λ calculus,
reduction strategies ,
and abstract machines

CS245 — Fall 2019

Authors: Kris Micinski + Thomas Gilray

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

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

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

```
((λ (x) x) (λ (x) x))
```

$$\Big\downarrow \beta$$

```
x[x ← (λ (x) x)]
```

Free variables

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

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

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

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

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)))
```

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

$\to_\beta$ `(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

$$(\lambda\ (x)\ E_0) \quad \rightarrow_\alpha \quad (\lambda\ (y)\ E_0[x \leftarrow y])$$

$$=_\alpha$$

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

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

η - expansion

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

Earlier…

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))
```

If we wanted to define *perform-β* (a function that performs a single β-reduction), with this type, what would the problem be?

$$\textbf{perform-}\beta : e \rightarrow e$$

Hint: Consider how it would execute on…

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

If we wanted to define *perform-β* (a function that performs a single β-reduction), with this type, what would the problem be?

$$\textbf{perform-}\beta : e \rightarrow e$$

Answer: possibly more than one redex! Therefore, multiple possible outputs for single input

$$\textbf{perform-}\beta : e \rightarrow \wp(e)$$

Instead, would go to a **set** of next exprs

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*

$$\begin{tikzcd} E_0 \arrow[d, "*"'] \\ (\lambda\ (x)\ \ldots) \end{tikzcd}$$

Evaluation to *normal form*

$$\begin{array}{c} E_0 \\ {\scriptstyle *}\Big\downarrow \\ \texttt{(λ (x) … (λ (z) ((a …) …)))} \end{array}$$

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))
```

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

$\rightarrow_\beta$ `(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

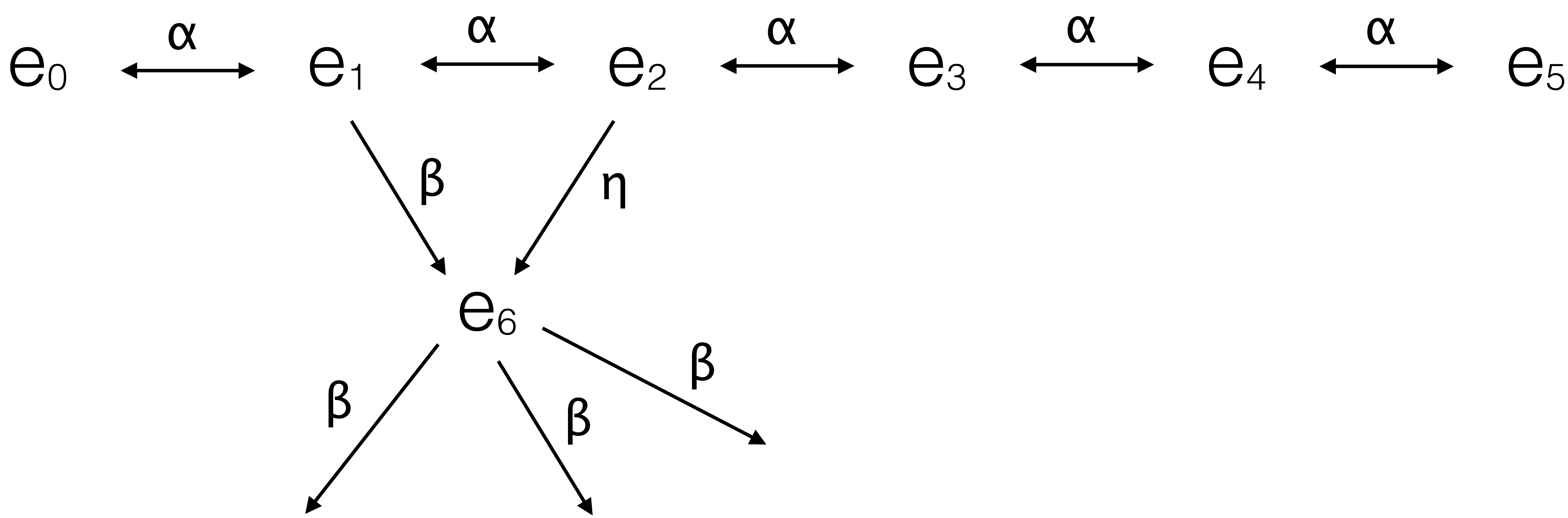
Diverging paths of evaluation must eventually join back together.

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

Church-Rosser Theorem

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

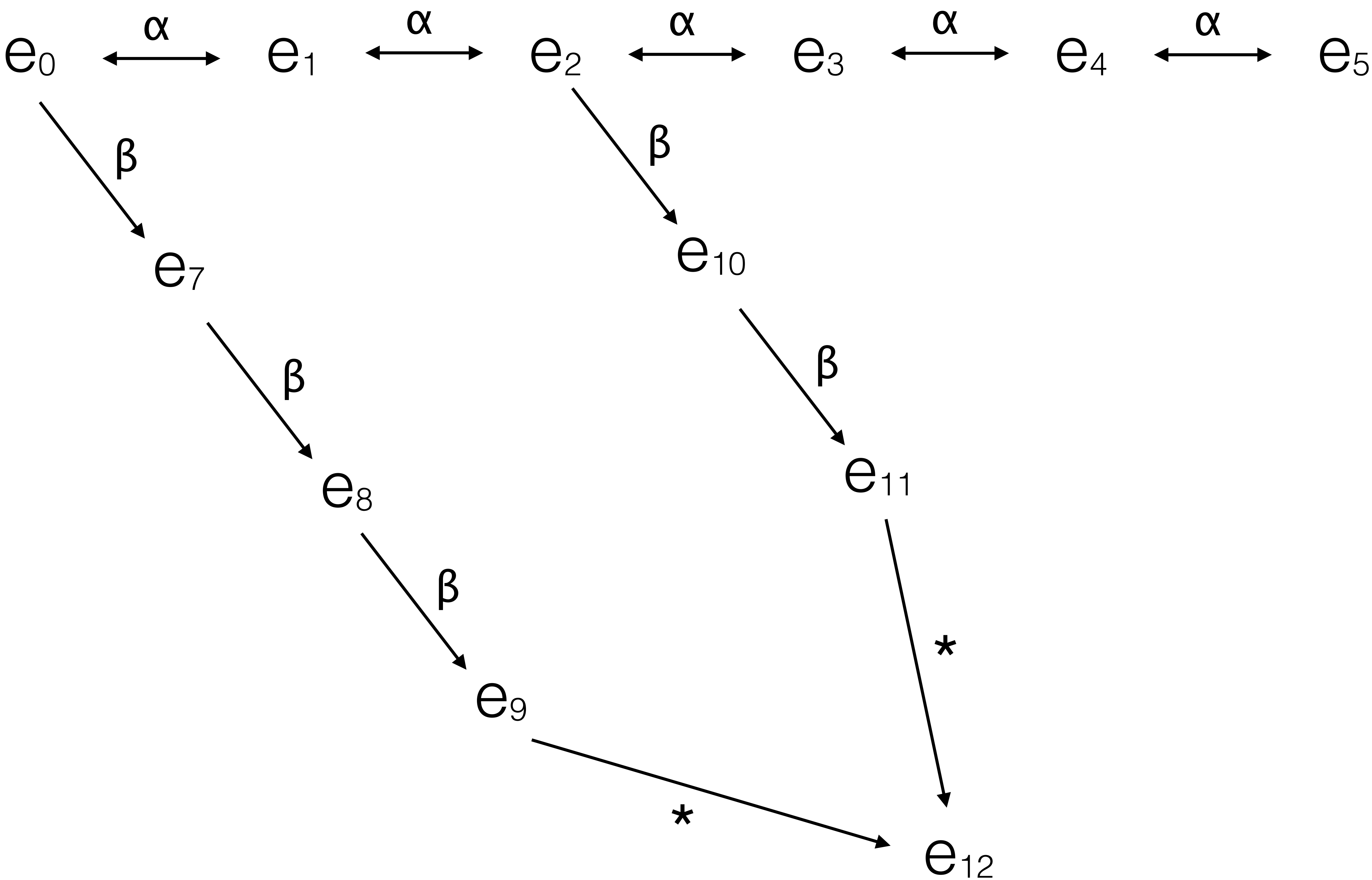
$$
\begin{tikzcd}
e_0 \arrow[rr, leftrightarrow, "\alpha"] & & e_1 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_2 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dl, "\eta"'] & & e_3 \arrow[rr, leftrightarrow, "\alpha"] & & e_4 \arrow[rr, leftrightarrow, "\alpha"] & & e_5 \\
& & & e_6 &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



e_0
α
e_1
α
e_2
α
e_3
α
e_4
α
e_5
β
η
e_6
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)



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*.

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))))
```

Abstract Machines

- Imaginary computer (“machine”) meant to interpret a language
- Have some combination of features (instruction, stack, store/heap, etc…)
- We define **state space** (“configurations”) mathematically
- Also define **transitions** or “steps” between configurations
- Also need to know how to **inject** a program into the state space
  - I.e., how do we come up with an initial state?
- Machines typically named based on their components (C, CC, CK, CEK, CESK, CKS, etc…)

CBV C Machine

Textual reduction semantics

$$\varsigma \in \Sigma = \mathit{Exp}$$

State space of machine is just exprs

$$\rightsquigarrow : \Sigma \to \Sigma$$

Need to define step fn

$$\mathit{inj} : \mathit{Exp} \to \Sigma$$

Need to define injection fn

Necessary components for abstract machine

For the “C Machine”, states will just be expressions (syntax), and injection (*inj*) will just be the identity function!

$$\rightsquigarrow : \Sigma \to \Sigma$$

We define **values** as lambda expressions

$$v \in \mathit{Val} = \mathit{Lam}$$

Rules

$$\big((\lambda\,(x)\ e_0)\ v\big) \rightsquigarrow e_0[x \mapsto v]$$

$$e_0 \rightsquigarrow e' \implies (e_0\ e_1) \rightsquigarrow (e'\ e_1)$$

$$e_1 \rightsquigarrow e' \implies (v\ e_1) \rightsquigarrow (v\ e')$$

Actually writing the code…

```
; Values (irreducible expressions) are just λs
(define (value? e)
  (match e
    [`(λ (,x) ,body) #t]
    [else #f]))
```

```
; Free variables (need this for capture-avoiding substitution)
(define (free-vars e)
  (match e
    [`(λ (,x) ,body) (set-subtract (free-vars body) (set x))]
    [(? symbol? x) (set x)]
    [`(,e0 ,e1) (set-union (free-vars e0) (free-vars e1))]))
```

```
; Return a fresh symbol that lies outside of the set s.
(define (fresh-var s)
  (let ([x (gensym)])
    (if (not (set-member? s x)) x
        (fresh-var s))))
```

```
; Perform capture-avoiding substitution from x to e1 within e.
; α-conversion as necessary to avoid captures. This uses the
; fresh-var function above to perform α-renaming
(define (capture-avoiding-subst e x e1)
  (match e
    [`(λ (,y) ,body)
     (if (equal? x y)
         ; Need to convert y to be something else
         ;
         ; Note that we use capture-avoiding-subst twice here: the
         ; innermost call is to perform the necessary α conversion.
         (let ([z (fresh-var (free-vars body))])
           `(λ (,z) ,(capture-avoiding-subst
                          (capture-avoiding-subst body y z)
                          x
                          e1)))
         `(λ (,y) ,(capture-avoiding-subst body x e1)))]
     [(? symbol? y) (if (equal? x y) e1 y)]
     [`(,e2 ,e3) `(,(capture-avoiding-subst e2 x e1)
                   ,(capture-avoiding-subst e3 x e1))]))
```

The step function itself is actually quite simple!

```
; The step function
(define (→ state)
  (match state
    [`((λ (,x) ,body) ,(? value? v))
     (capture-avoiding-subst body x v)]
    [`(,(? value? v) ,e1)
     `(,v ,(→ e1))]
    [`(,e0 ,e1)
     `(,(→ e0) ,e1)]))
```

To “fully evaluate” terms, we take transitive closure of step fn

```
; Evaluate an entire expression (just the transitive closure of →)
(define (eval-c expr)
  (let loop ([e expr])
    (if (value? e) e (loop (→ e)))))
```

Full code available here…

https://github.com/kmicinski/blog-stuff/blob/master/c-machine.rkt

Exercise: modify step fn to do CBN instead!

```
; The step function
(define (→ state)
  (match state
    [`((λ (,x) ,body) ,(? value? v))
      (capture-avoiding-subst body x v)]
    [`(,(? value? v) ,e1)
     `(,v ,(→ e1))]
    [`(,e0 ,e1)
      `(,(→ e0) ,e1)]))
```

Machine components

C

In this case, our state space is just an expression

Machine components

C

In this case, our state space is just an expression

We call this “C” because it represents the “control string” (i.e., the instruction upon which machine operates)

Abstract Machine Zoo

C Term-rewriting Machine

Evaluation contexts

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

$$\mathcal{E} ::= (\mathcal{E}\ \ e) \mid (v\ \ \mathcal{E}) \mid \square$$

(left-to-right) CBV evaluation

$$\mathcal{E} ::= (\mathcal{E}\ \ e) \mid \square$$

(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))
```

$$\mathscr{E} = \texttt{(□ (λ (w) w))}$$

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

Context and redex

$\mathscr{E}$[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))

                          ↓ 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))))
```

Abstract Machine Zoo

C	Term-rewriting Machine
CC	Context and Redex Machine
CK	Control / Continuation Machine

Next time…