The Lambda Calculus

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

This is a lecture/livecoding week. We will work through the slides and do some lecture, along with collaborative livecoding as we go. At the end of each lecture, I'll post the code.

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:

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

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

Exercise

Which of the following are **AExp**s:

- 10
- 20.5
- 10 + 3
- 10 + 3 * 4^2
- 5 - 3 * 2 + 3 * 1

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

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

Exercise

Which of the following are **AExp**s:

- **10**
- 20.5 — No, not integer constant
- **10 + 3**
- 10 + 3 * 4^2 — Exponent not allowed
- **5 - 3 * 2 + 3 * 1**

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

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

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?

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

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

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!

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

$$\mid (e\ e) \qquad \texttt{(e0 e1)}$$

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

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

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***

Variables

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))
```

$\big\downarrow \beta$

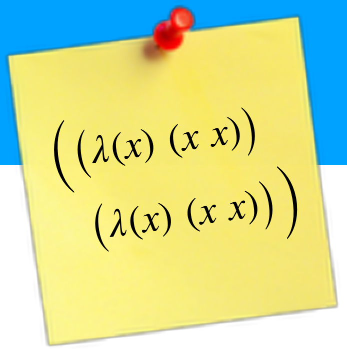
```
(λ (x) x)
```

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

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

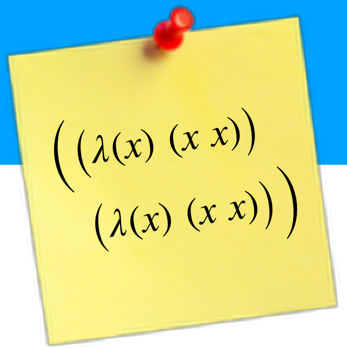
(**red**ucible **ex**pression)

Example



$$
\begin{array}{c}
\texttt{((}\lambda\texttt{ (x) x) (}\lambda\texttt{ (x) x))} \\
\big\downarrow \beta \\
\texttt{x[x} \leftarrow \texttt{(}\lambda\texttt{ (x) x)]}
\end{array}
$$

Example

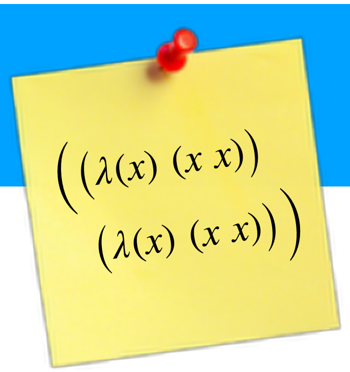



```
((λ (x) x) (λ (x) x))
```

$$\Big\downarrow \beta$$

```
(λ (x) x)
```

Example



Can you beta-reduce the following term 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!

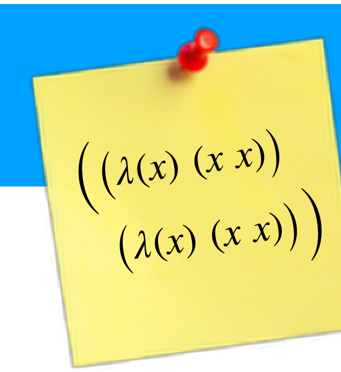
((λ (x) (x x)) (λ (x) (x x)))

β

((λ (x) (x x)) (λ (x) (x x)))

β

Example



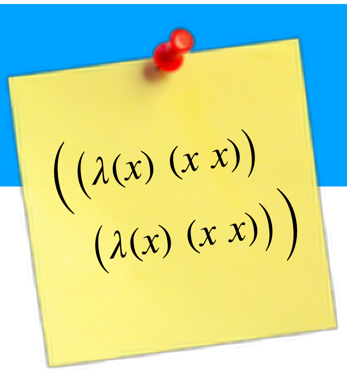
Evaluation with β reduction is nondeterministic!

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

$$\Big\downarrow \beta$$

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

Example



Evaluation with β reduction is nondeterministic!

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

β

or!

β

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

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

Exercise

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?

Exercise

`((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…

Exercise

```
((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.

Exercise

```
((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

We define the free variables of a lambda expression via the function FV:

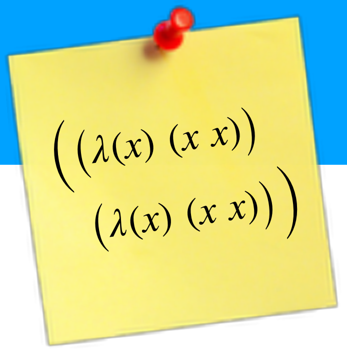
$$\mathbf{FV} : \mathbf{Exp} \rightarrow \mathscr{P}(\mathbf{Var})$$

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

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

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

Example



$$\mathbf{FV}((x\ \ y)) = \{x, y\}$$

$$\mathbf{FV}(((\lambda\ \ (x)\ \ x)\ \ y)) = \{y\}$$

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

$$\mathbf{FV}(((\lambda\ \ (y)\ \ ((\lambda\ \ (x)\ \ (z\ \ x))\ \ x)))) = \{z, x\}$$

Exercise

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)
```

Exercise

What are the free variables of each of the following terms?

`((λ (x) x) y)`

{y}

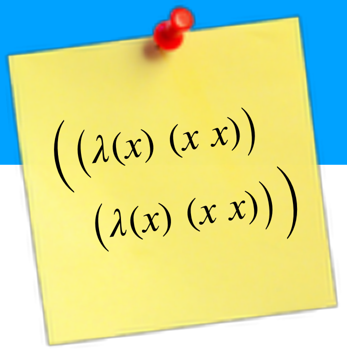
`((λ (x) (x x)) (λ (x) (x x)))`

{}

`((λ (x) (z y)) x)`

{x, y, z}

Example

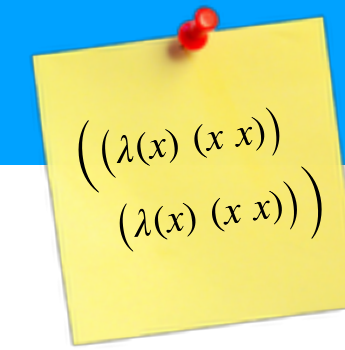


The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

Example



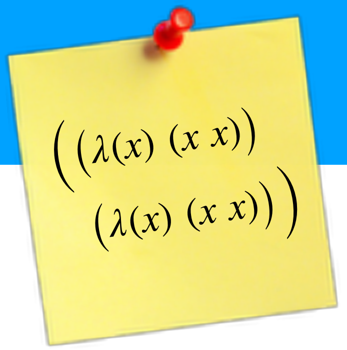
The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) a)[a ← (λ (b) b)]
```

Example



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

$$E_0[x \leftarrow E_1]$$

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

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

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

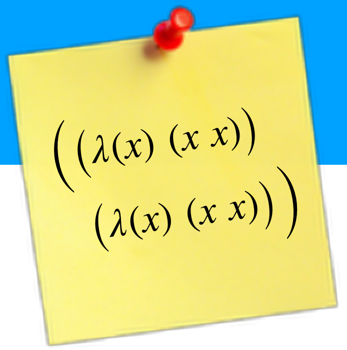
$$(\lambda \;\; (x) \;\; E_0)[x \leftarrow E] = (\lambda \;\; (x) \;\; E_0)$$

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

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

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

Example



Capture-avoiding substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\downarrow \beta$$

```
(λ (a) a)
```

Exercise

How can you beta-reduce the following expression using capture-avoiding substitution?

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

Exercise

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))
```

Exercise

How can you beta-reduce the following expression using capture-avoiding substitution?

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

Exercise

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)

Exercise

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

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

$$=_\alpha$$

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

$$\texttt{(}\lambda\ \texttt{(x) (E}_0\ \texttt{x))} \rightarrow_\eta \texttt{E}_0 \text{ where } x \notin FV(\texttt{E}_0)$$

η - expansion

$E_0 \quad \rightarrow_\eta \quad (\lambda\ (x)\ (E_0\ x))$ where $x \notin FV(E_0)$

Reduction

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

$$(\rightarrow^*)$$

reflexive/transitive closure

Evaluation

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

Evaluation to *normal form*

$$\begin{array}{c} \texttt{E}_0 \\ \big\downarrow {\scriptstyle *} \\ \texttt{(λ (x) …)} \end{array}$$

Evaluation to *normal form*

$$\mathtt{E}_0 \xrightarrow{*} \mathtt{(\lambda\ (x)\ \ldots\ (\lambda\ (z)\ ((a\ \ldots)\ \ldots)))}$$

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))
```

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

$\rightarrow_\beta$ `(λ (z) z)`

Evaluation Strategy

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

```
→β ((λ (y) y) (λ (z) z))
```

```
→β (λ (z) z)
```

Evaluation Strategy

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

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

$\rightarrow_\beta$ `(λ (z) z)`

Confluence

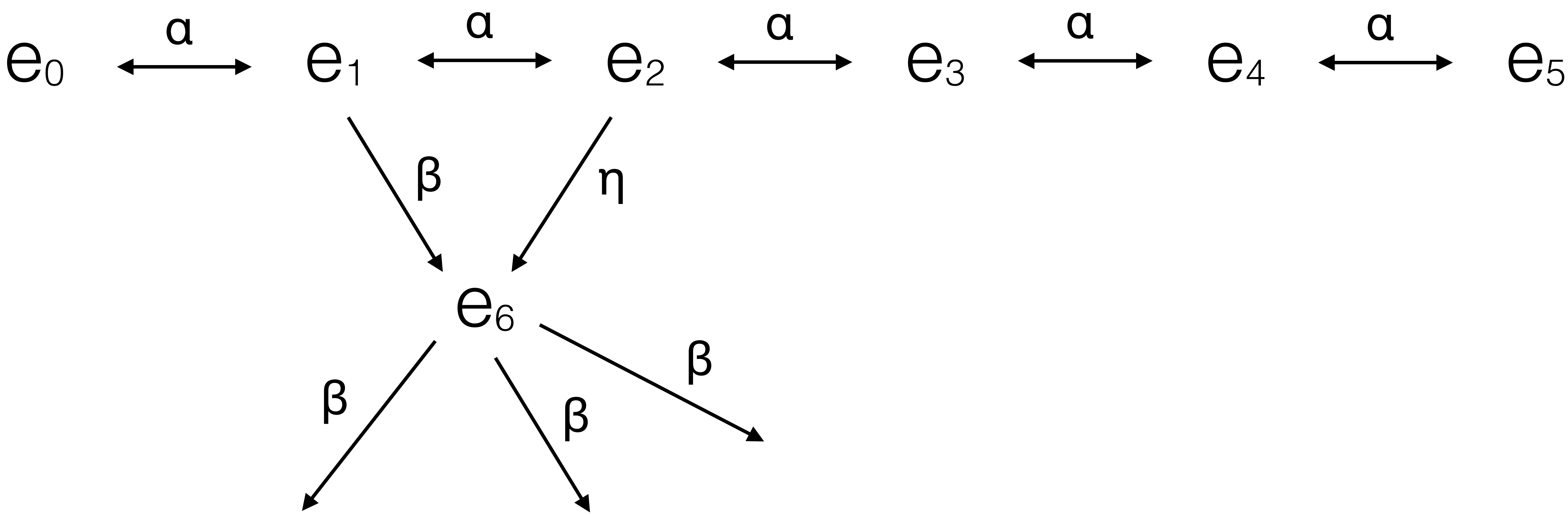
Diverging paths of evaluation must eventually join back together.

$$\begin{tikzcd}
 & E_0 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
E_1 \arrow[dr, "*"'] & & E_2 \arrow[dl, "*"] \\
 & E_3 &
\end{tikzcd}$$

Church-Rosser Theorem

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

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


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

$$
\begin{tikzcd}
e_0 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_1 \arrow[rr, leftrightarrow, "\alpha"] & & e_2 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_3 \arrow[rr, leftrightarrow, "\alpha"] & & e_4 \arrow[rr, leftrightarrow, "\alpha"] & & e_5 \\
& e_7 \arrow[dr, "\beta"] & & & & e_{10} \arrow[dr, "\beta"] & & & & & \\
& & e_8 \arrow[dr, "\beta"] & & & & e_{11} & & & & \\
& & & e_9 & & & & & & &
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
e_0 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_1 \arrow[rr, leftrightarrow, "\alpha"] & & e_2 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_3 \arrow[rr, leftrightarrow, "\alpha"] & & e_4 \arrow[rr, leftrightarrow, "\alpha"] & & e_5 \\
& e_7 \arrow[dr, "\beta"] & & & & e_{10} \arrow[dr, "\beta"] & & & & & \\
& & e_8 \arrow[dr, "\beta"] & & & & e_{11} \arrow[ddr, "*"] & & & & \\
& & & e_9 \arrow[drrrr, "*"'] & & & & & & & \\
& & & & & & & e_{12} & & &
\end{tikzcd}
$$

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*.

Exercise

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

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

Exercise

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))))
```