \cdots)\ e)$

new $v ::=$ num | bool | prim | closure$((\lambda (x \cdots) e), env)$

$\langle (\lambda (x \cdots) e), env, k\rangle \mapsto \langle$ closure$((\lambda (x \cdots) e), env), \emptyset, k\rangle$

$\langle v_n, —, kapp(($ closure$((\lambda (x_0 \cdots x_n) e), env), v_0 \cdots), (), k\rangle$
$\mapsto \langle e, env[x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n], k\rangle$

6-7/

desugar (let ([x0 e0] ... [xn en]) be)

= ((λ (x0 ... xn) be) e0 ... en)

(let ([x 5] [y 6])
  (+ x y))

=

((λ (x y) (+ x y))
 5 6)

(let ([x←5])
  (+ x
     (let ([x (+ 1 x)]) (+ x x))
     (+ x 4)))

= (+ 5
     (let ([x←(+ 1 5)]) (+ x x))
     (+ 5 4))

∅[x←5][x←6]

= (+ 5 (+ 6 6) (+ 5 4))

= (+ 5 12 9)

= 26

6-8/ desugar (let* () be) = be

desugar (let* ([x0 e0] [x1 e1] ...) be)

= (let ([x0 e0])
    (let* ([x1 e1] ...) be))

(let* [[x ← 5]
       [y (+ x 1)]
       [z (+ x y)]]
  (+ z z))

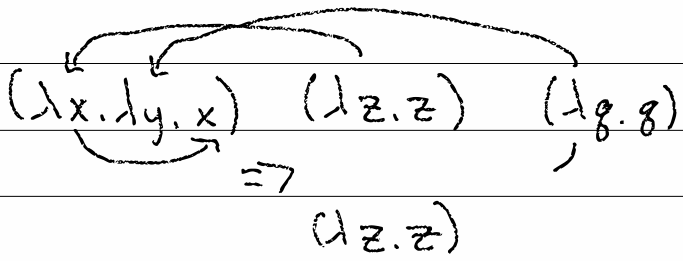
normal let: ((λ (x y z) (+ z z))
  5 (+ x 1) (+ x y))

let*: ((λ (x) ((λ (y) ((λ (z) (+ z z)) (+ x y))) (+ x 1))) 5)

7-1/ ((let [x <- 5])
(λ (y) (+ x y)))
6) => (+ 5 6)

Λ : e = x | λx. e | e e

id : λx. x



Alonzo Church's Lambda Calculus

Church encoding == Object-oriented Prog.

represent as interface interface vs representation

bool : Opt 2 -> Opt 2 -> Some Option 1 or 2

true := λx. λy. x

false := λx. λy. y

if := λb. λt. λf. b t f

if true M N => (λbtf. b t f) true M N

=> true M N => (λxy. x) M N => M

7-2/ Church-encoded numbers

numbers are iteration

num : Thing To Do ⇒ Something to do it to
⇒ Result of doing it N times

zero := $\lambda f.\ \lambda z.\ z$

one := $\lambda f.\ \lambda z.\ f\ z$

two := $\lambda f.\ \lambda z.\ f\ (f\ z)$

add1 := $\lambda n.\ \lambda f.\ \lambda z.\ f\ (n\ f\ z)$

add1 zero ⇒ $\lambda f.\lambda z.\ f\ (zero\ f\ z)$
⇒ $\lambda f.\lambda z.\ f\ z$ = one

plus := $\lambda n.\ \lambda m.\ \lambda f.\lambda z.\ m\ f\ (n\ f\ z)$

plus one one ⇒ $\lambda f.\lambda z.$ one f (one f z)
⇒ $\lambda f.\lambda z.$ one f (f z)
⇒ $\lambda f.\lambda z.$ f (f z) = two

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

mult two two ⇒ $\lambda f.\lambda z.$ two (two f) z
$\lambda f.\lambda z.$ (two f) (two f) z
$\lambda f.\lambda z.$ f (f (f (f z))) =
four

7-3/ zero? $= \lambda n.\ n\ (\lambda x.\ false)\ true$

zero? zero $\Rightarrow$ zero $(\lambda x.\ F)\ T$

$\Rightarrow T$

zero? one $\Rightarrow$ one $(\lambda x.\ F)\ T$

$\Rightarrow (\lambda x.\ F)\ T$

$\Rightarrow F$

fst (pair M N) $\Rightarrow$ M

snd (pair M N) $\Rightarrow$ N

pair := $\lambda x.\lambda y.\lambda sel.$ sel x y

fst := $\lambda p.$ p true

snd := $\lambda p.$ p false

fst (pair M N) $\Rightarrow$ (pair M N) true

$\Rightarrow$ (true M N) $\Rightarrow$ M

---

select (inl M) f g $\Rightarrow$ f M

select (inr N) f g $\Rightarrow$ g N

inl := $\lambda v.\ \lambda f.\lambda g.$ f v

inr := $\lambda v.\lambda f.\lambda g.$ g v

select := $\lambda o.\lambda f.\lambda g.$ o f g

copy := λn. n add1 zero

7-4/ sub1 := λn. fst (n F (pair zero zero))

F := λp. pair (snd p) (add1 (snd p))

sub := λn. λm. m sub1 n

fac := λn.

defn ↗ if (zero? n)

one

(mult n (fac (sub1 n)))

copy, not a reference ↙

mkfac := λfac. ←

(λn. if (zero? n) one ref

(mult n (fac (sub1 n))))

fac := Z mkfac

Z mkfac ⇒ fac

Z (λfac. M) ⇒ M [fac ← (Z mkfac)]

Z (λx. M) ⇒ M (Z (λx. M))

← fixed-point combinator (eager) aka Y

Z := λf. ((λx. (f (λv. (x x v))))

(λx. (f (λv. (x x v)))))

Z f = f (Z f)

Ex-5/ $\Omega := \omega\ \omega$

$\omega := \lambda x.\ x\ x$

$\Omega \Rightarrow \omega\ \omega \Rightarrow$

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

$\Rightarrow (\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$

$\Rightarrow \omega\ \omega$

$\Rightarrow \Omega$

The Lambda Calculus Σ J_2

A J_2 to convert Church to Normal :=

Church2Normal := $\lambda n.\ n\ (\lambda x.\ (+\ 1\ x))\ 0$

Church2Normal (fac (succ (plus two two)))

$\Rightarrow$ 120

8-1/ J_2: (define (even? x)
(if (zero? x) true
~~(not~~ (odd? (sub1 x))))
(define (odd? x)
(if (zero? x) false
(even? (sub1 x))))
(even? 10)

Δ = { even? , odd? }

odd? is undefined

J_3: (let* ([even? (λ (x) .... odd? ...)]
[odd? (λ (x) .... even? ...)])
(even? 10))

$J_3 \Rightarrow J_4$ = v := .... | (λ x (x ...) e)

↑ recursive name of fun

(let ([fac (λ ifac (n) (if (= n 0) 1
(* n (ifac (- n 1)))))])
(fac 5))

(desugar (λ (x ...) e)) => (desugar (λ rec (x...) e))

8-2/ OLD:

$$E[\ell\ v_0 \cdots v_n] = E[e[x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n]]$$

where $\ell = (\lambda\ (x_0 \cdots x_n)\ e)$

NEW:

$$E[\ell\ v_0 \cdots v_n] = E[e[f \leftarrow \ell][x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n]]$$

where $\ell = (\lambda\ f\ (x_0 \cdots x_n)\ e)$

$CEK_1 \Rightarrow CEK_2$

OLD: $\langle (\lambda\ (x_0 \cdots x_n)\ e), env, k \rangle \mapsto \langle clo((\lambda\ (x_0 \cdots x_n)\ e), env), mt, k \rangle$

$\langle \ell, env, k \rangle \mapsto \langle clo(\ell, env), mt, k \rangle$

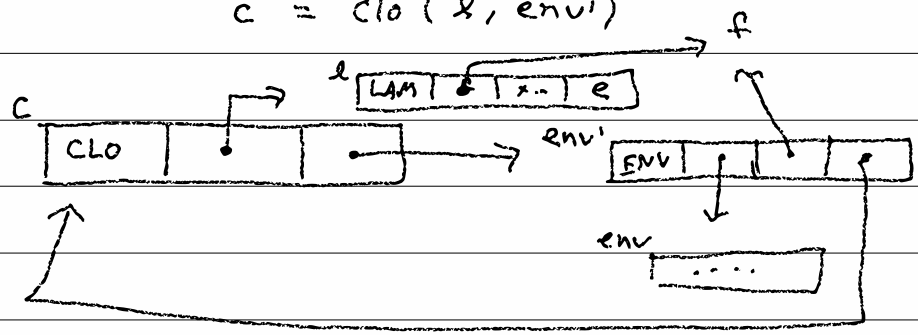
where $\ell = (\lambda\ (x_0 \cdots x_n)\ e)$

NEW: $\langle \ell, env, k \rangle \mapsto \langle\ c\ , mt, k \rangle$

where $\ell = (\lambda\ f\ (x_0 \cdots x_n)\ e)$

$env' = env[f \leftarrow c]$

$c = clo(\ell, env')$



8-3/ envp = make_env_ext (env, l→f, NULL);
clo = make_clo (l, envp);
envp → val = clo; // ← installs cycle!

```
nat-unfold :=
  (λ rec (f z n)
    (if (= n 0) z
        (f n (rec f z (- n 1)))))

fac n = nat-unfold (λ (n a) (* n a)) 1 n
```

9-1/ data structures

$v = \ldots \mid p$ (primitives) $\mid b$ (constants)

Numbers, $7 \in b$, $+ \in p$

Unit: $tt \in b$ // Void void m C

Pairs: pair, fst, snd $\in p$ $(\text{pair } v_1\ v_2) \in V$ $\forall v_1, v_2$

"and"

(pair 1 2)

$E[(\text{pair } v_1\ v_2)] \rightarrow E[(\text{pair } v\ v_1\ v_2)]$

$E[(\text{fst } (\text{pair } v_1\ v_2))] \rightarrow E[v_1]$

$E[(\text{snd } (\text{pair } v_1\ v_2))] \rightarrow E[v_2]$

Variants: "or" List = empty OR node

$v = \ldots \mid (\text{inl } v) \mid (\text{inr } v)$ Unit or (data x list)

$p = \ldots \mid \text{inl} \mid \text{inr}$

$e = \ldots \mid \text{case } e \text{ as } [(\text{inl } x) \Rightarrow e] [(\text{inr } x) \Rightarrow e]$

$E = \ldots \mid \text{case } E \text{ as } [(\text{inl } x) \Rightarrow e] [(\text{inr } x) \Rightarrow e]$

$E[\text{case } (\text{inl } v) \text{ as } [(\text{inl } x_1) \Rightarrow e_1] [(\text{inr } x_2) \Rightarrow e_2])$

$\mapsto E[e_1[x_1 \leftarrow v]]$

9-2/ CEK ...

$k = \ldots \mid \text{casek}\ \text{env}\ x_1\ e_1\ x_2\ e_2\ k$

$\langle \text{case}\ e_s\ \text{as}\ [(\text{inl}\ x_1) \Rightarrow e_1]\ [(\text{inr}\ x_2) \Rightarrow e_2],\ \text{env},\ k \rangle$

$\mapsto \langle e_s,\ \text{env},\ \text{casek}\ \text{env}\ x_1\ e_1\ x_2\ e_2\ k \rangle$

$\langle \text{inl}\ v,\ \_,\ \text{casek}\ \text{env}\ x_1\ e_1\ x_2\ e_2\ k \rangle$

$\mapsto \langle e_1,\ \text{env}[x_1 \leftarrow v],\ k \rangle$

$\langle \text{inr}\ v,\ \_,\ \text{casek}\ \text{env}\ x_1\ e_1\ x_2\ e_2\ k \rangle$

$\mapsto \langle e_2,\ \text{env}[x_2 \leftarrow v],\ k \rangle$

Bool = Unit or Unit

true = inl ()

false = inr ()

if c t f = case c as [(inl _) ⇒ t]
[(inr _) ⇒ f]

X or Y = Pair Boolean (X ∪ Y)

inl v = pair #false v

inr v = pair #true v

case es as [(inl x1) ⇒ e1] [(inr x2) ⇒ e2] =

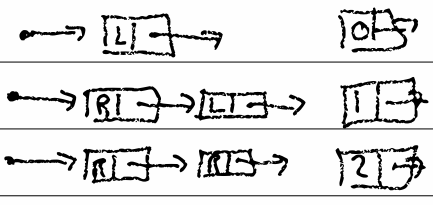
(let ([vs es]) (if (fst vs) (let ([x1 (snd vs)]) e1)
(let ([x2 (snd vs)]) e2)))

9-3/ Shape = (circle or (Rect or Triangle)

○ = inl ....

□ = inr (inl ...)

△ = inr (inr ...)



Algebraic Data Types

Type = 1 // Unit tt

0 // Nothing

Type × Type // Pair (pair e1 e2)

Type + Type // Variant (inl e1) (inr e2)

B // Base types 5

B = Int32 | ...

Bool = 1 + 1

true = inl tt

false = inr tt

Nat = 1 + Nat

0 Zero = inl tt

10 One = inr (inl tt)

110 two = inr (inr (inl tt))

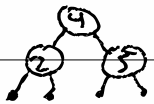
List A = 1 + (A × List A)

mt = inl tt

[1] = inr (pair 1 (inl tt))

[3, 4] = inr (pair 3 (inr (pair 4 (inl tt))))

Tree A = 1 + (Tree A × A × Tree A)



= inr (pair (inr (pair (inl tt) (pair 2 (inl tt))))
(pair 4 (inr (pair (inl tt) (pair 5 (inl tt))))))

q-4/

leaf = inl tt

node t_1 v t_2 = inr (pair t_1 (pair v t_2))

single v = node leaf v leaf

= node (single 2) 4 (single 5)

Option type : Maybe A = None | Just A

1 + A

nothing = inl tt

just v = inr v

hash-ref : Hash K V x K

-> Maybe V

Hash ID (Maybe People) x ID

-> Maybe (Maybe People)

seq : Maybe A x (A -> Maybe B) -> Maybe B

seq ma f = case ma as

[None => None]

[Just a => f a]

seq (hash-ref ht 17)

(λ (x) (hash-ref ht2 x))

9-5/ List A = 1 + Pair A (List A)

empty = inl tt

cons a d = inr (pair a d)

1, 2, 3 = cons 1 (cons 2 (cons 3 empty)))

map f l =

case l as

[(inl t) => empty]

[(inr x) => cons (f (fst x)) (map f (snd x))

map add1 (1, 2, 3) => (2, 3, 4)

filter even? (2, 3, 4) => (2, 4)

fold f z l =

case l as

[empty => z]

[cons a d => f a (fold f z d)]

sum l = fold + 0 l

sum (1, 2, 3) = (+ 1 (+ 2 (+ 3 0)))

map f l = fold (λ (a d) (cons (f a) d)) empty l

9-6/ fold fusion

fold f_1 z_1 (fold f_2 z_2 l)

=

fold (λ (a d_{12})

f_1 fst (f_2 a (snd d_{12})) snd)

(z_1, z_2) l

map f_1 (map f_2 l) = map ($f_1 \circ f_2$) l

sum ⇒ sum ↗

Haskell

90-1/

```
int x = 7;
int y = x;
x += x + 1;
return x-y;        => 1
```

```
(let* ([x 7]
       [y x])
  ....
  (- x y))
```

In C, a variable is a container storing a value

In JS, a variable is a named value

A box is a container that might change

p = ... | box | unbox | set-box!

```
(let* ([xb (box 7)]
       [y (unbox xb)])
  (set-box! xb (+ (unbox xb) 1))
  (- (unbox xb) y))                    => 1
```

=subst=>

```
(set-box! (box 7) (+ (unbox (box 7)) 1))
(- (unbox (box 7)) (unbox (box 7)))    => 0
```

```
(let ([yb (let ([xb (box 0)])
            (set-box! xb (+ 7 (unbox xb)))
            xb)])
  (unbox yb))    => 7
```

10-2/

```
(define (make-counter)
  (let ([cb (box 0)])
    (λ ()
      (set-box! cb (+ 1 (unbox cb)))
      (unbox cb))))
(let ([c1 (make-counter)]
      [c2 (make-counter)])
  (list (c1) (c2) (c2) (c2) (c1)))
⇒ (list 1 1 2 3 2)
```

Semantics of J_6

$v = \ldots \mid \sigma$ (pointers)

old $\Rightarrow : e \to e$ $\quad E[(\text{if true } e_t\ e_c)] \Rightarrow E[e_t]$

new $\Rightarrow : \Sigma \times e \to \Sigma \times e$

$\Sigma : \sigma \to v$

$\Sigma \times E[\text{if true } e_t\ e_c] \Longrightarrow \Sigma \times E[e_t]$

$\Sigma \times E[\text{box } v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\sigma]$ where σ is fresh

$\Sigma \times E[\text{unbox } \sigma] \Rightarrow \Sigma \times E[\Sigma[\sigma]]$ $\quad \to c$

$\Sigma \times E[\text{set-box! } \sigma\ v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\overset{v}{\#}]$ $\quad \to$ not c way

10-3/ Option 1: $CEK_3 \Rightarrow CESK_0$

CESK st = <e, env, sto, k>

sto = mt | sto[σ ↦ v]

<if ec et ef, env, sto, k> ↦ <ec, env, sto, ifk env et ef k>

<v, —, sto, appk (box) () — k>

↦ <σ, mt, sto[σ ↦ v], k>

option 2: box is a primitive that does

(box v) = [V_BOX, ptr to v]

(unbox (box v)) = ret ptr

(set-box! [V_BOX, ptr1] ptr2) = changes to memory

[V_BOX, ptr2]

---

obj_t* delta_setbox (obj_t* args) {

((vbox_t*) second(args)) -> v = first(args);

return make_v_void(); }

10-4/ Option 1:

p = ... | set-fst! | set-snd!

```
(let [[p (pair 1 2)]]
  (set-fst! p 3)
  (fst p))    => 3
```

Option 2:

```
mpair x y = pair (box x) (box y)
mfst p = unbox (fst p)
mset-fst! p nx = set-box! (fst p) nx
```

```
desugar (begin) = (void)    // tt
desugar (begin e) = e
desugar (begin e x ...) = (let ([_ e]) (begin x ...))
desugar (begin0 e x ...) = (let ([y e])
                             (begin x ... y))
desugar (when c e ...) = (if c (begin e ...) void)
desugar (unless c e ...) = (when (not c) e ...)
desugar (while c e ...) =
  ((λ loop () (when c e ... (loop))))
```

10-5/

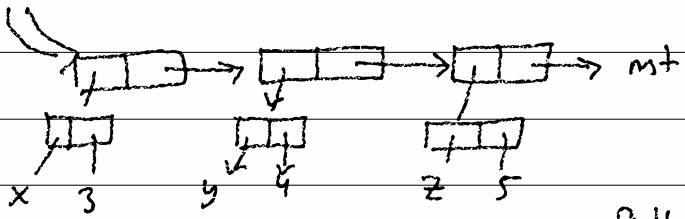
desugar (for [x := e_i ; e_c ; e_f] e_b ...)

= (let ([x e_i])
  (while e_c
    e_b ... e_f))

```
(let ([sum (box 0)])
  (for [x := (box 0) ; (unbox x) ≤ 10 ; (set-box! x (+ 1 (unbox x)))]
    (set-box! sum (+ (ub sum) (ub x)))
  (unbox sum))
```

{ x: 3, y: 4 }.y ⟹ (pair 3 4) x → fst, y → snd

{x: 3, y: 4, z: 5} ⟹ x → fst, y → fst snd, z → rst snd



mtobj = inl tt

field-set o f v =

cons (cons f v) o

11-1

```
(let ([sum (box 0)])
  (for [x (box 0)]
       (< (unbox x) 10)
       (set-box! x (+ 1 (unbox x)))
       (set-box! sum (+ (unbox x)
                        (unbox sum))))
  (unbox sum))
```

⇒

$J_0 \to J_7$: $e = \ldots \mid set!\ x\ e$

$E = \ldots \mid set!\ x\ E$

```
(let ([sum 0])
  (for [x 0] [< x 10) (set! x (+ 1 x))
    (set! sum (+ sum x))])
  sum)
```

(set! x e)

⇒

Lvalue = something with an address

(set! 1 2) "1=2" X

"f(1) = 2" X

"o.f = 2" ✓

11-2

OLD:

$$\Sigma \;/\; E[(\lambda\ (x)\ e)\ v] \Rightarrow$$
$$\Sigma \;/\; E[e[x \leftarrow v]]$$

NEU:

$$\Sigma \;/\; E[(\lambda\ (x)\ e)\ v] \Rightarrow$$
$$\Sigma[\sigma \rightarrow v] \;/\; E[e[x \leftarrow (\text{unbox}\ \sigma)]]$$

$$\emptyset \;/\; ((\lambda\ (x)\ (+\ x\ 7)\ 10) \xRightarrow{\text{OLD}} \emptyset \;/\; (+\ 10\ 7)$$
$$\xRightarrow{\text{NEU}} \emptyset[\sigma_1 \rightarrow 10] \;/\; (+\ (\text{unbox}\ \sigma_1)\ 7)$$
$$\Rightarrow \emptyset[\sigma_1 \rightarrow 10] \;/\; (+\ 10\ 7)$$

$$\Sigma \;/\; E[(\text{set!}\ (\text{unbox}\ \sigma)\ v)] \Rightarrow$$
$$\Sigma[\sigma \rightarrow v] \;/\; E[v]$$

11-3

The machine implements J_6
(set-box!, but no set!)

The desugar transforms J_7 into J_6
(set!)

desugar x = (unbox x)

desugar (set! x e) = (set-box! x (desugar e))

desugar (λ (x_0) e) = if (set! x _) $\in$ e
(λ (t_0) (let ([x_0 (box t_0)])
(desugar e))) o.w. (λ (x_0) (de e))

desugar
((λ (y) (set! y (+ y 1)) (+ y y)) 10) =
(λ (t_0) (let ([y (box t_0)])
(set-box! y (+ (unbox y) 1))
(+ (unbox y) (unbox y)))) 10)

((λ (x) (+ 1 x)) 10) $\Rightarrow$
((λ (t) (let [(x (box t))] (+ 1 (unbox x)))) 10)

desugar-top e = desugar (mutated-vars e)
e

letrec

11-4 ~~(let*~~ ([fac (λ rec (n)
(if (= n 0) 1
(* n (rec (- n 1))))]
[even? (λ rec (n)
(if (= n 0) true
(odd? (- n 1))))]
[odd? (λ rec (n)
(if (= n 0) false
(even? (- n 1))))]
...)

11-5/ (let () e) $\overset{d}{\Rightarrow}$ e

(let ([x_0 e_0] ... [x_n e_n])
e_b) $\overset{d}{\Rightarrow}$ ((λ (x_0 ... x_n) e_b)
e_0 ... e_n)

(let* () e) $\overset{d}{\Rightarrow}$ e

(let* ([x_0 e_0] [x_1 e_1] ... [x_n e_n]) e_b)
$\overset{d}{\Rightarrow}$ (let ([x_0 e_0])
(let* ([x_1 e_1] ... [x_n e_n]) e_b)

(letrec ([x_0 e_0] ... [x_n e_n]) e_b)
$\overset{d}{\Rightarrow}$ (let ([x_0 FALSE] ... [x_n FALSE])
(set! x_0 e_0) ...
(set! x_n e_n)
e_b)

11-6/ (let ([fac FALSE] [even? FALSE] [odd? FALSE])
(set! fac (λ (n) (if (= n 0) 1
(* n (fac (- n 1))))
(set! even? (λ (n) (if (= n 0) TRUE
(odd? (- n 1)))))
(set! odd? (λ (n) (if (= n 0) FALSE
(even? (- n 1)))))
(even? (fac 5)))

(letrec ([f (λ (n) (g 0))]
[x (f 5)]
[g (λ (m) m)])
x) → // fails w/ can't apply bool

Strategy 1: error-prone (Racket)
clang FALSE to UNDEFINED
Strategy 2: limitary (ML)
restrict RHS of letrec to fun defn (λ)
Strategy 3: hard to implement (Coq / Racket)
analyze the program and figure the problem

12-1) (/ 1 0) — δ is partial and undefined on δ(/, (1 0))

(5 3) — you can't call numbers as fns

(set-box! 7 0) — you can't set-box! numbers

(+ (+ 1 1) (+ 2 2))

→ (+ 2 (+ 2 2))

→ (+ 2 4) → 6

(5 3) ↛

cp = (V_NUM 3) Kp = (KAPP [(VNUM 5)] [] ∅ K)

switch (cp → tag) // V_NUMS

case V_NUM:

switch (Kp → tag) // K_APP

switch(Kp → vs [0] → tag) { // V_NUM

case V_PRIMX: .... δ(...) ... ;

case V_CLO : .... make new env, lookup code ...;

default: exit(1);

OLD: b = bool | num

12-2/ NEW: b = bool | num | ~~error~~ err_e

OLD: $E[(\lambda\ (x)\ e)\ v] \rightarrow E[e[x \leftarrow v]]$

$E[p\ \ v] \rightarrow E[\delta(p, v)]$

NEW: $E[u\ \ v] \rightarrow$ "Not a function" $\in err_e$

$u \neq p$ and $u \neq (\lambda\ (x)\ e)$ ↑ an abort

$E[X] \rightarrow E[Y]$ when you throw away the context

$J_7 \rightarrow J_8$: $e = \ldots \mid abort\ e$

E does not contain abort E

(+ 1 (+ 2 (abort (+ 3 (+ 4 0))))))

E = (+ 1 (+ 2 HOLE))

E[(abort (+ 3 (+ 4 0)))]

$E[abort\ e] \Longrightarrow e$

CEK:

$\langle abort\ e, env, k \rangle \mapsto \langle e, env, kret \rangle$

↖ throw away

12-3/ int f (int x) {

x += 8;

return x;

x *= 2;

x++;

return x+3;

}

f (5);

return is a "local" abort

⟷ k=

(λ ()

x *= 2;

x++

return x+3)

(define (fac n)

(if (< n 0) (abort "only non-neg ~~integers~~")

(if (= n 0) 1

(* n (fac (- n 1))))))

12-4/ (+ 1
(try
(+ 2
(throw 3)) ⇏ 10
catch ⇏ 6
(λ (x) (+ x 4)))) ⇒ 8

$J_8 \Rightarrow J_9$: e = ... | throw e |
try e catch e
E = ... | try e catch E
| try E catch v
L = E except no try case

E[try v_1 catch v_2] → E[v_1]
E[try L[throw e_1] catch v_2] → E[v_2 e_1]
L[throw e_1] → L[abort e_1] → e_1

E = (+ 1 hole) L = (+ 2 hole)
e_1 = 3 v_2 = (λ (x) (+ x 4)
→ (+1 ((λ x. (+ x 4)) 3))
→ (+1 (+ 3 4)) → (+1 7) → 8

12-5/ $CEK_4 \Rightarrow CEK_5$

$K = \ldots \mid kif\ env\ e\ e\ k \mid kapp\ (v \ldots)\ (e \ldots)\ env\ k \mid kTryPre\ env\ e\ k \mid kret \mid kTryPost\ v\ k$

$\langle try\ e_1\ catch\ e_2, env, k \rangle \mapsto \langle e_2, env, kTryPre\ env\ e_1\ k \rangle$

$\langle v, -, kTryPre\ env\ e\ k \rangle \mapsto \langle e, env, kTryPost\ v\ k \rangle$

$\langle v_1, -, kTryPost\ v_2\ k \rangle \mapsto \langle v_1, -, k \rangle$

$\langle throw\ e_1, env, kTryPost\ v_1\ k \rangle \mapsto \langle e_1, env, kAPP\ (v_1)\ ()\ -\ k \rangle$

$\langle throw\ e_1, env, kif\ env'\ e_t\ e_f\ k \rangle \mapsto \langle throw\ e_1, env, k \rangle$

$\langle throw\ e_1, env, kapp\ (v \ldots)\ (e \ldots)\ env'\ k \rangle \mapsto \langle throw\ e_1, env, k \rangle$

$\langle throw\ e_1, env, kTryPre\ env'\ e\ k \rangle \mapsto \langle throw\ e_1, env, k \rangle$

$\langle throw\ e_1, env, kret \rangle \mapsto \langle e_1, env, kret \rangle$

12-6

OLD: (5 3) ⇏

MID: (5 3) → error ("Not a fun")

NEW: (5 3) → throw error "Not a fun"

((λ (f) (f 3)) 5)

((λ (f)
(try (f 3) catch (λ (x) 15)))
5) ⇒ 15

E [u v] → E [throw "Not a fun"]
if u ∉ P and u ∈ (λ (x) e)

(abort) J_9 = try, throw

13-1 | $J_8 \Rightarrow J_{10}$ EP

$e = \ldots \mid$ call/cc e

$$E[\text{call/cc } v] \Rightarrow E[v\ (\lambda (x)\ (\text{abort } E[x]))]$$

```
(+ 1
   (call/cc (λ (esc)
              (let ([throw (λ (y) (esc (+ y 4)))])
                (+ 2 (throw 3))))))
```

⇒ E = (+ 1 ■) // E[(call/cc ↗)]

(+ 1 ((λ (esc) ...) (λ (x) (abort (+ 1 x)))))

⇒ (+ 1 (+ 2 ((λ (y) ((λ (x) (abort (+ 1 x))) (+ y 4))) 3)))

⇒ (+ 1 (+ 2 ((λ (x) (abort (+ 1 x))) 7)))

⇒ (+ 1 (+ 2 (abort (+ 1 7))))

⇒ (+ 1 7) ⇒ 8 E[abort e] ⟹ e

132/ desugar (try e_1 catch e_2) =>

try-catch (λ () e_1) e_2

desugar (let/cc x e) => callcc (λ (x) e)

standard library:

```
throw := (λ (x) ((unbox last-handler) x))
last-handler := (box (λ (x) (abort x)))
try-catch := (λ (body new-handler)
  (let ([old-handler (unbox last-handler)])
    (begin0 (let/cc here (set-box! last-handler
                           (λ (x) (here (new-handler x))))
              (body))
      (set-box! last-handler old-handler))))
```

13-3/ return statements:

```
(define (fac x)
  (when (< x 0) (return false))
  (if (= x 0) 1
      (* x (fac (- x 1)))))
```

=>

desugar (define (f x ...) b)
=> (define (f x ...) (let/cc return b))

break/continue:

OLD
desugar (while c b) => ((λ loop () (when c b (loop))))

NEW => ((λ loop ()
(when c
(let/cc break (let/cc continue b)
(loop)))))

[3-4] R^{5}RS (definition of Scheme) 1998
(mechalder)

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

$CEK_4 \xrightarrow{(abort)} CEK_6$ (call/cc)

$v = \ldots \mid kcont\ k$

$\langle v, -, kApp\ [call/cc]\ []\ \_\ k \rangle$
$\mapsto \langle kcont\ k, -, kApp\ [v]\ []\ \_\ k \rangle$
$\langle v, -, kApp\ [kcont\ k]\ []\ \_\ \_ \rangle$
$\mapsto \langle v, \emptyset, k \rangle$

13-5/ compiler from J_{10} (call/cc) $\rightarrow$ J_8 (abort?) / J_7 (fail)

CPS - continuation-passing style

input:

(+ 1 (call/cc (λ (esc) (+ 2 (esc 3)))))

old: + : V × V → V call/cc : ~~(K → V) → V~~ (K → V) → V

new: + : V × V × (V → A) → A

~~call/cc~~ call/cc : ((V → A), (V → A)) → A) × (V → A) → A

call/cc := (λ (f k)
(f (λ (v nk) (k v)) k))

(call/cc (λ (esc k)
(esc 3 (λ (ans) (+ 2 ans k)))
(λ (ans) (+ 1 ans top))))

⇒ (esc 3 (λ (ans) (+ 2 ans k)))
[esc ~~=~~ (λ (v nk) (+ 1 v top)),
k ↦ (λ (ans) (+ 1 ans top))]

⇒ (+ 1 3 top) ⇒ (top 4) ⇒ 4

13-6 cps-m :

$a = b \mid x$

$call = (a \ \ldots)$

$v = b \mid (\lambda\ (x \ldots)\ call)$

$e = (\lambda\ (top)\ call)$

$\langle (\lambda\ (x \ldots)\ call), env \rangle$

$\mapsto \langle clo($

It's simple!

$a = v \mid x$

$call = (a \ \ldots)$

$v = b \mid (\lambda\ (x \ldots)\ call)$

; $st = \langle call, env \rangle$

$\langle (a_0\ a_1 \ldots a_n), env \rangle$

$\mapsto \langle c, env' \rangle$

where $\alpha(a_0) = (\lambda\ (x_1 \ldots x_n)\ c)$

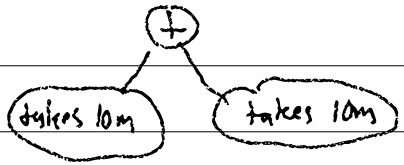
$v_1 \ldots v_n = map\ \alpha\ (a_1 \ldots a_n)$

$env' = \emptyset[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$

$\alpha\ b = b$ $\alpha\ x = env[x]$

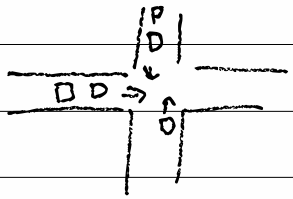
15-1) Concurrency — logical simultaneity

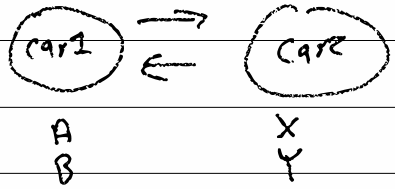
Parallelism — concrete simultaneity



Serial / Sequential — 20m

Parallel — 10m





possible programs:

A B X Y — P → Z

X Y A B — Z → I

A X B Y — interleave 1st 2nd

X A Y B — 2st

X A B Y — car2 / car1

A X Y B — car1 / car2

A Y X B X

Concurrent has N ans

Parallel has 1 ans

15-2/ spawn! : (→ α) → void

exit! : () → void

< = happens before

```
(begin (spawn! (λ () (print 'A)        A < B
                     (print 'B)))      X < Y
       (spawn! (λ () (print 'X)
                     (print 'Y)))
       (exit!))

(define RQ empty)    ;; RQ = ready queue
(define (spawn! t)
  (set! RQ (cons (λ () (begin (t) (exit!))) RQ)))
(define (exit!)
  (if (empty? RQ) (void)
      (let ([next (first RQ)])
        (set! RQ (rest RQ))
        (next))))
```

15-3/ (let ([m 10])
  (spawn! (λ () (set! m (* 2 m))))
  (spawn! (λ () (set! m (+ 2 m))))
  (exit!))

$10 \xrightarrow{\times} 20 \xrightarrow{+} 22$ $10 \xrightarrow{+} 12 \xrightarrow{\times} 24$

make-channel : () → Chan
send! : Chan × A → ()
recv! : Chan → A

(let ([ch (make-channel)])
  (spawn! (λ () (send! ch 20)))   ← A_1
  (spawn! (λ () (print (+ 2 (recv! ch)))))   ← B
  (exit!))

A_2: (send! ch 8)

AB = 22   BA = dead   "match pair"

(spawn! (λ () (print (- (recv! ch) 2))))

ABC = 22   BAC = 18   CAB = 22
ACB = 18   BCA = 0   CBA = 0

22, 8 — B, C
18, 20 — C, B

15-4/

make-chan : () => Chan A

send! : Chan A x A -> ()

recv! : Chan A => A

Empty channel ↙ ↘ SendCh

Chan A = Box (1 + (A x (() => B) x Chan B)
+ ((A => C) x Chan A))

RECV ch

```
make-chan := (box ++)
send! ch v :=
    case (unbox ch) of
        RecvCh f ch' -> (spawn! (λ () (f v)))
                        (set-box! ch ch')
        next -> (let/cc me
                    (set-box! ch (SendCh v me next))
                    (exit!))
recv! ch :=
    case (unbox ch) of
        SendCh v f ch' -> (spawn! f)
                          (set-box! ch ch'); v
        next -> (let/cc me
                    (set-box! ch (RecvCh me next))
                    (exit!))
```

15-5/ make-promise : (() -> A) -> Promise A

fulfill : Promise A -> A

```
make-promise f := (let [ch (make-chan)]
                    (spawn! (λ() (send! ch (f))))
                    ch)
fulfill pch := recv! pch
```

16-1/ errors like (+ 1 #true)
resulted in (abort "can't add bools")

#1 → 2 (C logic) JS
#2 → crash i.e. abort (~~Python~~ logic)
#3 → exception (Python logic)

#1: unsound #2: UNSAFE
#3: SAFE

p := .... | number? | box? | bool? | prim? | pair? | ...
| function-arity

((λ (x) (+ 1 x)) 1 2)

+

(define (safe+ x y)
(if (and (number? x) (number? y))
(+ x y) unsafe+ = #%+
(+ throw "Not a number")))

16-2/

```
(define (f x) (* x 2))
(f 1 2)
```

desugar $(e_0 \ldots e_n) = (\text{SAFE-CALL } e_0 \ (\text{list } e_1 \ldots e_n))$

```
(define (f x) (SAFE-APPLY * (list x 2)))
(SAFE-APPLY f (list 1 2))
```

```
(define (SAFE-APPLY f args)
  (if (function? f)
      (if (= (length args) (function-arity f))
          (#%apply f args)
          (throw "wrong num"))
      (throw "not fun")))
```

v_e v_0

$\langle$ [illegible], $-$, KAPP([#%apply [illegible]] [] $-$ k) $\rangle$

$\longmapsto \langle$ [illegible] vN_1, $-$, KApp $[v_0\ v_1 \ldots v_{n-1}]$ [] $-$ k $\rangle$

where $v_e = (\text{list } v_1 \ldots v_n)$

(apply + (list 2 3)) $\Rightarrow$ (+ 2 3) $\Rightarrow$ 5

16-3/ bst-insert : BST? × Num? -> BST

```
(define bst-insert b v)
  (unless (BST? b) (throw ...)
  (unless (num? v) (throw ...)
  ... )
```

MOD:

unsafe-f_1 : .... unsafe-f_2 ....

f_2

f_3

f_1 : Check (unsafe-f_1 ...)

f_2 : check (unsafe-f_2 ...)

protect

apply-contract : Value × Contract × Blame × Blame

bst-insert

```
(apply-contract unsafe-bst-insert
                (fun-cte (list BST? num?)
                         BST?)
                "me" "them"))
```

16-4/ desugar (with-module
(module
body ...
#:exports
[f ctc] ...) user ...)
=> (apply (λ (f ...) user ...)
(letrec (body ....)
(list (contract f ctc ME THEM)
...)))

Contract = FlatCtc (A -> Bool)
FunCtc (List contract) contract
BST? = FlatCtc (λ (x) ...)

protect v (FlatCtc pred) pos neg =
(if (pred v) v (error "pos" "is not flat"))
protect v (FunCtc dom rng) pos neg =
(if (function? v) (= (length dom) (fun-arity f))
(error pos "not a fun or wrong arity")
(λ args
(if (! (= (len args) (length dom))) (error neg))
(protect (apply v (map (λ (a ctc) (protect a ctc neg pos))
args doms)
rng pos neg)))

16-5/

(predef. map

⊕ (Functc ⊖ (list (Functc ⊕ (list Num?)
⊖ Bool?)
(Listctc Num?)
⊕ (Listctc Bool?)))

17-1 Syntactic Extension

(time e) ⇒ return value of e
and print how long it took
to run

now ∈ p

E[(λx.b) v] ⇒
E[b[x←v]]

(define (time' x)
(let* ([before (now)]
[ans (x)]
[after (now)])
(displayln (- after before))
ans))

~~(time (fib 100))~~
(time' (λ () (fib 100)))

cpp
#define TIME(e) (time' (λ () e))

C macros operate on text of programs
NOT values of the program

17-2/ cpp problems

```
#define SUB -␣2
...
return x SUB;   ⇒   x - 2
...

#define FOO 2);
...
    printf("The number is %d", FOO
    return 1;
```

"(+ 2 2)" ⟺ 

text-editor — program tools (compiler/lm)

17-3/ macros : syntax → syntax (like desugar)

macros-by-example (1986) - we'll implement

```
(define-syntax-rules let          ← macro name
  [ (let ([x xe] ...) be ...)     ← pattern
    ((λ (x ...) be ...) xe ...)]) ← template
```

(dsr name [pat template] ...)

expand : Set of Rules × Sexpr → Sexpr

(let ([x 3] [y 4]) (+ x y))

```
expand Σ (cons mac rest) =
  if Σ(mac) = defn then
      expand1 defn (cons mac rest)
  else
      App ( expand Σ mac , map (expand Σ) rest)
expand Σ (Num n) = (Num n)
expand Σ (cons 'lambda (cons args body)) =
    Lambda args (expand Σ body)
```

17-4 / expand I

pato

defn = (list (pair (let ([x xe] ...) be ...) tempo
((λ (x ...) be ...) xe ...)))

use = (let ([a 3] [b 4]) (+ a b))

match pato use = match : Pat x Sexpr ⇒ Env or #f

[x ↦ (a b)
xe ↦ (3 4)
be ↦ ((+ a b))]

transcribe tempo ∫ =

((λ (a b) (+ a b)) 3 4)

expand d Σ (define-syntax-rules mac defns)

= Σ[mac] = defns

match : Pat × Sexpr → Env or false

match () () = ∅

match b b = ∅

match x se = [x ↦ (se, se)]

match (p '...) se =

  merge (map (match p) se)

match (cons pl pr) (cons l r) =

  match pl l ∪ match pr r

merge : List (Env or false) → Env

merge [] = ∅

merge [... false ...] = false

merge [σ₀ σ₁ ...] =

  new hash where k ... = keys of σ₀

    k ↦ (level+1, vs)

  where σ₀(k) = (level, —)

    vs = (list snd(σ₀(k)) snd(σ₁(k)) ...)

17-6)

```
match ([x xe] ...)
      ([a 3] [b 4]) =
merge (map (match [x xe])
           (list [a 3] [b 4])) =
merge (list (match [x xe] [a 3])
            (match [x xe] [b 4])) =
merge (list [x ↦ (0, a), xe ↦ (0, 3)]
            [x ↦ (0, b), xe ↦ (0, 4)]) =
[ x ↦ (1, (a * b))
  xe ↦ (1, (3 4)) ]
```

177/ transcribe : env x template ⇒ sexpr

transcribe σ () = ()

transcribe σ b = b

transcribee σ x = se where σ(x) = (0, se)

tr σ (p ...) =

map (λ (σ') (tr σ' p))

(decompose σ)

tr σ (cons ta td) = (cons (tr σ ta)

(tr σ td))

decompose : env → list env

decompose σ = (list σ0 ... σn)

where n = length of-val σ [x]) for some x

σi[x] = (level - 1, vs [i])

where (level, vs) = σ [x]

17-8/ (define-syntax-rules or
[(or) true]
[(or x y ...)
(let ([tmp x])
(if tmp tmp (or y ...)))])

(or false 3 4) => 3
(begin (display "H:!")
3)

(let ([tmp$_{red}$ 7])
(or false tmp$_{red}$)) => 7
(let ([tmpred 7])
(let ([tmpbl false])
(if tmpbl tmpbl
(let ([tmp$_{gr}$ tmp$_{red}$])
(if tmp$_{gr}$ tmp$_{gr}$ false))))) =>

macro : stx → stx

18-1/ Backtracking Non-determinism

```
nde = (ans v)                    : Ans
    | fail                       : ⊥
    | (choice nde nde)           : (ND A) × ND A → ND A
    | (bind nde f)               : ND A → (A → ND B) →
                                   ND B

run : ND a → ~~list~~ a stream a
```

```
empty-stream : empty
stream-cons : a × (→ stream a) → stream a
stream-first : stream a → a
stream-rest : stream a → stream a
```

```
run p = sols (list (st p (kont:return)))
sols '() = empty-stream
sols (cons (st p k) q) =
  case p of (bind p' f) → sols (cons (st p' (kont:bind f k)) q)
            (choice p1 p2) → sols (cons (st p1 k) (st p2 k) q)
            (fail) → sols q
            ans v → case k of return → stream-cons v (() (sols q))
                              bind f k → sols (cons (st (f v) k) q)
```

18-2/ query : Question → ND ans

query q = bind (searchN DB ∅ (list q))
(λ (env) (ans (transcribe env q)))

searchN : Rules × Env × List(Questions) → ND Env

searchN rules env [] = ans env

q : qs = bind (searchN rules env qs)
(λ (env') (search* rules env' rules q))

search* : Rules × Env × Rules × Question → ND Env

search* all env same q = case some of

[] → fail | ruleN : rules →

choice (search* all env rules q)
(search1 all env ruleN q)

search1 : Rules × Env × Rule × Question → ND Env

search1 all env (conclusion, deps) q =

(bind (match conclusion q)
(λ (env') (searchN all env/env' deps)))

18-3/ Non-determinism Expressions are a Monad

```
Monad A =
    return : A -> Monad A
    bind   : Monad A -> (A -> Monad B)
                     -> Monad B

ListMonad
    return x = [ x ]
    bind mx f = map f mx

bind (return 1) add1 => [2]
bind (list 1 2 3) add1 => [2 3 4]
bind x (λ (a)
    bind (list (addI a) (sub1 a))
        mult2 )) =>
```

19-1 Memory Management

$e = b \mid (e\ e) \mid x$

$v = b \mid (\lambda x.\ e)$

$k = ret \mid kfun(e, env, k) \mid karg(v, k)$

$\langle x, env, k \rangle \mapsto \langle env[x], \emptyset, k \rangle$

$\langle \lambda x.e, env, k \rangle \mapsto \langle clo(\lambda x.e, env), \emptyset, k \rangle$

$\langle e_1\ e_2, env, k \rangle \mapsto \langle e_1, env, kfun(e_2, env, k) \rangle$

$\langle v, \emptyset, kfun(e_2, env, k) \rangle \mapsto \langle e_2, env, karg(v, k) \rangle$

$\langle v, \emptyset, karg(clo(\lambda x.e, env), k) \rangle \mapsto \langle e, env[x \mapsto v], k \rangle$

If one object has two pointers to it, we say it is "aliased"

MM = how to manage scarce memory

- how to allocate — malloc impl
- when to free — you call free

right ~~wrong~~ oom ~~crash~~

soundness =

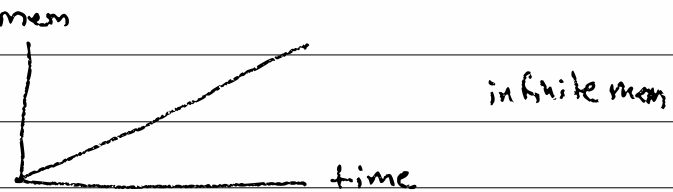
efficiency { space, time

performance

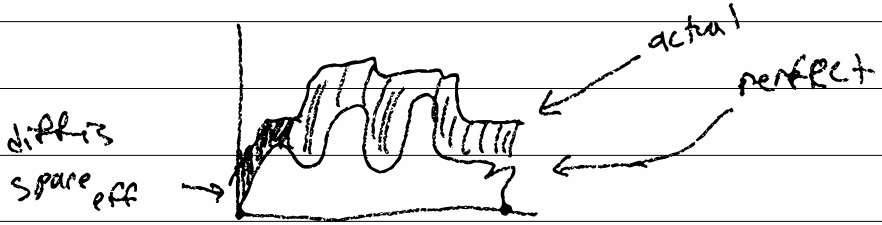
19-2 soundness — right wrong OOM crash

efficiency

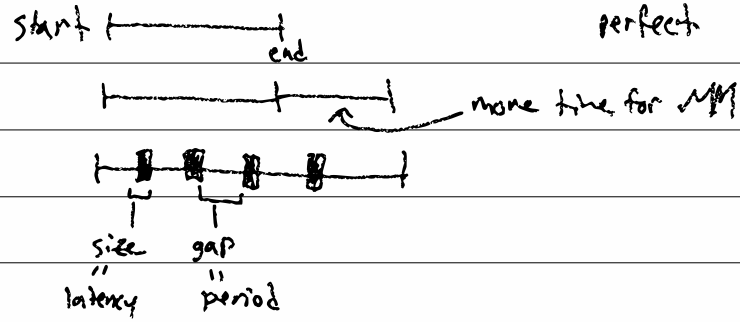
— space

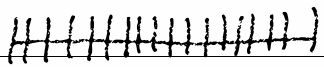


peak usage = highest point

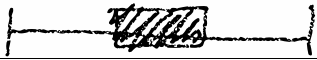


— time



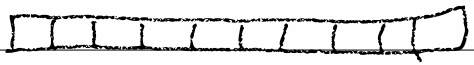


low latency



high latency

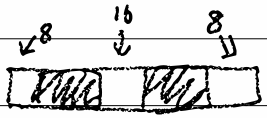
14-3/ C-style malloc & free

mem = 

malloc(sizeof(node)) = give me 16 spots

free(ptr) = figure out adj free space

O(lg n)



= fragmentation

~~fine eff~~ = malloc=free = O(lg n)

~~space eff~~ = O(lg n)

soundness = (not)

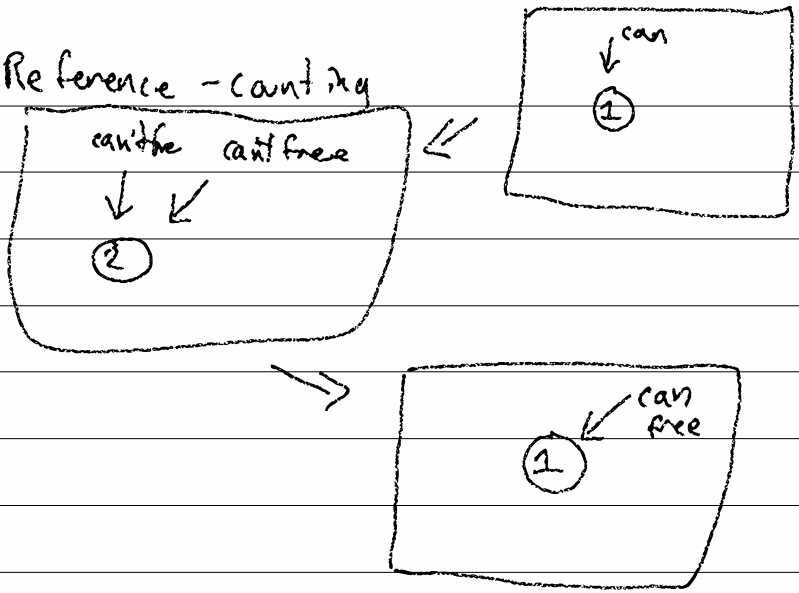
performance = •

} depend on programs

sadly:

turn all aliases into copies (halt copying, X2 proc)

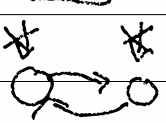
14-4/ Reference-counting



malloc: ref=1 ref: ref++ unref: ref--
if (ref==0) {free}

sound: who calls ref/unref? if programmer ⇒ unsound
if lang ⇒ sound

space eff: every object stores count
LL(16) ⇒ |16 + CNT| CNT = [1, 8]



cycles are never deleted

time eff: ∝ work of program O(w) not O(n)
mem

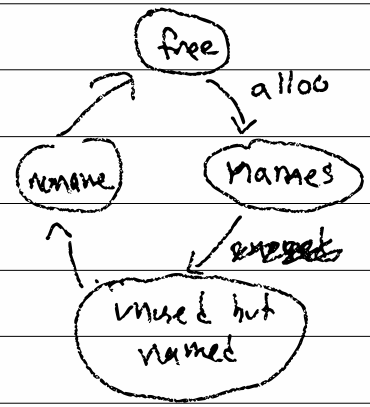
latency: cascading deletes of large frees = high latency
manage w/ queues

20-1/ Garbage Collection (a sound memory manager)

John McCarthy for LISP in 1959

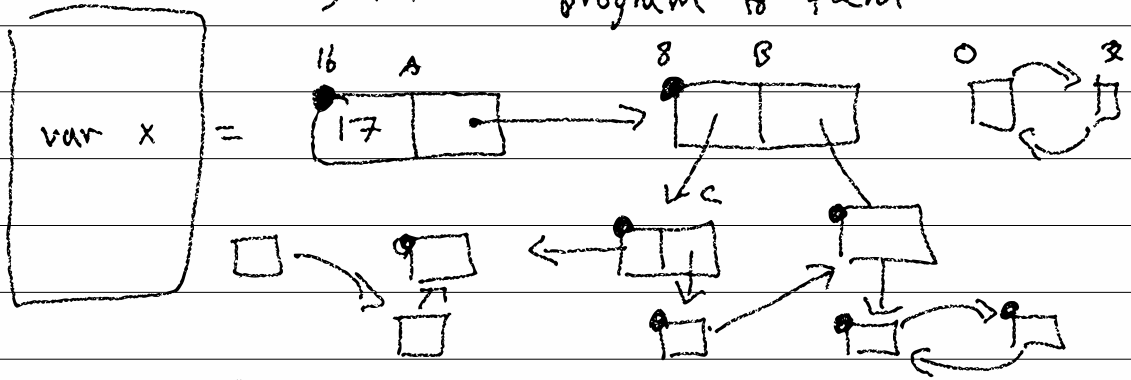
Mark & Sweep

```
....
var x = ....;
line 30: .... x .... x .. x ...
    if (cc) {
line 42: ... x ...;
    [ never see x again
    return;
```



the working set = all objects with paths from program to them

root-set = program



c = x[1][0]

garbage = mem - working

20-2/ mark = from root set, mark all children recursively

sweep = go through mem, delete if no mark

remove mark o.w.

CESK+MS : $st = \langle \sigma, \sigma, \sigma, \Sigma \rangle$ $\sigma = \emptyset \mid n$

$\Sigma = \Sigma[\sigma \mapsto *]$ $k = kret$

$e = v \mid @\,\sigma\,\sigma \mid x \mid kfun(\sigma, \sigma_k)$

$v = b \mid \boxed{\lambda x.\, \sigma} \mid clo(\sigma, \sigma) \mid karg(\sigma, \sigma)$

$env = \sigma[x \mapsto \sigma]$

$\langle \sigma_e, \sigma_{env}, \sigma_k, \Sigma \rangle$ = case $\Sigma(\sigma_e)$ of

$x \Rightarrow \langle \Sigma(\sigma_{env})[x], \emptyset, \sigma_k, \Sigma \rangle$

$\lambda x.\sigma_b \Rightarrow \langle \sigma_{clo}, \emptyset, \sigma_k, \Sigma' \rangle$ $\Sigma' = \Sigma[\sigma_{clo} \mapsto clo(\sigma_e, \sigma_{env})]$

$@\,\sigma_L\,\sigma_R \Rightarrow \langle \sigma_L, \sigma_{env}, \sigma_{k'}, \Sigma' \rangle$ $\Sigma' = \Sigma[\sigma_{k'} \mapsto kfun(\sigma_R, \sigma_{env}, \sigma_k)]$

$\_ \Rightarrow$ case $\Sigma(\sigma_k)$ of

$kfun(\sigma_R, \sigma_{env}, \sigma_{k'}) \Rightarrow \langle \sigma_R, \sigma_{env}, \sigma_k, \Sigma' \rangle$ $\Sigma' = \Sigma[\sigma_k \mapsto \overset{karg}{\sigma_e}(\sigma_{k'})]$

$karg(\sigma_{clo}, \sigma_{k'}) \Rightarrow \langle \sigma_{body}, \sigma_{env}'', \sigma_{k'}, \Sigma' \rangle$

where $\Sigma(\sigma_{clo}) = clo(\sigma_e, \sigma_{env}')$ $\Sigma(\sigma_e) = \lambda x.\, \sigma_{body}$

$\Sigma'[\sigma_{env}'' \mapsto \sigma_{env}'[x \mapsto \sigma_@]]$

20-3 $\langle \sigma_e, \sigma_{env}, \sigma_\kappa, \Sigma \rangle \overset{MS}{\mapsto} (\langle \sigma_{e'}, \sigma_{env'}, \sigma_{\kappa'}, \Sigma' \rangle)$

$MS(\langle \sigma_e, \sigma_{env}, \sigma_\kappa, \Sigma \rangle) \overset{st}{=} \langle \sigma_e, \sigma_{env}, \sigma_\kappa, \Sigma' \rangle$

$w = \text{working}(st) = \text{mark}$

$\Sigma' = \text{removeAllBut}(\Sigma, w) = \text{sweep}$

$\text{sweep}(\emptyset, w) = \emptyset$ $\quad \text{sweep}(\Sigma[\sigma \mapsto v], w) = \Sigma''$

where $\Sigma' = \text{sweep}(\Sigma, w)$

$\Sigma'' = \text{if } \sigma \in w \text{ } \Sigma'[\sigma \to v]$ o.w. Σ'

$\text{mark}(\langle \sigma_e, \sigma_{env}, \sigma_\kappa, \Sigma \rangle) =$

$\text{mark}(\sigma_e, \Sigma) \cup \text{mark}(\sigma_{env}, \Sigma) \cup$

$\text{mark}(\sigma_\kappa, \Sigma)$

$\text{mark}(\sigma, \Sigma) = \text{case } \Sigma(\sigma) \text{ of}$

$b \mapsto \emptyset \mid x \mapsto \emptyset \mid @\,\sigma_L\ \sigma_R \Rightarrow \overset{\text{mark}(\sigma_L)}{\cup\, \text{mark}(\sigma_R)}$

$\lambda x.\sigma_b \Rightarrow \text{mark}(\sigma_b, \Sigma) \mid clo(\sigma_e, \sigma_{env}) \mapsto M(\sigma_e, \Sigma) \cup M(\sigma_{env}, \Sigma)$

$\sigma_{env}[x \mapsto \sigma_v] \Rightarrow M(\sigma_{env}) \cup M(\sigma_v) \mid kret \mapsto \emptyset$

$kfun(\sigma_R, \sigma_{env}, \sigma_k) \mapsto M(\sigma_R) \cup M(\sigma_{env}) \cup M(\sigma_k) \mid$

$karg(\sigma_{clo}, \sigma_k) \mapsto M(\sigma_{clo}) \cup M(\sigma_k)$

$n = m\text{-}em$

20-4/ sound = YES

space = fragmentation "optimal"

mark bits $O(\lg n)$

time = malloc = $O(\lg n)$

mark = $O(\text{work})$

sweep = $O(n)$ (not great)

latency = default bad

but possible to be better (real-time GCs)

21-1/ M&S — time: O(mem) (looks at working+garbage)

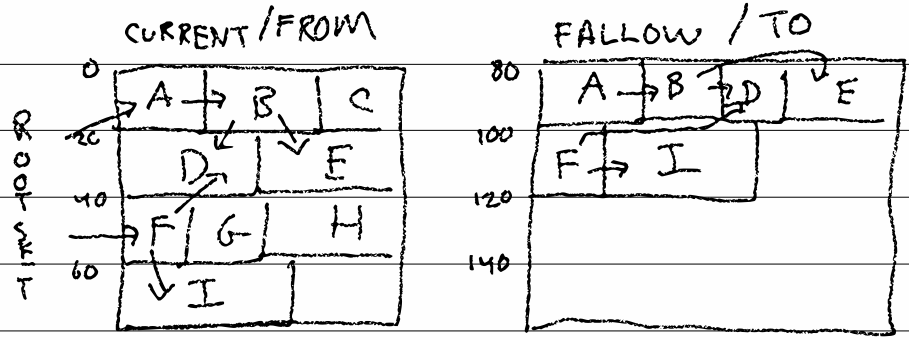
malloc: O(lg n)

space: optimal

Stop + Copy: time: O(working)

malloc: O(1)

space: x2



time 0: rs[x] = A (@ 0) → A (@ 80)

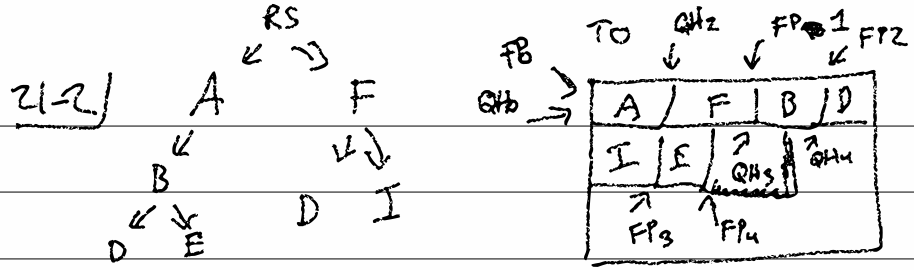
time 1: A.data = B (@ 5) → B (@ 85)

time 3: B.left = D(@ 20) → D (@ 95)

20/ D = [OBJ, 42, 50]

95/D = [OBJ, 42, 50]

20/D = [MOVED, 95]



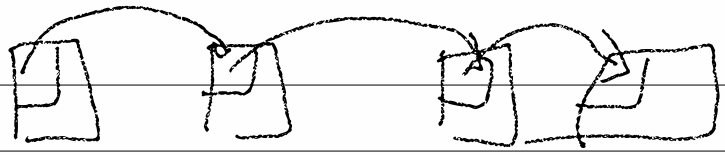
Cheney

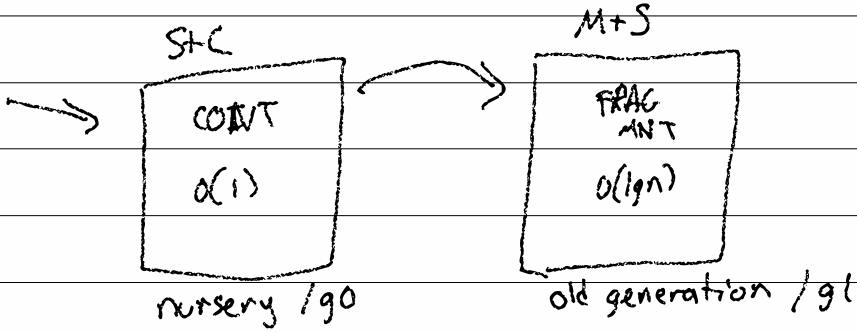
enqueue (o) = store it at FP, update the ref, FWD ptr

dequeue () = read the QH,
move it 1 obj
enqueue children

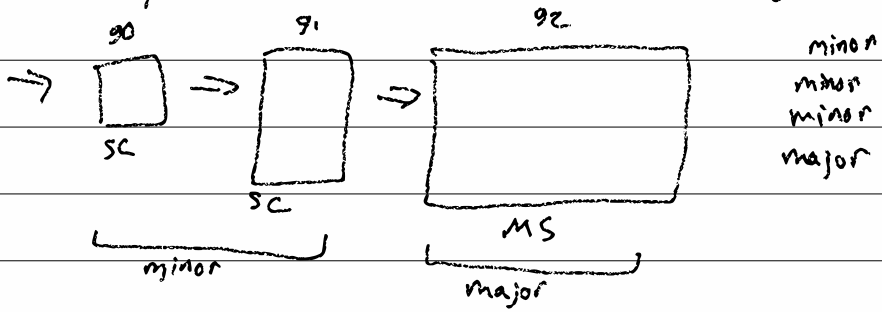
1. enqueue RS
2. while (QH ≠ FP)
   dequeue

22-1





Generational Hypothesis: Most objects die young.



Inter-generational Pointers: pointers from g_0 to g_1

or g_1 to g_0

NEW to OLD ← not a problem because major ⇒ minor

OLD to NEW — minor ⇒ ignore g_1

cache the references from last major

monitor all ptr creation and add OLD⇒NEW to cache

22-2/ Radioactive Decay Model

Objects have a garbage half-life.

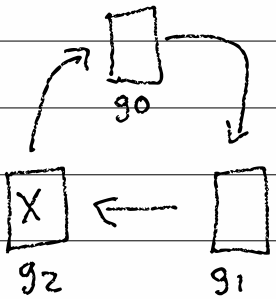
→

The longer you wait, the more garbage you'll find

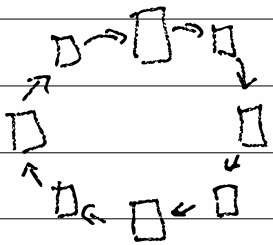
Clinger – Clock Collector

S+C – 2-Clock

3 clock:



8-clock



23-1) errors are undefined behavior

$(+\ 5\ \#true) \not\Rightarrow$

$(if\ 5\ 3\ 4) \not\Rightarrow$

$(7\ 3) \not\Rightarrow$

$E[(\lambda x.\ e)\ v] \rightarrow E[e[x \leftarrow v]]$

$E[if\ true\ e_t\ e_f] \rightarrow E[e_t]$

$E[p\ v \ldots] \Rightarrow E[u]$ where $u = \delta(p, v \ldots)$

$\delta(+, 5\ \#one) = \bot$

```
(define (+ x y) (#%+ x y)))
                 ↗
          real primitive
(if (and (num? x) (num? y))
    ...
    (error ...))
```

Consequences — more & performance

here:	+ :	any	any →	num or error
before:	+ :	any	any →	num or ⊥
want:	+ :	num	num →	num

232/ A static type system is an analysis of a program that correctly predicts (the values the program evaluates to. / the behavior of the program.)

eval : Program ⇒ Ans or ⊥

type : Program ⇒ Prediction

abstract : Ans ⇒ Prediction

concrete : Prediction ⇒ P(Ans)

abstract 5 = Number concret bool = {true, false}

abstract true = bool concrete enum = {0, 1, -1, ...}

$\forall x.$ concrete(abstract(x)) $\ni x$

$\forall x.\ \forall y \in$ concrete(x) ⇒ abstract(y) = x

type(pr) = p iff abs(eval(pr)) = p

↗ Soundness theorem

progress: If type(e) = T then $e \to e'$

preservation: If type(e) = T and $e \to e'$ then

type(e') = T

23-3/ $e = num \mid (+\ e\ e) \mid (*\ e\ e)$

$T = Num$

Inference rules ① $\forall e.\ e = num \Rightarrow type(e) = Num$

② $\forall e_1, e_2.\ type(e_1) = Num \wedge type(e_2) = Num$
$\Rightarrow type(+\ e_1\ e_2) = Num$

③ $\forall e_1, e_2.\ ty(e_1) = Num \wedge ty(e_2) = Num$
$\Rightarrow type(*\ e_1\ e_2) = Num$

$(+\ 1\ (*\ 2\ 3))$ $= Num$ apply rule #2

$ty(1) = Num$ apply #1 $\wedge$ $ty(*\ 2\ 3) = Num$ apply #3

$ty(2) = N$ apply #1 $\wedge$ $ty(3) = Num$ apply #1

$e : T$ means $type(e) = T$

implicitly $\forall$ vars

$\frac{a_1\ a_2 \cdots a_n}{b_1}$ $:= a_1 \wedge a_2 \wedge \ldots a_n \Rightarrow b_1$

n $\frac{}{num : Num}$

a $\frac{e_1 : Num \quad e_2 : Num}{(+\ e_1\ e_2) : Num}$

m $\frac{e_1 : Num \quad e_2 : Num}{(*\ e_1\ e_2) : Num}$

23-4/

$$\frac{\dfrac{1 : \text{Num}}{(+\ 1\ (*\ 2\ 3)) : \text{Num}}\ n \qquad \dfrac{\dfrac{2 : \text{Num}}{(*\ 2\ 3) : \text{Num}}\ n \qquad \dfrac{3 : \text{Num}}{}\ n}{}\ m}{}\ \text{Add}$$

$e = v \mid (p\ e\ \ldots) \mid (\text{if}\ e\ e\ e)$

$v = b$

$b = \text{num} \mid \text{bool}$

$T = \text{Num} \mid \text{Bool}$

$p = \ldots \mid + \mid - \mid * \mid / \mid < \mid = \mid \ldots$

$\Delta(p) = (T_1 \ldots T_n) \Rightarrow T$

$$\frac{}{\text{num} : \text{Num}}\ n \qquad \frac{}{\text{bool} : \text{Bool}}\ b$$

$$\frac{e_1 : T_1 \quad \ldots \quad e_n : T_n}{(p\ e_1 \ldots e_n) : T}\ p$$

$$\frac{e_c : \text{Bool} \quad e_t : T \quad e_f : T}{\text{if}\ e_c\ e_t\ e_f : T}\ i$$

$\Delta(+) = (\text{Num Num}) \Rightarrow \text{Num}$

$\Delta(=) = (\text{Num Num}) \Rightarrow \text{Bool}$

$\Delta(\text{not}) = (\text{Bool}) \Rightarrow \text{Bool}$

(if (= 5 5) (* -1 (+ 5 0)) (- 5 10))

$$\frac{\dfrac{}{5 : \text{Num}}\ n \qquad \dfrac{}{\text{true} : \text{Num}}}{(+\ 5\ \text{true})}\ p$$

$\Delta(+) = (\text{Num Num}) \Rightarrow \text{Num}$

$\times$

$$\frac{\dfrac{}{5 : \text{Bool}} \qquad \dfrac{}{0 : \text{Num}}\ n \qquad \dfrac{}{1 : \text{Num}}\ n}{(\text{if}\ 5\ 0\ 1)}\ i$$

$\times$

23-5/ $\dfrac{(+\ \ 1\ \ (\text{if}\ (=\ 5\ 5)\ \ 6\ \ \text{false}))}{}$ P

$\Delta(+) = (N\ N) \Rightarrow N$

$\dfrac{1 : N \qquad (\text{if}\ (=\ 5\ 5)\ 6\ \#f) : N}{}$;

$\dfrac{(=\ 5\ 5) : \text{Bool} \qquad 6 : N \qquad \#f : N}{}$ P

$\Delta(=) = (N\ N) \Rightarrow B$ ✗

$5 : N \qquad 5 : N$

(A v B) => P iff A => P ∧ B => P

---

e = v | x | (p e ...) | (if e e e)
| let x := e ; in e

v = b p = ... OLD e : T

T = Num | Bool NEW $\Gamma \vdash e : T$

$\Gamma : x \Rightarrow T$

$\dfrac{T = \Gamma(x)}{\Gamma \vdash x : T}$ $\qquad$ $\dfrac{\Gamma \vdash e_c : B \quad \Gamma \vdash e_t : T \quad \Gamma \vdash e_e : T}{\Gamma \vdash (\text{if}\ e_c\ e_t\ e_e) : T}$

let c := (= 5 5) ; in
(if c 6 false)

$\dfrac{\Gamma \vdash e_x : T_x \quad \Gamma[x \mapsto T_x] \vdash e_b : T}{\Gamma \vdash \text{let}\ x := e_x\ \text{in}\ e_b : T}$

$$e = \ldots \mid (e\ e)$$

$$v = \ldots \mid \lambda x.e \mid \lambda x{:}T.\ e$$

$$T = \ldots \mid T \Rightarrow T$$

$$\frac{\Gamma \vdash e_a : T_D \qquad \Gamma \vdash e_f : T_D \Rightarrow T_R}{\Gamma \vdash (e_f\ e_a) : T_R}$$

$$\not\frac{\Gamma[x \mapsto T_D] \vdash e_b : T_R}{\Gamma \vdash \lambda x.\ e_b : T_D \Rightarrow T_R}$$

$$\frac{\Gamma[x \mapsto T_D] \vdash e_b : T_R}{\Gamma \vdash \lambda x{:}T_D.\ e_b : T_D \Rightarrow T_R}$$

$\forall x \in V,\ e_b \in e,\ T_D \in T,\ T_R \in T,\ \Gamma \in \Gamma_s,$

if $(\Gamma)\ [x \mapsto T_D] \vdash (e_b) : [T_R]$ then

$(\Gamma) \vdash (\lambda x.\ e_b) : T_D \Rightarrow [T_R]$

$\lceil$ type $\Gamma\ (\lambda x.\ e_b) = T_D \Rightarrow T_R$

where T_R = type $\Gamma[x \mapsto T_D]\ e_b$

T_D = ???

Type taxes

1. Annotations to make type system computable / faster
2. Changing program to make it happy
   (if (=5 5) 6 false)

24-2 Gradual Typing

e := ... | (untyped T u)

u := just like e., but no annotations
λ | (typed e)

OLD: $\Gamma \vdash e : T$

NEW: $\Gamma \vdash e : T \rightsquigarrow u$

$$\frac{}{\Gamma \vdash num : Num \rightsquigarrow num}$$

$$\frac{\Delta(p) = (T_1 \ldots T_n) \to T \qquad \Gamma \vdash e_1 : T_1 \rightsquigarrow e_1' \ \ldots}{\Gamma \vdash (p\ e_1 \ldots e_n) : T \rightsquigarrow (\#\%p\ e_1' \ldots e_n')}$$

$C = CTC_+(T)$ $\qquad$ $\Gamma \vdash e \rightsquigarrow u'$

$$\frac{}{\Gamma \vdash (untyped\ T\ e) : T \rightsquigarrow (contract\ C\ u'\ "untyped"\ "typed")}$$

p ⇒ #%p

+ ⇒ unsafe-+

$\Gamma \vdash u \rightsquigarrow u'$ just goes through u unchanged

$CTC_+(Bool) = bool?$ $\qquad$ $CTC_+(Num) = num?$

$CTC_+(T_D \to T_R) = CTC_-(T_2) \to CTC_+(T_R)$ $\qquad$ ...

$CTC_-(B) = any$ $\qquad$ $CTC_-(Num) = any$

$CTC_-(T_D \to T_R) = CTC_+(T_D) \to CTC_-(T_R)$

$$\frac{\Gamma \vdash e : T \rightsquigarrow u}{\Gamma \vdash (typed\ e) \rightsquigarrow u}$$

243/ recursion

$$\frac{\Gamma[x \mapsto T_D][rec \mapsto T_D \to T_R] \vdash e : T_R}{\Gamma \vdash (\lambda x.\, e) : T_D \to T_R}$$

data

$$T = \ldots \mid T \times T \mid T + T \mid 1 \mid 0$$

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash pair\ e_1\ e_2 : T_1 \times T_2}$$

$$\frac{\Gamma \vdash e : T_1}{\Gamma \vdash inl\ e : T_1 + T_2} \qquad \frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash fst\ e : T_1}$$

inl e → inl e T

inr e → inr T e

$$\frac{\Gamma \vdash e : T_2}{\Gamma \vdash inr\ T_1\ e : T_1 + T_2}$$

$$\frac{\Gamma \vdash e_s : T_1 + T_2 \quad \Gamma[x \mapsto T_1] \vdash e_1 : T \quad \Gamma[y \mapsto T_2] \vdash e_2 : T}{\Gamma \vdash case\ e_s\ of\ inl\ x \Rightarrow e_1 : T \quad inr\ y \Rightarrow e_R}$$

24-4 / T = ... | Box T

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{box } e : \text{Box } T} \qquad \frac{\Gamma \vdash e : \text{Box } T}{\Gamma \vdash \text{unbox } e : T}$$

$$\frac{\Gamma \vdash e_b : \text{Box } T \quad \Gamma \vdash e_v : T}{\Gamma \vdash \text{set-box! } e_b\ e_v : 1}$$

$$\frac{\Gamma \vdash e : (\text{Kont } A) \Rightarrow A}{\Gamma \vdash \text{call/cc } e : A}$$

Kont X =
X ⇒ 0
≃ X ⇒ False
≃ ¬X
1 ≙ True
0 ≙ False

Curry-Howard Isomorphism

programs	<=>	proofs
:		:
types	<=>	propositions

HOL Isabelle **Coq** Agda

24-5/ List A = 1 + (A × List A)

1: "A" List Num List Bool

2: List A is defined as List A

1 = Polymorphism 2 = Recursive Types

Polymorphism

T = ... | A | ∀A.T

e = ... | Λ A.e | e[T]

∀A. A → A

∀A. A → Num

λx.5 λx.x

(e1 [Num] 6)

e1 = ΛA.λx:A.5

ΛA.λx:A.x

$$\frac{\Gamma, A \vdash e : T}{\Gamma \vdash \Lambda A.e : \forall A.T} \qquad \frac{\Gamma \vdash T \quad \Gamma \vdash e : \forall A.T'}{\Gamma \vdash e[T] : T'[A \leftarrow T]}$$

$$\frac{\Gamma \vdash T \quad \Gamma[x \mapsto T] \vdash e : T'}{\Gamma \vdash \lambda x{:}T.e : T \to T'}$$

class List<X> { X elem; List<X> next; }

auto l = new List<Num>();

e[T]

24-6) $T = \ldots \mid \mu A.\, T$

$e = \ldots \mid \text{fold } e \mid \text{unfold } e$

$$\frac{\Gamma \vdash e : T[A \leftarrow \mu A.\, T]}{\Gamma \vdash \text{fold } e : \mu A.\, T}$$

$$\frac{\Gamma \vdash e : \mu A.\, T}{\Gamma \vdash \text{unfold } e : T[A \leftarrow \mu A.\, T]}$$

constructors always call fold

accessors always unfold

"data List A = Empty | Node A (List A)"

$\Rightarrow$ Empty = ... fold ...

Node v l = ... fold ...

caseList l mt node = ... unfold ...

case ... mt

node v l