Analysis Sensitivity / Polyvariance

Kristopher Micinski
CIS 700 —Program Analysis: Foundations and Applications
Fall ’19, Syracuse University

AAM is a **general** strategy for building abstract interpreters from abstract machines

But what do the results **mean**?

For the first part of this lecture—let's just figure out the answer intuitively (without implementing the analysis)

This week we will study **finite** analyses

Today, will study 0CFA—finite flow analysis for Scheme

On Wednesday, k-CFA—Improves precision of 0CFA

0CFA

Asks the question:
for each **variable**, which possible **lambdas** could be **bound** to that variable?

```
((lambda (x) x) (lambda (y) y))
```

Nothing..

The issue is that this could require transitive reasoning…

```
(((lambda (x) x) (lambda (y) y)) (lambda (z) z))
```

```
((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z)))
```

```
(((lambda (x) x)
  (if #t (lambda (y) y) (lambda (z) z)))
 (lambda (a) a))
```

Practice: what flows where?

Data flow depends on **control** flow

```
((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z)))
```

```
(((lambda (x) x)
  (if #t (lambda (y) y) (lambda (z) z)))
 (lambda (a) a))
```

Data flow depends on **control** flow

This is an approximation, because at runtime **lambdas** don't get bound to variables, but **closures** do.

0CFA conflates all possible environments that could be closed alongside a piece of syntax

0CFA "ignores" the environment

0CFA

```
((lambda (x) x) (lambda (y) y))
```

$\rho = []$

Closure

Thinking back to last time…

$\hat{\varsigma} \longmapsto_{\widehat{CESK}^{\star}_{t}} \hat{\varsigma}'$, where $\kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 5. The abstract time-stamped CESK* machine.

[Van Horn and Might, '10]

Defines a **family** of interpreters
(Instantiate by choosing alloc / tick appropriately.)

$\hat{\varsigma} \longmapsto_{\widehat{CESK}^\star_t} \hat{\varsigma}'$, where $\kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 5. The abstract time-stamped CESK* machine.

[Van Horn and Might, ’10]

In this lecture, we'll look at 0CFA…

$\hat{\varsigma} \longmapsto_{\widehat{CESK^\star_t}} \hat{\varsigma}'$, where $\kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 5. The abstract time-stamped CESK* machine.

[Van Horn and Might, '10]

Today’s insight

- We can define a **tiny language**—namely the lambda calculus—that encapsulates a Turing-equivalent formulation of our language.

- We can give this language a simple machine-based semantics via textual substitution (term-rewriting).

- Therefore, we will have encapsulated a fully-precise model of Core Scheme by the combination of (a) applications of the desugarings, and (b) the simple term-rewriting semantics of the lambda calculus.

The Lambda Calculus

- A system for calculating based entirely on computing with functions.
- Developed as a foundation for mathematics (originally used to model the natural numbers) by ***Alonzo Church*** in 1936.
- Church's thesis: *"Every effectively calculable function (effectively decidable predicate) is general recursive"*, i.e., can be computed using the λ-calculus. Used to show there exist unsolvable problems.
- One of the simplest Turing-equivalent languages!
    - Church, with his student Alan Turing, proved the equivalent expressiveness of Turing machines and the λ-calculus (called the ***Church-Turing thesis***).
- Still makes up the heart of all functional programming languages!

The Lambda Calculus

lambdas are just anonymous functions!

$e \in \textbf{Exp} ::= (\lambda\ (x)\ e)$	λ-abstraction
$\mid (e\ e)$	function application
$\mid x$	variable reference

$$x \in \textbf{Var} ::= \langle \textbf{variables} \rangle$$

Textual-reduction semantics

- One way of designing a formal semantics is as a relation over terms in the language—one that reduces the term textually.

- This is usually ***small-step***—each eval step must terminate (meaning there are no *premises above the line* in our rules of inference and no recursive use of the interpreter within a step.)

- Consider a small-step semantics for our arithmetic language:

$$a \in \textbf{AExp} \;\; ::= \;\; n \mid a + a \mid a - a \mid a \times a$$

$$n, m \in \textbf{Num} \;\; ::= \langle \textbf{integer constants} \rangle$$

Textual-reduction semantics

$$a \in \textbf{AExp} \ ::= \ n \mid a + a \mid a - a \mid a \times a$$

$$n, m \in \textbf{Num} \ ::= \langle \textbf{integer constants} \rangle$$

- Rules to reduce terms in this language match operations that have two numeric operands already and apply the operation, textually substituting a numeric value for the operation; e.g.:

$$\frac{}{a_0 \times a_1 \Rightarrow n_0 * n_1}$$ **where** a_0 is n_0 and a_1 is n_1

- For example: $2 * 3 + 4 * 5 \Rightarrow 2 * 3 + 20 \Rightarrow 6 + 20 \Rightarrow 26$

- Is there another way to evaluate 2*3 + 4*5 using similar rules?

Big Step Interpreters

Interpreter works in one “big step”

```
(define (eval env e)
  …)
```

```
(eval env `(,e0 ,e1))
```

```
(eval env e0)
```

To evaluate (e0 e1), interpreter recursively calls **itself** to evaluate e0

Big Step Interpreters

Interpreter works in one “big step”

```
(define (eval env e)
  …)
```

```
(eval env `(,e0 ,e1))
```

Returns result of e0, then continues evaluation…

```
(eval env e0)
```

To evaluate (e0 e1), interpreter recursively calls **itself** to evaluate e0

In a big-step semantics, the recursive nature of expression evaluation is exposed via the **interpreter's stack**.

```
(((((e0 e1) e2) e3) e4) … en)
```

A call to evaluate a nested expression will push a frame onto the interpreter's stack for each subexpression.

We often think of the lambda calculus as **small-step**, as each rule transforms an input to an output term.

The Lambda Calculus

lambdas are just anonymous functions!

$$
\begin{aligned}
e \in \textbf{Exp} ::= &\ (\lambda\ (x)\ e) && \text{λ-abstraction} \\
\mid &\ (e\ e) && \text{function application} \\
\mid &\ x && \text{variable reference}
\end{aligned}
$$

$$x \in \textbf{Var} ::= \langle \textbf{variables} \rangle$$

The Lambda Calculus

The lambda-calculus is the functional core of Racket (as of other functional languages).

Just the following subset of Racket is Turing-equivalent!

$$e \in \textbf{Exp} ::= (\lambda\ (x)\ e) \qquad \texttt{(lambda (x) e)}$$

$$\mid\ (e\ e) \qquad \texttt{(e e)}$$

$$\mid\ x \qquad \texttt{x}$$

$$x \in \textbf{Var} ::= \langle\textbf{variables}\rangle$$

The Lambda Calculus

The lambda-calculus is the functional core of Racket (as of other functional languages).

Just the following subset of Racket is Turing-equivalent!

$$e \in \textbf{Exp} ::= (\lambda\ (x)\ e) \qquad \texttt{(lambda (x) e)}$$

$$\mid (e\ e) \qquad \texttt{(e e)}$$

$$\mid x \qquad \texttt{x}$$

$$x \in \textbf{Var} ::= \langle \textbf{variables} \rangle$$

Lambda Abstraction

An expression, *abstracted* over all possible values for a formal parameter, in this case, x.

$$(\lambda\ (x)\ e)$$

Formal parameter

Function body

Application

When the first expression is evaluated to a value (in this language, all values are functions!) it may be invoked / applied on its argument.

$$(e \;\; e)$$

Expression in ***function position***

Expression in ***argument position***

Variable

Variables are only defined/assigned when a function is applied and its parameter bound to an argument.

$$x$$

Variable reference

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

We define a rule for step-by-step evaluation called ***Beta-reduction***

$$\Big\downarrow \beta$$

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

$$\Big\downarrow \beta$$

```
((λ (x) x) (λ (x) x))
```

$$\Big\downarrow \beta$$

```
(λ (x) x)
```

Textual substitution. This says:
replace every x in E_0 with E_1.

$$\underbrace{\texttt{((}\lambda\texttt{ (x) }E_0\texttt{) }E_1\texttt{)}}_{\text{redex}} \rightarrow_\beta E_0[x \leftarrow E_1]$$

redex

(**red**ucible **ex**pression)

Note: this is a definition you will want to remember! The term "redex" is frequently used to mean reducible expression in many different types of semantics.

```
((λ (x) x) (λ (x) x))
```

$$\Big\downarrow \beta$$

```
x[x ← (λ (x) x)]
```

```
((λ (x) x) (λ (x) x))
```

$$\Big\downarrow \beta$$

```
(λ (x) x)
```

Try an example. Can you beta-reduce this term?
Can you beta-reduce it more than once?

```
((λ (x) (x x)) (λ (x) (x x)))
```

```
((λ (x) (x x)) (λ (x) (x x)))
```

β reduction may continue indefinitely (i.e., in non-terminating programs)

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

This specific program is known as Ω (Omega)

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

((λ (x) (x x)) (λ (x) (x x)))

β

Ω is the smallest non-terminating program!

((λ (x) (x x)) (λ (x) (x x)))

β

Note how it reduces to itself in a single step!

((λ (x) (x x)) (λ (x) (x x)))

β

((λ (x) (x x)) (λ (x) (x x)))

β

Evaluation with β reduction is nondeterministic!

```
(((λ (w) w) (λ (x) x)) ((λ (y) y) (λ (z) z)))
```

$$\Big\downarrow \beta$$

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

Evaluation with β reduction is nondeterministic!

```
(((λ (w) w) (λ (x) x)) ((λ (y) y) (λ (z) z)))
```

β

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

or!

β

```
(((λ (w) w) (λ (x) x)) (λ (z) z))
```

Try an example. Perform each possible β-reduction

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

How many different β-reductions are possible from the above?

Answer

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

$\Big\downarrow \beta$

```
((λ (x) (x x)) (λ (z) (z z)))
```

Can reduce inner redex…

Answer

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

$$\Big\downarrow \beta$$

```
((λ (y) ((λ (z) (z z)) y)) (λ (z) (z z)))
```

Or the outer redex.

Answer

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

$\downarrow \beta$

```
((λ (y) ((λ (z) (z z)) y)) (λ (z) (z z)))
```

Can’t reduce this since we don’t (yet) know about the particular value (function) z in call position.

Free variables

$$\textbf{FV} : \textbf{Exp} \rightarrow \mathscr{P}(\textbf{Var})$$

$$\textbf{FV}(x) \triangleq \{x\}$$

$$\textbf{FV}((\lambda\ (x)\ e_b)) \triangleq \textbf{FV}(e_b) \setminus \{x\}$$

$$\textbf{FV}(e_f\ e_a)) \triangleq \textbf{FV}(e_f)\ \cup\ \textbf{FV}(e_a)$$

Free variables

FV((x y)) = {x, y}

FV(((λ (x) x) y)) = {y}

FV(((λ (x) x) x)) = {x}

FV(((λ (y) ((λ (x) (z x)) x))) = {z, x}

Try an example. What are the free variables of each of the following terms?

```
((λ (x) x) y)

((λ (x) (x x)) (λ (x) (x x)))

((λ (x) (z y)) x)
```

Try an example. What are the free variables of each of the following terms?

```
((λ (x) x) y)
```

{y}

```
((λ (x) (x x)) (λ (x) (x x)))
```

{}

```
((λ (x) (z y)) x)
```

{x, y, z}

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\big\downarrow \beta$$

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) a)[a ← (λ (b) b)]
```

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) (λ (b) b))   ✘
```

Capture-avoiding substitution

$$E_0[x \leftarrow E_1]$$

$$x[x \leftarrow E] = E$$

$$y[x \leftarrow E] = y \quad \text{where} \quad y \neq x$$

$$(E_0 \ \ E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \ \ E_1[x \leftarrow E])$$

$$(\lambda \ (x) \ \ E_0)[x \leftarrow E] = (\lambda \ (x) \ \ E_0)$$

$$(\lambda \ (y) \ \ E_0)[x \leftarrow E] = (\lambda \ (y) \ \ E_0[x \leftarrow E])$$

$$\text{where} \quad y \neq x \quad \text{and} \quad y \notin FV(E)$$

β-reduction cannot occur when $y \in FV(E)$

Capture-avoiding substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) a)
```

✔

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

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

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

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

$$\Big\downarrow \beta$$

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

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

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

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

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

You cannot! This redex would require:

```
(λ (y) z)[z ← (λ (x) y)]
```

(y is free here, so it would be captured)

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

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

$\rightarrow_\alpha$ `(λ (y) ((λ (z) (λ (w) z)) (λ (x) y)))`

$\rightarrow_\beta$ `(λ (y) (λ (w) (λ (x) y)))`

Instead we alpha-convert first.

α - renaming

```
(λ (x) (λ (y) x))          (λ (a) (λ (b) a))
```

These two expressions are equivalent—they only differ by their variable names (x = a; y = b)

α - renaming

$$\texttt{(λ (x) E}_0\texttt{)} \quad \to_\alpha \quad \texttt{(λ (y) E}_0\texttt{[x} \leftarrow \texttt{y])}$$

$$=_\alpha$$

α renaming/conversions can be run backward,
so you might think of it as an equivalence relation

α - renaming

α renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

```
((λ (x) (λ (x) x)) (λ (y) y))
```

α - renaming

α renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

```
((λ (x) (λ (x) x)) (λ (y) y))
```

Can’t perform naive substitution w/o capturing x.

α - renaming

α renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

```
((λ (x) (λ (x) x)) (λ (y) y))
```

Fix by α renaming to z

α - renaming

α renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

```
((λ (x) (λ (z) z)) (λ (y) y))
```

Fix by α renaming to z

α - renaming

α renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

```
((λ (x) (λ (z) z)) (λ (y) y))
```

Could now perform beta-reduction with naive substitution

η - reduction

$$(\lambda\ (x)\ (E_0\ x)) \rightarrow_\eta E_0 \text{ where } x \notin FV(E_0)$$

η-expansion

`E0` $\rightarrow_\eta$ `(λ (x) (E0 x))` where $x \notin$ FV(`E0`)

Reduction

$$(\rightarrow) = (\rightarrow_\beta) \cup (\rightarrow_\alpha) \cup (\rightarrow_\eta)$$

$$(\rightarrow^*)$$

reflexive/transitive closure

Evaluation

$$\begin{tikzcd} E_0 \arrow[d, "*"'] \\ E_8 \arrow[d, "*"'] \\ ? \end{tikzcd}$$

Evaluation to *normal form*

$$E_0 \xrightarrow{*} \texttt{(λ (x) …)}$$

Evaluation to *normal form*

$$E_0 \xrightarrow{*} \texttt{(λ (x) … (λ (z) ((a …) …)))}$$

In ***normal form***, no function position can be a lambda;
this is to say: *there are no unreduced redexes left*!

Evaluation Strategy

$$\begin{tikzcd}
 & E_0 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
E_1 & & E_2
\end{tikzcd}$$

Evaluation Strategy

```
((λ (x) ((λ (y) y) x)) (λ (z) z))

        →η ((λ (y) y) (λ (z) z))

        →β (λ (z) z)
```

Evaluation Strategy

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

$\to_\beta$ `((λ (y) y) (λ (z) z))`

$\to_\beta$ `(λ (z) z)`

Evaluation Strategy

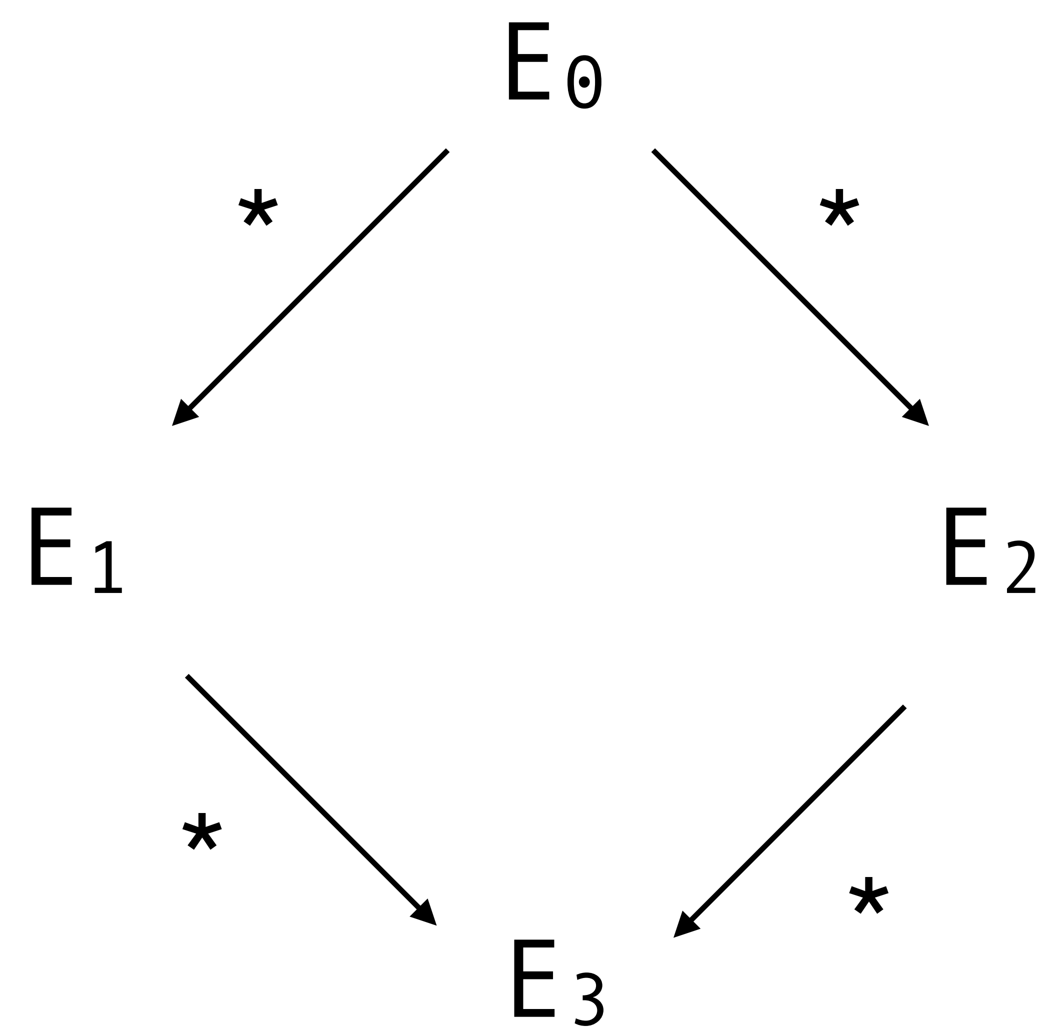
```
((λ (x) ((λ (y) y) x)) (λ (z) z))

        →β ((λ (x) x) (λ (z) z))

        →β (λ (z) z)
```

Confluence

Diverging paths of evaluation must eventually join back together.

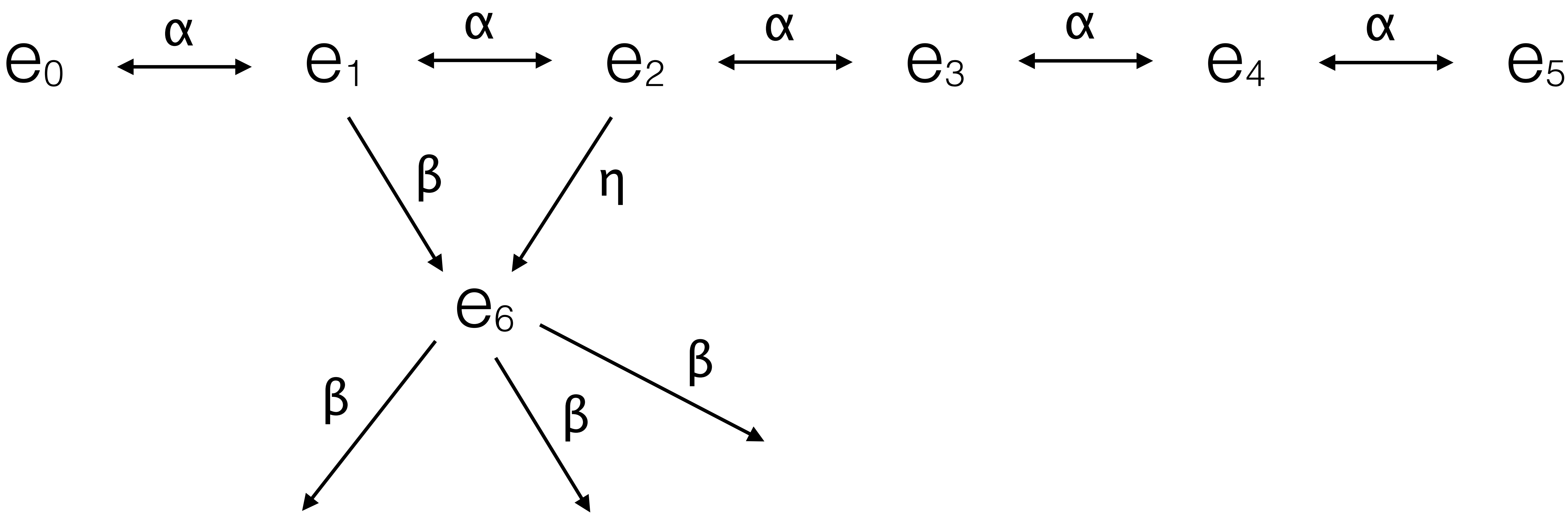
Church-Rosser Theorem

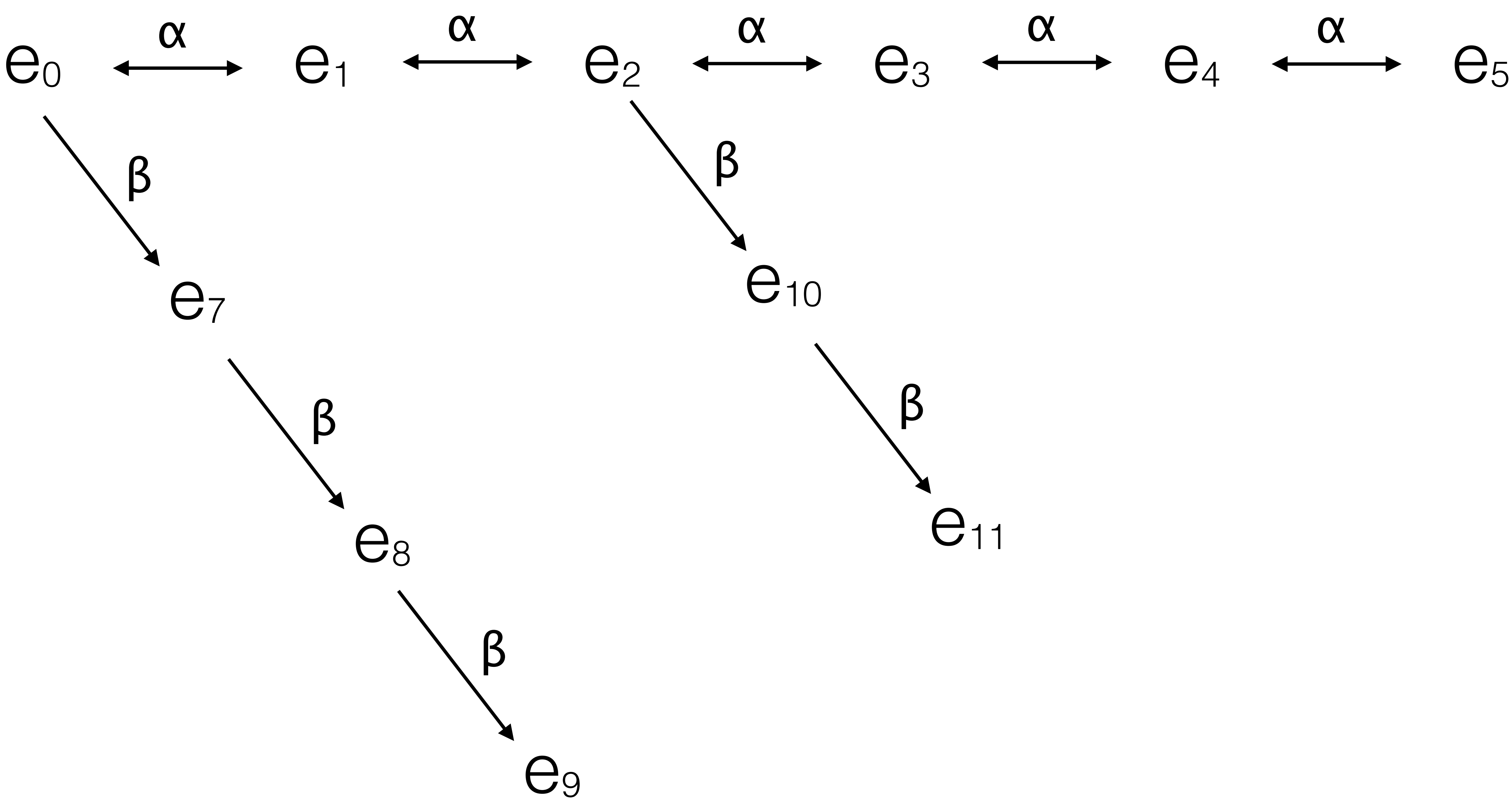
If—starting from E_0—we can reach *both* E_1 and E_2, then we **must** be able to get to some E_3 starting from *either* E_2 or E_1.



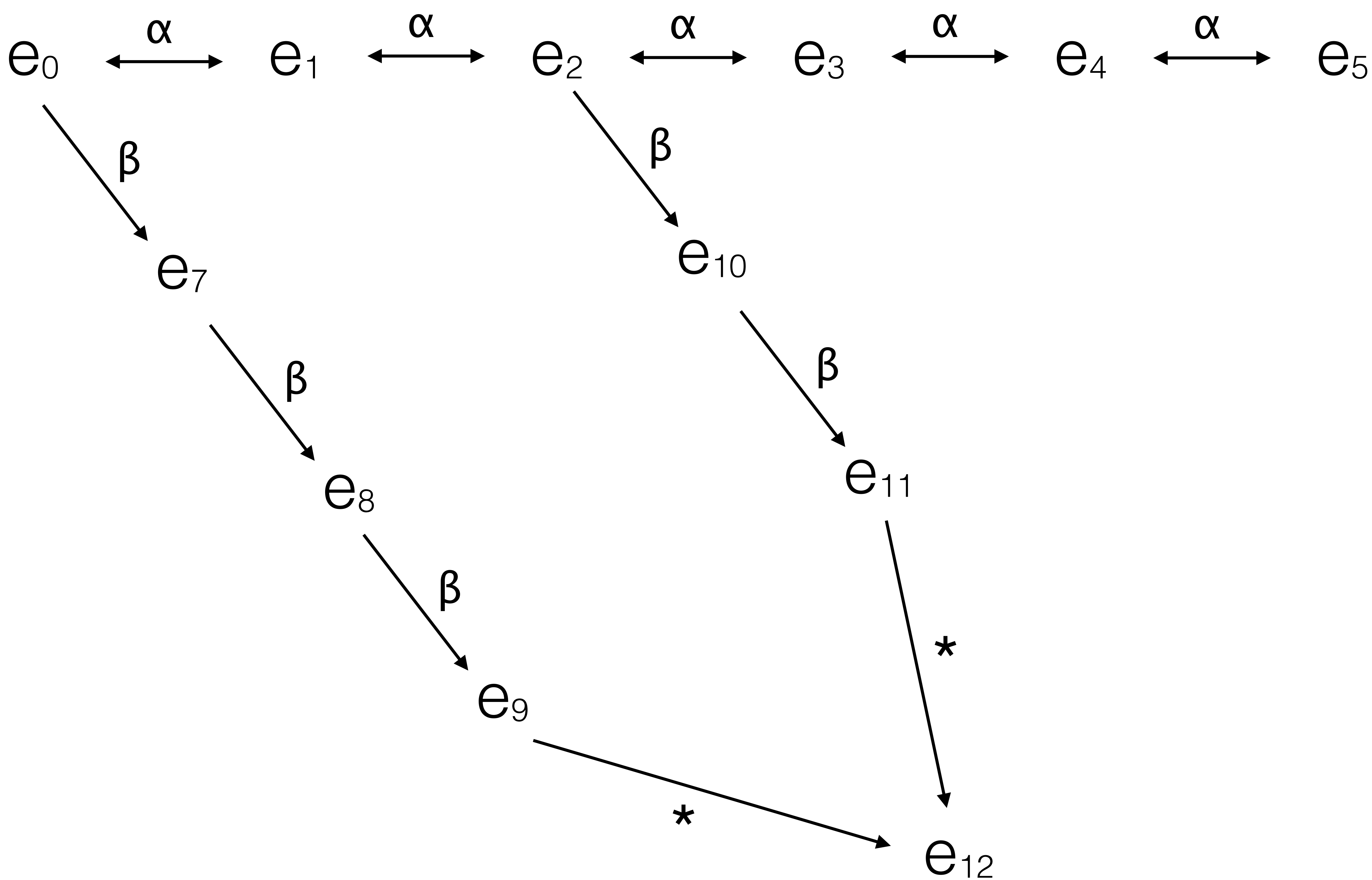
$$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$$

$$\begin{tikzcd}
e_0 \arrow[r, leftrightarrow, "\alpha"] & e_1 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_2 \arrow[r, leftrightarrow, "\alpha"] \arrow[dl, "\eta"] & e_3 \arrow[r, leftrightarrow, "\alpha"] & e_4 \arrow[r, leftrightarrow, "\alpha"] & e_5 \\
& & e_6 & & & &
\end{tikzcd}$$


e0
α
e1
α
e2
α
e3
α
e4
α
e5
β
η
e6
β
β
β


e0
α
e1
α
e2
α
e3
α
e4
α
e5
β
e7
β
e8
β
e9
β
e10
β
e11

Confluence (i.e., Church-Rosser Theorem)



Applicative evaluation order

Always evaluates the *innermost* leftmost redex first.

Normal evaluation order

Always evaluates the *outermost* leftmost redex first.

Applicative evaluation order

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

Normal evaluation order

```
(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))
```

Call-by-value (CBV) semantics

Applicative evaluation order, *but not under lambdas*.

Call-by-name (CBN) semantics

Normal evaluation order, *but not under lambdas*.

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

"If I only let you use the lambda calculus, can you still write normal programs (e.g., ones that use recursion/+/if/etc...)?"

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

"If I only let you use the lambda calculus, can you still write normal programs (e.g., ones that use recursion/+/if/etc…)?"

YES!

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

We will start from core scheme…

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\text{Scheme IR} \arrow[r, "\texttt{interp}"] \arrow[d, "\texttt{church-encode}"'] & 2 \\
\lambda\text{-calculus} \arrow[r, "\texttt{interp}"] & \texttt{(λ (f) (λ (x) (f (f x))))} \arrow[u, dashed, "\texttt{church->nat}"']
\end{tikzcd}
$$

Today, we'll start with:

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (+ e e) | (* e e)
    | (cons e e) | (car e) | (cdr e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
```

Desugaring Let

```
(let ([x e] …) ebody)
```

```
(let ([x e] …) ebody)

        ↓

((λ (x …) ebody) e …)
```

Currying

```
(λ (x y z) e)  ⟶  (λ (x)
                    (λ (y)
                      (λ (z) e)))

(λ (x) e)      ⟶  (λ (x) e)

(λ () e)       ⟶  (λ (_) e)
```

```
(f a b c d)   ⟶   ((((f a) b) c) d)

(f a)         ⟶   (f a)

(f)           ⟶   (f (λ (x) x))
```

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | (if e e e)
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
```

Conditionals & Booleans

$$\texttt{(if #t } e_T\ e_F\texttt{)} \longrightarrow e_T$$

$$\texttt{(if #f } e_T\ e_F\texttt{)} \longrightarrow e_F$$

$$
\begin{array}{cc}
\texttt{((}\lambda\ \texttt{(t f) t)}\ e_T\ e_F\texttt{)} & \texttt{((}\lambda\ \texttt{(t f) f)}\ e_T\ e_F\texttt{)} \\
\downarrow & \downarrow \\
\texttt{((}\lambda\ \texttt{(t f) t)}\ v_T\ v_F\texttt{)} & \texttt{((}\lambda\ \texttt{(t f) f)}\ v_T\ v_F\texttt{)} \\
\downarrow & \downarrow \\
v_T & v_F
\end{array}
$$

What issues arise with this encoding?

$$((\lambda\ (t\ f)\ t)\ e_T\ \Omega)$$

$$\downarrow$$

....

```
((λ (t f) (t)) (λ () e_T) (λ () Ω))
```

$$\downarrow$$

```
((λ () e_T))
```

$$\downarrow$$

$$e_T$$

$$\downarrow$$

$$v_T$$

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ℕ | ‘()
x ::= <vars>
```

Natural Numbers

Hint: turn all nouns into verbs!

(Focus on the *behaviors* that are implicit in values.)

```
(λ (f) (λ (x) (fᴺ x)))
```

$(\lambda\ (f)\ (\lambda\ (x)\ (f^N\ x)))$

```
0:  (λ (f) (λ (x) x))

1:  (λ (f) (λ (x) (f x)))

2:  (λ (f) (λ (x) (f (f x))))

3:  (λ (f) (λ (x) (f (f (f x)))))
```

```
church+ = (λ (n) (λ (m)
               (λ (f) (λ (x)
                      …))))
```

```
church+ = (λ (n) (λ (m)
             (λ (f) (λ (x)
               ((n f) ((m f) x))))))
```

```
church* = (λ (n) (λ (m)
             (λ (f) (λ (x)
                ...))))
```

```
church* = (λ (n) (λ (m)
            (λ (f) (λ (x)
              ((n (m f)) x))))))
```

$$fN^{M} = fN*M$$

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ‘()
x ::= <vars>
```

Lists

The fundamental problem:

We need to be able to case-split.

The solution:

We take two callbacks as with `#t`, `#f`!

```
‘() = (λ (when-cons) (λ (when-null)
          (when-null)))
```

```
(cons a b) = (λ (when-cons) (λ (when-null)
                 (when-cons a b)))
```

Try an Example. How can we define null?

Try an Example. How can we define null?

```
church:null? = (λ (p)
                 (p (λ (a b) #f)
                    (λ () #t)))
```

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
x ::= <vars>
```

Ω

```
((λ (x) (x x)) (λ (x) (x x)))
```

Key: U takes a function and calls it on itself

Ω

```
((λ (x) (x x)) (λ (x) (x x)))
```

U

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
               (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))])
  (fib 3))
```

What can I type right here to make fib work?

(Hint: the answer can be written in 5 characters)

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
               (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))])
  (fib 3))
```

What can I type right here to make fib work?

(f f)

Y combinator

```
(letrec ([fact (λ (n)
                 (if (= n 0)
                     1
                     (* n (fact (- n 1)))))])
  (fact 5))
```

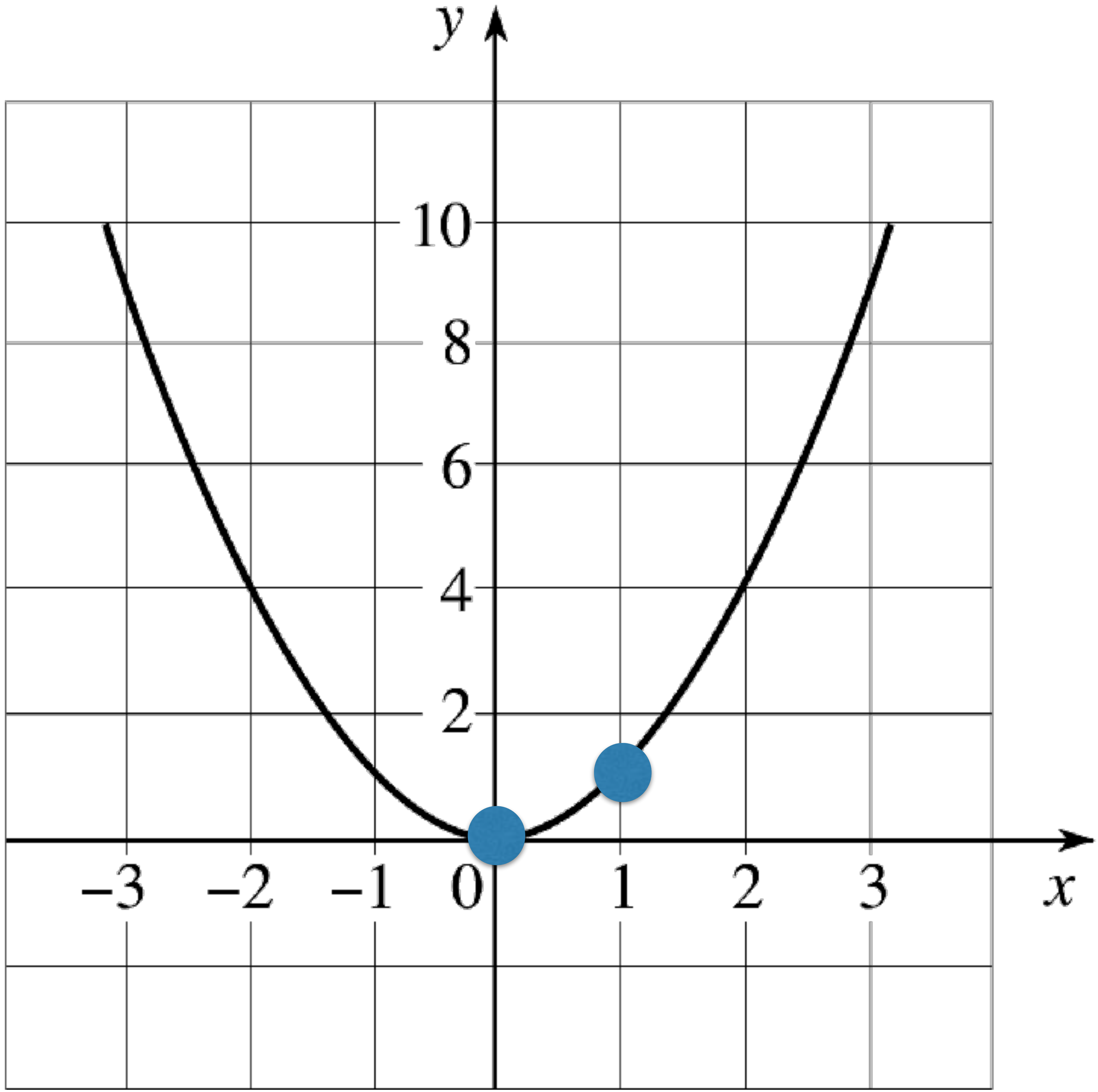
Key idea: instead of

```
(let ([mk (λ (mk) (λ (n)
                    (if (= n 0)
                        1
                        (* n ((mk mk) (- n 1))))))])
  ((mk mk) 5))
```

Y

$$(Y\ f) = f\ (Y\ f)$$

(It’s a fixed-point combinator!)


y
10
8
6
4
2
–3
–2
–1
0
1
2
3
x

Three step process for deriving Y

```
(Y f) = f (Y f)
```

```
Y = (λ (f) (f (Y f)))
```

1. Treat as definition

```
mY = (λ (mY)
       (λ (f)
         (f ((mY mY) f))))
```

2. Lift to mk-Y, use self-application

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

3. Eta-expand

U-combinator: `(U U)` **is Omega**

```
Y  =  (U (λ (y) (λ (f)
              (f (λ (x) (((y y) f) x))))))
```

```
(let ([fact (Y (λ (fact) (λ (n)
                  (if (= n 0)
                      1
                      (* n (fact (- n 1))))))])
  (fact 5))
```

Try an example!!!

```
(define Y ((λ (x) (x x)) (λ (y) (λ (f)
                                   (f (λ (x) (((y y) f) x)))))))
```

```
(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2)))))
```

Rewrite this to use the Y combinator instead

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

De-churching

```
(define (church->nat cv)
                        )
```

```
(define (church->list cv)





                              )
```

```
(define (church->bool cv)

                          )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)




                          )



(define (church->bool cv)

                       )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) ‘())))


(define (church->bool cv)

                          )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) ‘())))


(define (church->bool cv)
        ((cv (λ () #t))
         (λ () #f)))
```

```
(letrec ([map (λ (f lst)
                (if (null? lst)
                    '()
                    (cons (f (car lst))
                          (map f (cdr lst)))))])

  (map (λ (x) (+ 1 x))
       '(0 5 3)))
```

```
(define lst
  (((((((((λ (Y-comb)
            (λ (church:null?)
              (λ (church:cons)
                (λ (church:car)
                  (λ (church:cdr)
                    (λ (church:+)
                      (λ (church:*)
                        (λ (church:not)
                          ((λ (map)
                            ((map
                              (λ (x)
                                ((church:+ (λ (f) (λ (x) (f x))))
                                 x)))
                             ((church:cons (λ (f) (λ (x) x)))
                              ((church:cons
                                (λ (f)
                                  (λ (x) (f (f (f (f (f x))))))))
                               ((church:cons
                                 (λ (f) (λ (x) (f (f (f x))))))
                                (λ (when-cons)
                                  (λ (when-null)
                                    (when-null (λ (x) x)))))))))
                           (Y-comb
                            (λ (map)
```

```
> (map church->nat (church->list lst))
'(1 6 4)
```

Try an example.

Write a lambda term other than Ω which also does not terminate

(Hint: think about using some form of self-application)

Write a lambda term other than Ω which also does not terminate

```
((λ (y) ((λ (x) (y x)) y))
 (λ (y) ((λ (x) (y x)) y)))
```

```
((λ (u) ((u u) u))
 (λ (u) ((u u) u)))
```

```
((λ (x) x)
 ((λ (u) (u u))
  (λ (u) (u u))))
```

Evaluation contexts

Restrict the order in which we may simplify a program's redexes

$$\begin{aligned}\mathcal{E} \; ::= \; & (\mathcal{E} \;\; e) \\ | \; & (v \;\; \mathcal{E}) \\ | \; & \square\end{aligned}$$

(left-to-right) CBV evaluation

$$\begin{aligned}\mathcal{E} \; ::= \; & (\mathcal{E} \;\; e) \\ | \; & \square\end{aligned}$$

(left-to-right) CBN evaluation

```
v  ::=  (λ (x) e)

e  ::=  (λ (x) e)
     |  (e e)
     |  x
```

Context and redex

For CBV a redex must be `(v v)`
For CVN, a redex must be `(v e)`

$$\mathscr{E}[\,\overbrace{\texttt{(v v)}}^{\texttt{r}}\,] =$$

`(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))`

$\mathscr{E}$ = `(□ (λ (w) w))`

`r = ((λ (x) ((λ (y) y) x)) (λ (z) z))`

Context and redex

$$\mathscr{E}[\texttt{r}] =$$

```
(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))
```

$\mathscr{E}$ = `(□ (λ (w) w))`

```
r = ((λ (x) ((λ (y) y) x)) (λ (z) z))
```

$\rightarrow_\beta$ `((λ (y) y) (λ (z) z))`

Put the reduced redex back in its evaluation context…

```
𝓔 = (□ (λ (w) w))

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))
                   →β ((λ (y) y) (λ (z) z))
```

$$\Big\downarrow \; \mathcal{E}[r]$$

```
(((λ (y) y) (λ (z) z)) (λ (w) w))
```

Exercises—can you evaluate…

```
1)   (((λ (y) y) (λ (z) z)) (λ (w) w))


2)   ((λ (u) (u u)) (λ (x) (λ (x) x)))


3)   (((λ (x) x) (λ (y) y))
      ((λ (u) (u u)) (λ (z) (z z))))
```