# State

CS242

Lecture 7

# Recall: The Lambda Calculus

Grammar:

$e \rightarrow x \mid \lambda x.e \mid e\ e$

Beta reduction:

$(\lambda x.e_1)\ e_2 \rightarrow e_1[x := e_2]$

(Keep in mind the two arrows on the two lines mean different things!)

# Recall The Ugly Bit: Substitution

$x [x := e] = e$

$y [x := e] = y$

$(e_1 e_2) [x := e] = (e_1 [x := e]) (e_2 [x := e])$

$(\lambda x.e_1) [x := e] = \lambda x.e_1$

$(\lambda y.e_1) [x := e] = \lambda y.(e_1 [x := e])$  if $x \neq y$ and $y \notin FV(e)$

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

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

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

*Is there a better way of explaining beta reduction?*

# Recall: Type Rules

$$\frac{}{A, x: t \vdash x: t} \quad \text{[Var]}$$

$$\frac{}{A \vdash i: \text{int}} \quad \text{[Int]}$$

$$\frac{A, x: t \vdash e: t'}{A \vdash \lambda x{:}t.e: t \to t'} \quad \text{[Abs]}$$

$$\frac{A \vdash e_1: t \to t' \quad A \vdash e_2: t}{A \vdash e_1\, e_2: t'} \quad \text{[App]}$$

# Structural Operational Semantics

- Logical systems have long been used to present type rules

- Why not do the same thing for evaluation rules?

- SOS were introduced by Gordon Plotkin
  - Now the most popular method for describing language semantics

# Judgments

Type judgments

$A \vdash e : t$

*Under type assumptions A for the free variables of e, it is provable that e has type t.*

# Judgments

Value judgments

E ⊢ e → v

E is an *environment*: A mapping from variables to values.

*In the environment E assigning values to the free variables of e, it is provable that e reduces to value v.*

# Comments

What are values?

*A subset of the programs that can't be further reduced.*

In the pure lambda calculus, lambda abstractions λx.e are the values.

If we add constants such as 1,2,..., true, false, +, *, ... these would also be values

- possible results of computations

# Evaluation Rules

$$\frac{}{E \vdash x \to E(x)} \quad \text{[Var]}$$

$$\frac{}{E \vdash \lambda x.e \to \langle \lambda x.e, E \rangle} \quad \text{[Abs]}$$

$$\frac{}{E \vdash i \to i} \quad \text{[Int]}$$

$$\frac{E \vdash e_1 \to \langle \lambda x.e_0, E' \rangle \quad E \vdash e_2 \to v \quad E'[x: v] \vdash e_0 \to v'}{E \vdash e_1\, e_2 \to v'} \quad \text{[App]}$$

Note: $E[x: v]$ is the same environment as $E, x:v$. $E$ is extended (or updated if $x$ is already present) at point $x$ to return $v$.

# Example

$$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$$

$$\vdash 1 \rightarrow 1$$

$$x:1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x:1]>$$

---

$$\vdash \lambda x.x \rightarrow <\lambda x.x,[]> \qquad \vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x:1]> \qquad x: <\lambda y.x,[x:1]> \vdash x \rightarrow <\lambda y.x,[x:1]>$$

---

$$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x:1]> \qquad\qquad \vdash 2 \rightarrow 2 \qquad x:1, y:2 \vdash x \rightarrow 1$$

---

$$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow 1$$

# Example

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

# Example

$$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow \qquad\qquad \vdash 2 \rightarrow$$

$$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$$

# Example

⊢ λx.x →                    ⊢ (λx.λy.x) 1 →
_____

       ⊢ (λx.x) ((λx.λy.x) 1) →                         ⊢ 2 →
       _____

               ⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

⊢ λx.x → <λx.x,[]>        ⊢ (λx.λy.x) 1 →
──────────────────────────────────────────────────────
         ⊢ (λx.x) ((λx.λy.x) 1) →                              ⊢ 2 →
         ──────────────────────────────────────────────────────
              ⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

⊢ λx.λy.x →

⊢ 1 →

⊢ λx.x → <λx.x,[]>     ——————————————————
                        ⊢ (λx.λy.x) 1 →
———————————————————————————————————————————

     ⊢ (λx.x) ((λx.λy.x) 1) →                          ⊢ 2 →
     ————————————————————————————————————————————————————————

          ⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow$

_____

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$     $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow$

_____

$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow$                    $\vdash 2 \rightarrow$

_____

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

# Example

⊢ λx.λy.x → <λx.λy.x,[]>

⊢ 1 → 1

_____

⊢ λx.x → <λx.x,[]>         ⊢ (λx.λy.x) 1 →

_____

⊢ (λx.x) ((λx.λy.x) 1) →                                    ⊢ 2 →

_____

⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow 1$

$x:1 \vdash \lambda y.x \rightarrow$

---

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$     $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow$

---

$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow$         $\vdash 2 \rightarrow$

---

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

# Example

$\vdash \lambda x.\lambda y.x \rightarrow \langle\lambda x.\lambda y.x,[]\rangle$

$\vdash 1 \rightarrow 1$

$x{:}1 \vdash \lambda y.x \rightarrow \langle\lambda y.x,[x{:}1]\rangle$

$\vdash \lambda x.x \rightarrow \langle\lambda x.x,[]\rangle$ $\qquad$ $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow$

$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow$ $\qquad\qquad\qquad$ $\vdash 2 \rightarrow$

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow 1$

$x:1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x:1]>$

─────────────────────────────────

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$   $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x:1]>$

──────────────────────────────────────────────────

$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow$   $\vdash 2 \rightarrow$

──────────────────────────────────────────────────

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

# Example

⊢ λx.λy.x → <λx.λy.x,[]>

⊢ 1 → 1

x:1 ⊢ λy.x → <λy.x,[x:1]>
_____

⊢ λx.x → <λx.x,[]>          ⊢ (λx.λy.x) 1 → <λy.x,[x:1]>          x: <λy.x,[x:1]> ⊢
_____

⊢ (λx.x) ((λx.λy.x) 1) →                                              ⊢ 2 →
_____

⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

⊢ λx.λy.x → <λx.λy.x,[]>

⊢ 1 → 1

x:1 ⊢ λy.x → <λy.x,[x:1]>

────────────────────────────

⊢ λx.x → <λx.x,[]>     ⊢ (λx.λy.x) 1 → <λy.x,[x:1]>     x: <λy.x,[x:1]> ⊢ x → <λy.x,[x:1]>

──────────────────────────────────────────────────────────────────────

⊢ (λx.x) ((λx.λy.x) 1) →                          ⊢ 2 →

──────────────────────────────────

⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

⊢ λx.λy.x → <λx.λy.x,[]>

⊢ 1 → 1

x:1 ⊢ λy.x → <λy.x,[x:1]>

⊢ λx.x → <λx.x,[]>          ⊢ (λx.λy.x) 1 → <λy.x,[x:1]>          x: <λy.x,[x:1]> ⊢ x → <λy.x,[x:1]>

⊢ (λx.x) ((λx.λy.x) 1) → <λy.x,[x:1]>                                              ⊢ 2 →

⊢ ((λx.x) ((λx.λy.x) 1)) 2 →

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow 1$

$x:1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x:1]>$

---

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$      $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x:1]>$      $x: <\lambda y.x,[x:1]> \vdash x \rightarrow <\lambda y.x,[x:1]>$

---

$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x:1]>$      $\vdash 2 \rightarrow 2$

---

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow 1$

$x{:}1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x{:}1]>$

---

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$      $\vdash (\lambda x.\lambda y.x) \; 1 \rightarrow <\lambda y.x,[x{:}1]>$      $x{:} <\lambda y.x,[x{:}1]> \vdash x \rightarrow <\lambda y.x,[x{:}1]>$

---

$\vdash (\lambda x.x) \; ((\lambda x.\lambda y.x) \; 1) \rightarrow <\lambda y.x,[x{:}1]>$      $\vdash 2 \rightarrow 2$      $x{:}1, y{:}2 \vdash x \rightarrow$

---

$\vdash ((\lambda x.x) \; ((\lambda x.\lambda y.x) \; 1)) \; 2 \rightarrow$

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow 1$

$x:1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x:1]>$

---

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$     $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x:1]>$     $x: <\lambda y.x,[x:1]> \vdash x \rightarrow <\lambda y.x,[x:1]>$

---

$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x:1]>$     $\vdash 2 \rightarrow 2$     $x:1,\ y:2 \vdash x \rightarrow 1$

---

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow$

Alex Aiken     CS 242     Lecture 7

# Example

$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$

$\vdash 1 \rightarrow 1$

$x:1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x:1]>$

$\overline{\phantom{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}$

$\vdash \lambda x.x \rightarrow <\lambda x.x,[]>$     $\vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x:1]>$     $x: <\lambda y.x,[x:1]> \vdash x \rightarrow <\lambda y.x,[x:1]>$

$\overline{\phantom{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}$
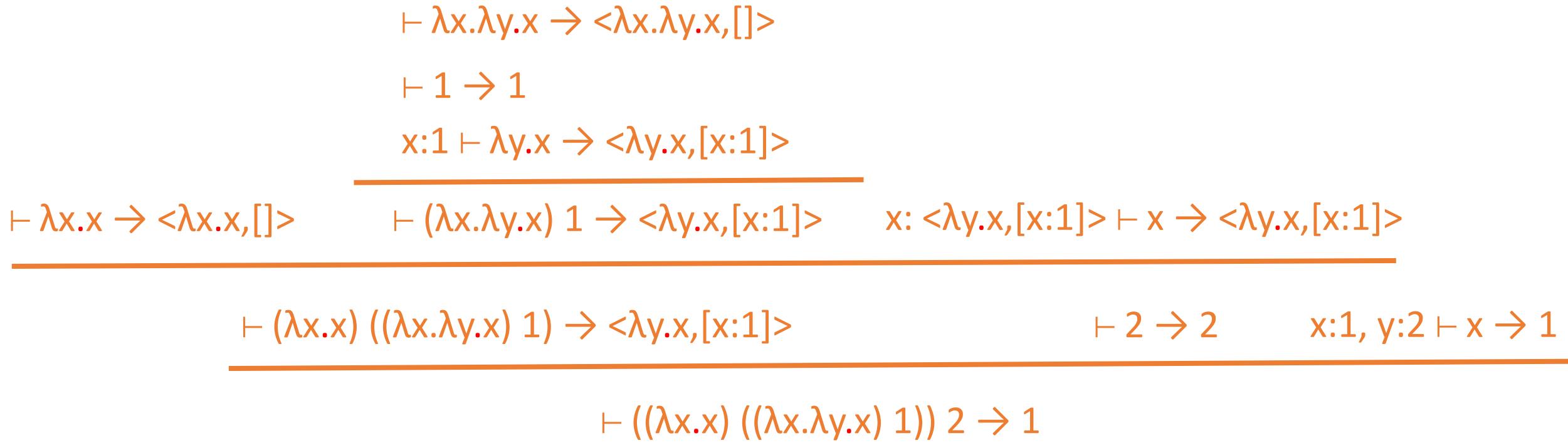
$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x:1]>$     $\vdash 2 \rightarrow 2$     $x:1,\ y:2 \vdash x \rightarrow 1$

$\overline{\phantom{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}$

$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow 1$

# Example

$$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$$

$$\vdash 1 \rightarrow 1$$

$$x{:}1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x{:}1]>$$

---

$$\vdash \lambda x.x \rightarrow <\lambda x.x,[]> \qquad \vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x{:}1]> \qquad x{:}<\lambda y.x,[x{:}1]> \vdash x \rightarrow <\lambda y.x,[x{:}1]>$$

---

$$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x{:}1]> \qquad \vdash 2 \rightarrow 2 \qquad x{:}1,\ y{:}2 \vdash x \rightarrow 1$$

---

$$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow 1$$

The tree structure of the derivation reflects the structure of the computation, not the structure of the expression.   Interior nodes are function calls (applications), leaves are constants and abstractions  that evaluate to themselves or variables that evaluate to their binding in the environment.

# Example

$$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$$

$$\vdash 1 \rightarrow 1$$

$$x:1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x:1]>$$

---

$$\vdash \lambda x.x \rightarrow <\lambda x.x,[]> \qquad \vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x:1]> \qquad x: <\lambda y.x,[x:1]> \vdash x \rightarrow <\lambda y.x,[x:1]>$$

---

$$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x:1]> \qquad\qquad \vdash 2 \rightarrow 2 \qquad x:1,\ y:2 \vdash x \rightarrow 1$$

---

$$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow 1$$

Note how substitution is built into the rules: The binding of a variable is passed down through the tree from the point of definition (at the function application) to the point where the variable is used via the environment.

# Example

$$\vdash \lambda x.\lambda y.x \rightarrow <\lambda x.\lambda y.x,[]>$$

$$\vdash 1 \rightarrow 1$$

$$x{:}1 \vdash \lambda y.x \rightarrow <\lambda y.x,[x{:}1]>$$

---

$$\vdash \lambda x.x \rightarrow <\lambda x.x,[]> \qquad \vdash (\lambda x.\lambda y.x)\ 1 \rightarrow <\lambda y.x,[x{:}1]> \qquad x{:}<\lambda y.x,[x{:}1]> \vdash x \rightarrow <\lambda y.x,[x{:}1]>$$

---

$$\vdash (\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1) \rightarrow <\lambda y.x,[x{:}1]> \qquad\qquad \vdash 2 \rightarrow 2 \qquad x{:}1,\ y{:}2 \vdash x \rightarrow 1$$

---

$$\vdash ((\lambda x.x)\ ((\lambda x.\lambda y.x)\ 1))\ 2 \rightarrow 1$$

The evaluation rules also capture aspects of implementations. The environment, for example, represents the call stack.

# Another Example

$$\frac{}{x: <\lambda x.x\ x,[]> \vdash x \to <\lambda x.x\ x,[]>} \quad \frac{}{x: <\lambda x.x\ x,[]> \vdash x \to <\lambda x.x\ x,[]>} \quad x: <\lambda x.x\ x,[]> \vdash x\ x \to$$

$$\frac{}{\vdash \lambda x.x\ x \to <\lambda x.x\ x,\ []>} \quad \frac{}{\vdash \lambda x.x\ x \to <\lambda x.x\ x,\ []>} \quad x: <\lambda x.x\ x,[]> \vdash x\ x \to$$

$$\vdash (\lambda x.x\ x)\ (\lambda x.x\ x) \to$$

Proofs are finite objects and as a result infinite computations have no derivations in this semantics!  There are more elaborate versions of SOS that address this issue.

*But isn't this lecture supposed to be about state?*

# The Lambda Calculus with State

e → x  |  λx.e  |  e e  |  i  |  new  |  !e  |  e := e

new            allocate a new memory location $x$ and return a pointer to $x$
               (initialize $x$ to $0$)
!e             dereference a pointer
$e_1 := e_2$   assign to a pointer, value is $e_2$

# Example

e → x | λx.e | e e | i | new | !e | e := e

!((λy.K y (y := 1)) new) → 1

# Rules?

$$\frac{}{E \vdash \text{new} \to} \text{[New]}$$

*What is state?*

# Stores

A *store* is a mapping from memory locations to values.

$S = [\ l_1 = 1,\ l_2 = 42\ ]$

1 is stored at location $l_1$

42 is stored at location $l_2$

Note that locations are values.

# New

$$l \notin dom(S)$$

$$E, S \vdash new \rightarrow l, S[l = 0]$$

[New]

*Evaluating an expression takes place in an initial store and produces a value and a new store.*

# Dereference

$$\frac{E, S_0 \vdash e \rightarrow l, S_1}{E, S_0 \vdash\ !e \rightarrow S_1(l), S_1} \; \text{[Ref]}$$

# Assignment

$$E, S_0 \vdash e_1 \rightarrow l, S_1$$

$$E, S_1 \vdash e_2 \rightarrow v, S_2$$

$$\overline{\rule{0pt}{0pt}\hspace{6cm}} \quad \text{[Assign]}$$

$$E, S_0 \vdash e_1 := e_2 \rightarrow v, S_2[l = v]$$

Note how the presence of a store requires us to specify the order of evaluation so that store updates happen in a predictable order. Also note how the store is "threaded" through the computation.

# Modified Evaluation Rules

$$\frac{}{E, S \vdash x \to E(x), S} \text{ [Var]}$$

$$\frac{}{E, S \vdash \lambda x.e \to <\lambda x.e, E>, S} \text{ [Abs]}$$

$$\frac{}{E, S \vdash i \to i, S} \text{ [Int]}$$

$$\frac{\begin{array}{c} E, S_0 \vdash e_1 \to <\lambda x.e_0, E'>, S_1 \\ E, S_1 \vdash e_2 \to v, S_2 \\ E'[x: v], S_2 \vdash e_0 \to v', S_3 \end{array}}{E, S_0 \vdash e_1\, e_2 \to v', S_3} \text{ [App]}$$

Even expressions that don't use the store directly need to thread it through the evaluation in case subexpressions do use the store.

# Example with State

$[y:l_0], [\ l_0 = 0\ ] \vdash y \rightarrow l_0, [\ l_0 = 0\ ]$

$[y:l_0], [\ l_0 = 0\ ] \vdash 1 \rightarrow 1, [\ l_0 = 0\ ]$

---

$[y:l_0], [\ l_0 = 0\ ] \vdash y \rightarrow l_0, [\ l_0 = 0\ ]$ ꘎ $[y:l_0], [\ l_0 = 0\ ] \vdash y := 1 \rightarrow 1, [\ l_0 = 1\ ]$

---

$[y:l_0], [\ l_0 = 0\ ] \vdash K\ y\ (y := 1) \rightarrow l_0, [\ l_0 = 1\ ]$

---

$[], [] \vdash \lambda y.K\ y\ (y := 1) \rightarrow \langle\lambda y.K\ y\ (y := 1), []\rangle, []$ ꘎ $[], [] \vdash new \rightarrow l_0, [\ l_0 = 0\ ]$

---

$[], [] \vdash (\lambda y.K\ y\ (y := 1))\ new \rightarrow l_0, [l_0 = 1]$

---

$[], [] \vdash !((\lambda y.K\ y\ (y := 1))\ new) \rightarrow 1, [l_0 = 1]$

$E, S_0 \vdash e_1 \rightarrow v_1, S_1$

$E, S_1 \vdash e_2 \rightarrow v_2, S_2$

---

$E, S_0 \vdash K\ e_1\ e_2 \rightarrow v_1, S_2$

Evaluation rule for primitive K combinator

# Discussion

- The operational semantics provides a precise model of state.

- The model is also close to how most programming languages work.

- Observe
  - The store is like an extra argument passed to every expression evaluation.
  - The store is never copied: No evaluation step uses more than one store at a time, and every store is modified at most once.
  - The state is unstructured – the whole store is passed to every step of evaluation even if only a part of it is used

# Summary

- Explaining state requires a significant extension of simple functional semantics
  - Need a store object with allocation, reads, and writes
  - Behavior of programs is much different than pure functions

- Semantics of state shows the costs
  - Sequential order of evaluation must be defined
  - Uncontrolled aliasing
  - Unstructured state exposes too much