Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Untyped λ-Calculus

- How do you compute with the λ-calculus?
- Roughly speaking, by substitution:
- (λ x. e_1) e_2 ⇒* e_1 [e_2 / x]
- * Modulo all kinds of subtleties to avoid free variable capture

Transition Semantics for λ-Calculus

$$\frac{E \text{ –> } E''}{E\ E' \text{ --> } E''\ E'}$$

- Application (version 1 - Lazy Evaluation)

$$(\lambda\ x\ .\ E)\ E' \text{ --> } E[E'/x]$$

- Application (version 2 - Eager Evaluation)

$$\frac{E' \text{ --> } E''}{(\lambda\ x\ .\ E)\ E' \text{ --> } (\lambda\ x\ .\ E)\ E''}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \text{ --> } E[V/x]}$$

V - variable or abstraction (value)

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is Turing Complete
  - Can express any sequential computation
- Problems:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar

Typed vs Untyped λ-Calculus

- The *pure* λ-calculus has no notion of type: (f f) is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms
- Simply typed λ-calculus is less powerful than the untyped λ-Calculus: NOT Turing Complete (no general recursion)

α Conversion

- α-conversion:

  λ x. exp --α--> λ y. (exp [y/x])
- Provided that
  1. y is not free in exp
  2. No free occurrence of x in exp becomes bound in exp when replaced by y

λ x. x (λ y. x y) -X-> λ y. y(λ y.y y)

α Conversion Non-Examples

1. Error: y is not free in termsecond

$$\lambda x.\ x\ y\ \cancel{\text{--}\alpha\text{-->}}\ \lambda y.\ y\ y$$

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

$$\lambda x.\ \underbrace{\lambda y.\ x\ y}_{\text{exp}}\ \cancel{\text{--}\alpha\text{-->}}\ \lambda y.\ \underbrace{\lambda y.\ y\ y}_{\text{exp[y/x]}}$$

But $\lambda x.\ (\lambda y.\ y)\ x$ --α--> $\lambda y.\ (\lambda y.\ y)\ y$

And $\lambda y.\ (\lambda y.\ y)\ y$ --α--> $\lambda x.\ (\lambda y.\ y)\ x$

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if:

1. It is an equivalence relation
2. If $e_1 \sim e_2$ then
   - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
   - $\lambda x.\ e_1 \sim \lambda x.\ e_2$

α Equivalence

- α equivalence is the smallest congruence containing α conversion

- One usually treats α-equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: $\lambda x.\ (\lambda y.\ y\ x)\ x \sim\alpha\sim \lambda y.\ (\lambda x.\ x\ y)\ y$

- $\lambda x.\ (\lambda y.\ y\ x)\ x$ --α--> $\lambda z.\ (\lambda y.\ y\ z)\ z$ so
  $\lambda x.\ (\lambda y.\ y\ x)\ x \sim\alpha\sim \lambda z.\ (\lambda y.\ y\ z)\ z$
- $(\lambda y.\ y\ z)$ --α--> $(\lambda x.\ x\ z)$ so
  $(\lambda y.\ y\ z) \sim\alpha\sim (\lambda x.\ x\ z)$ so
  $\lambda z.\ (\lambda y.\ y\ z)\ z \sim\alpha\sim \lambda z.\ (\lambda x.\ x\ z)\ z$
- $\lambda z.\ (\lambda x.\ x\ z)\ z$ --α--> $\lambda y.\ (\lambda x.\ x\ y)\ y$ so
  $\lambda z.\ (\lambda x.\ x\ z)\ z \sim\alpha\sim \lambda y.\ (\lambda x.\ x\ y)\ y$
- $\lambda x.\ (\lambda y.\ y\ x)\ x \sim\alpha\sim \lambda y.\ (\lambda x.\ x\ y)\ y$

Substitution

- Defined on α-equivalence classes of terms
- P [N / x] means replace every free occurrence of x in P by N
  - P called *redex*; N called *residue*
- Provided that no variable free in P becomes bound in P [N / x]
  - Rename bound variables in P to avoid capturing free variables of N

Substitution

- $x\ [N / x] = N$
- $y\ [N / x] = y$ if $y \neq x$
- $(e_1\ e_2)\ [N / x] = ((e_1 [N / x])\ (e_2 [N / x]))$
- $(\lambda x.\ e)\ [N / x] = (\lambda x.\ e)$
- $(\lambda y.\ e)\ [N / x] = \lambda y.\ (e\ [N / x])$
  provided $y \neq x$ and y not free in N
  - Rename y in redex if necessary

Example

(λ y. y z) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] --α--> (λ w.w z) [(λ x. x y) / z] = λ w. w (λ x. x y)

Example

- Only replace free occurrences
- (λ y. y z (λ z. z)) [(λ x. x) / z] =
  λ y. y (λ x. x) (λ z. z)

Not

λ y. y (λ x. x) (λ z. (λ x. x))

β reduction

- β Rule: (λ x. P) N --β--> P [N /x]
- Essence of computation in the lambda calculus
- Usually defined on α-equivalence classes of terms

Example

- (λ z. (λ x. x y) z) (λ y. y z)
  --β--> (λ x. x y) (λ y. y z)
  --β--> (λ y. y z) y --β--> y z
- (λ x. x x) (λ x. x x)
  --β--> (λ x. x x) (λ x. x x)
  --β--> (λ x. x x) (λ x. x x) --β--> ….

α β Equivalence

- α β equivalence is the smallest congruence containing α equivalence and β reduction
- A term is in *normal form* if no subterm is α equivalent to a term that can be β reduced
- Hard fact (Church-Rosser): if e_1 and e_2 are $\alpha\beta$-equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

- Not all terms reduce to normal forms
- Not all reduction strategies will produce a normal form if one exists

Lazy evaluation:

- Always reduce the left-most application in a top-most series of applications (i.e. Do not perform reduction inside an abstraction)
- Stop when term is not an application, or left-most application is not an application of an abstraction to a term

Example 1

- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))
- Lazy evaluation:
- Reduce the left-most application:
- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y)) --β--> (λ x. x)

Eager evaluation

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then β-reduce the application

Example 1

- (λ z. (λ x. x))((λ y. y y) (λ y. y y))
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
- (λ z. (λ x. x))((λ y. y y) (λ y. y y))

--β--> (λ z. (λ x. x))((λ y. y y) (λ y. y y))

--β--> (λ z. (λ x. x))((λ y. y y) (λ y. y y))…

Example 2

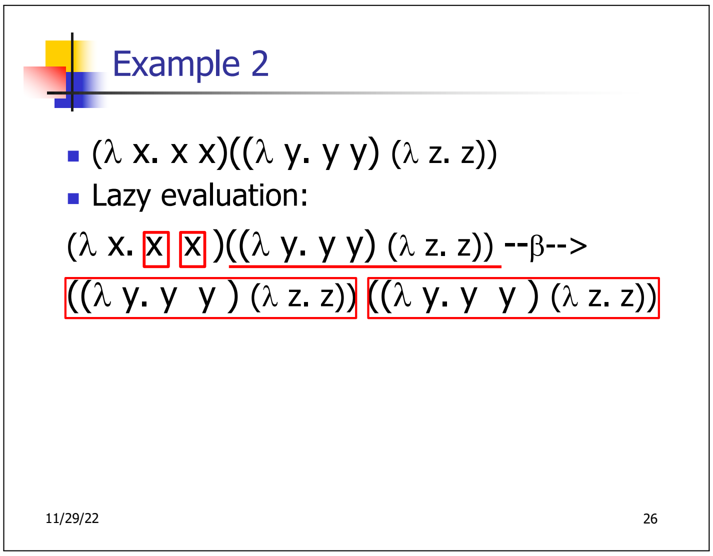
- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. $\boxed{x}$ $\boxed{x}$)$\underline{((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))}$ --β-->

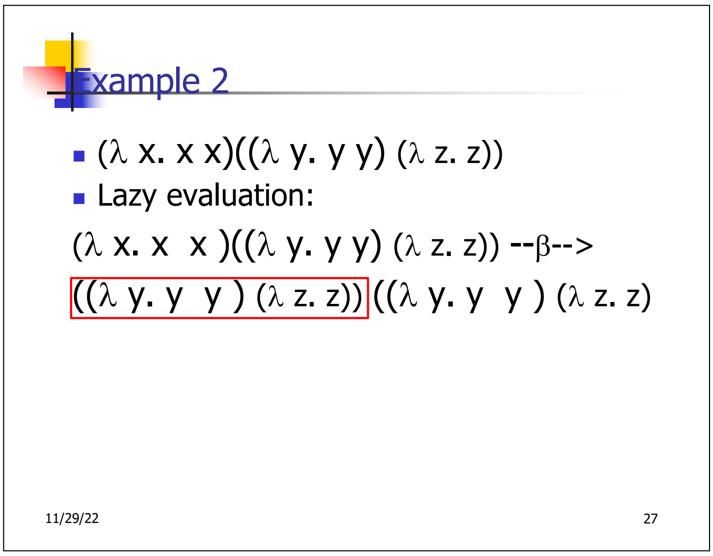


Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. $\boxed{x}$ $\boxed{x}$)($\underline{(\lambda y.\ y\ y)\ (\lambda z.\ z))}$ --β-->

$\boxed{((\lambda y.\ y\ \ y\ )\ (\lambda z.\ z))}$ $\boxed{((\lambda y.\ y\ \ y\ )\ (\lambda z.\ z))}$

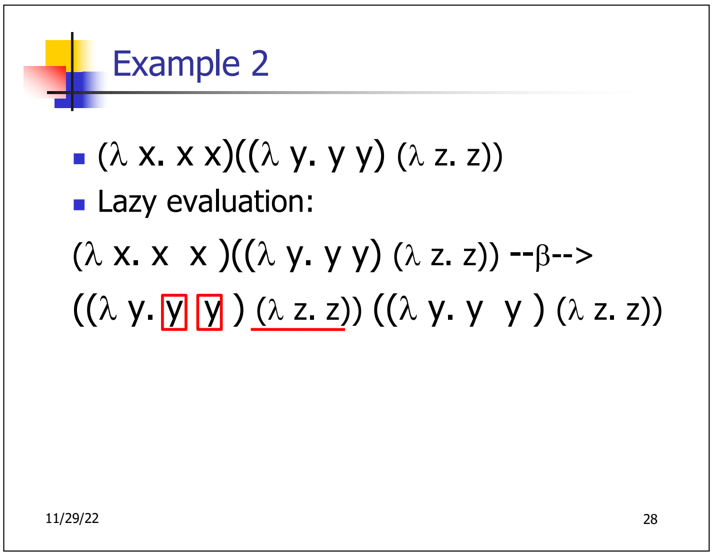


Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

$\boxed{((\lambda y.\ y\ \ y\ )\ (\lambda z.\ z))}$ ((λ y. y y) (λ z. z)

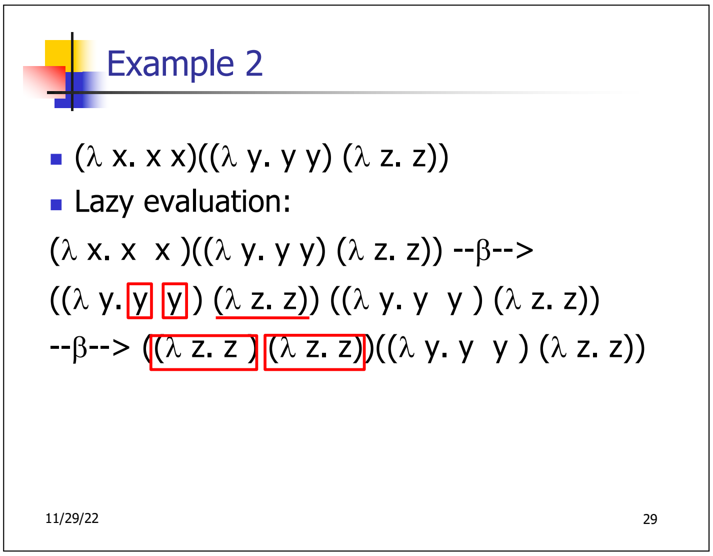


Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. $\boxed{y}$ $\boxed{y}$) $\underline{(\lambda z.\ z)}$) ((λ y. y y) (λ z. z))



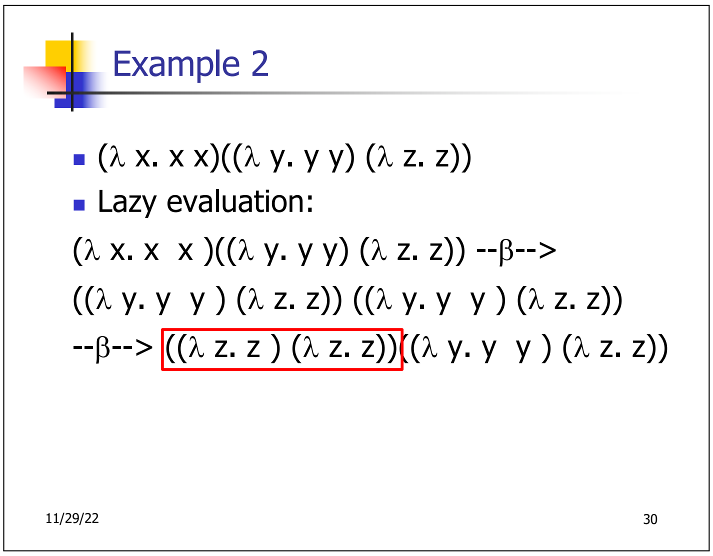
Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. $\boxed{y}$ $\boxed{y}$) $\underline{(\lambda z.\ z)}$) ((λ y. y y) (λ z. z))

--β--> ($\boxed{(\lambda z.\ z\ )}$ $\boxed{(\lambda z.\ z)}$)((λ y. y y) (λ z. z))



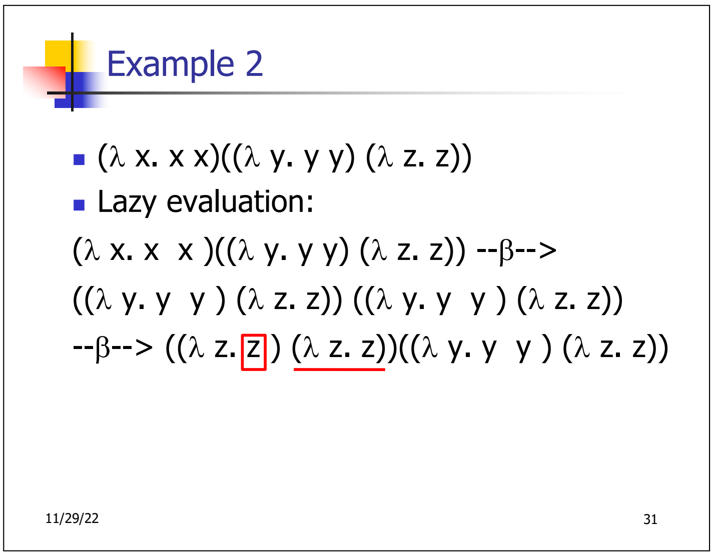
Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> $\boxed{((\lambda z.\ z\ )\ (\lambda z.\ z))}$((λ y. y y) (λ z. z))



Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. $\boxed{z}$) $\underline{(\lambda z.\ z)}$)((λ y. y y) (λ z. z))

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z))

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z) ~β~ λ z. z

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Eager evaluation:

(λ x. x x) ((λ y. y y) (λ z. z)) --β-->

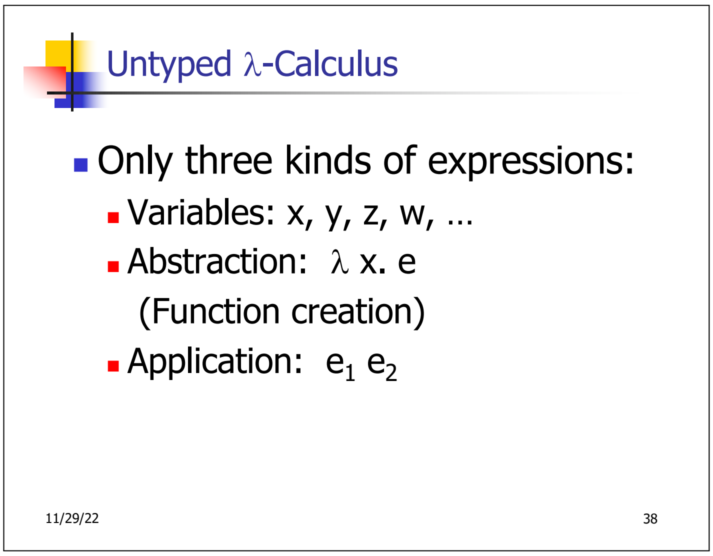
(λ x. x x) ((λ z. z) (λ z. z)) --β-->

(λ x. x x) (λ z. z) --β-->

(λ z. z) (λ z. z) --β--> λ z. z

η (Eta) Reduction

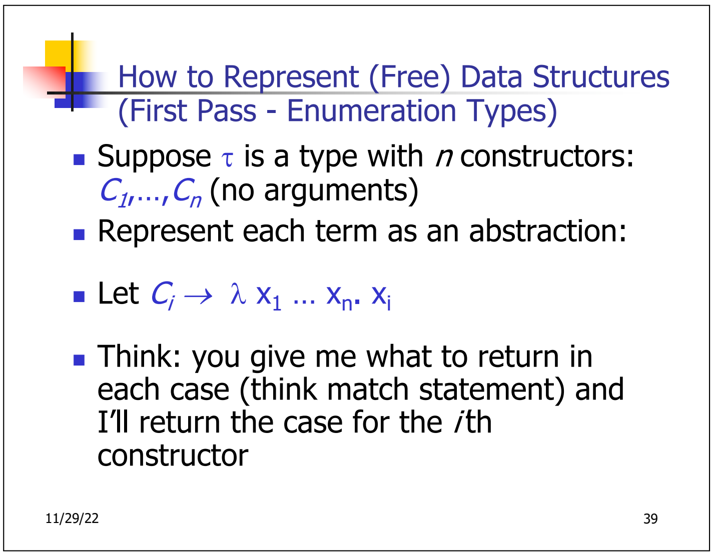
- η Rule: λ x. f x --η--> f if x not free in f
  - Can be useful in each direction
  - Not valid in Ocaml
    - recall lambda-lifting and side effects
  - Not equivalent to (λ x. f) x --> f (inst of β)
- Example: λ x. (λ y. y) x --η--> λ y. y



Untyped λ-Calculus

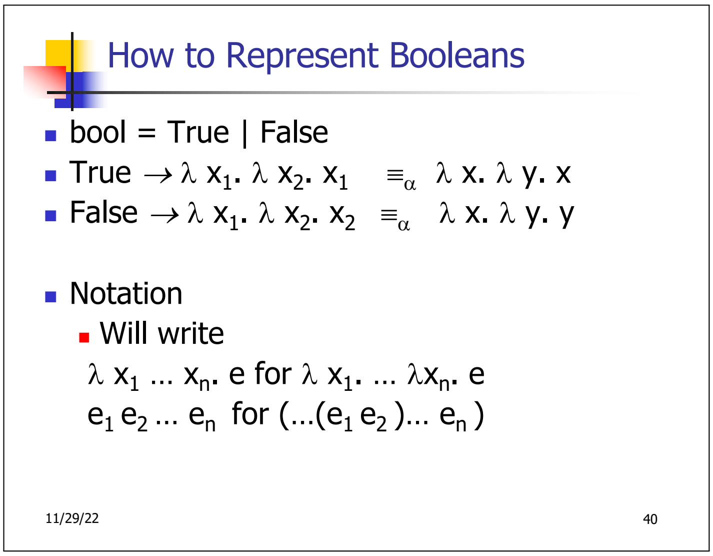
- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation)
  - Application: $e_1\ e_2$



How to Represent (Free) Data Structures (First Pass - Enumeration Types)

- Suppose τ is a type with *n* constructors: $C_1,\ldots,C_n$ (no arguments)
- Represent each term as an abstraction:
- Let $C_i \rightarrow \lambda\ x_1 \ldots x_n.\ x_i$
- Think: you give me what to return in each case (think match statement) and I'll return the case for the *i*th constructor



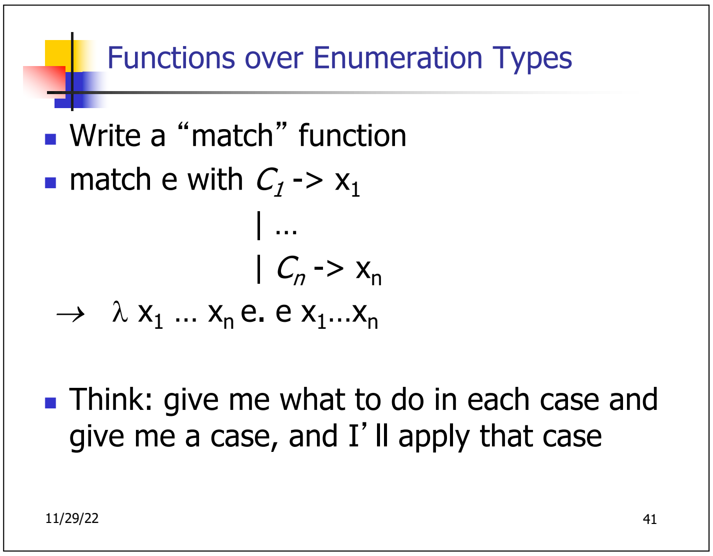
How to Represent Booleans

- bool = True | False
- True $\rightarrow \lambda\ x_1.\ \lambda\ x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\ x.\ \lambda\ y.\ x$
- False $\rightarrow \lambda\ x_1.\ \lambda\ x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\ x.\ \lambda\ y.\ y$

- Notation
  - Will write

  $\lambda\ x_1 \ldots x_n.\ e$ for $\lambda\ x_1.\ \ldots\ \lambda x_n.\ e$

  $e_1\ e_2 \ldots e_n$ for $(\ldots(e_1\ e_2)\ldots\ e_n)$



Functions over Enumeration Types

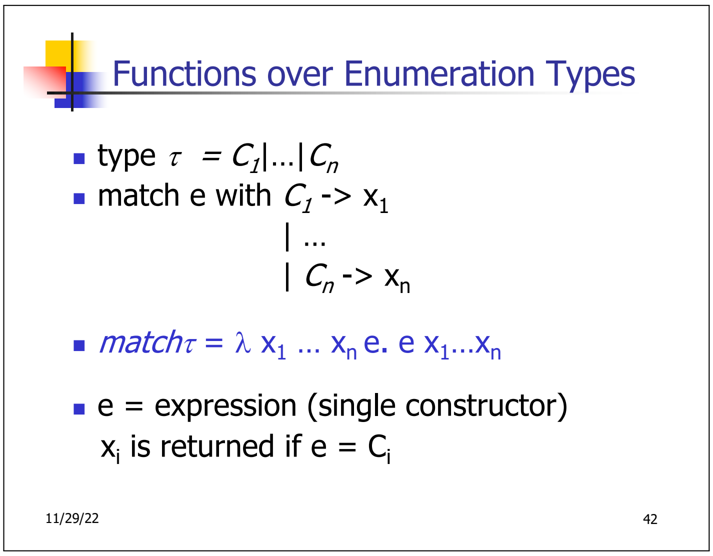
- Write a "match" function
- match e with C_1 -> x_1

  | …

  | C_n -> x_n

$\rightarrow \quad \lambda\ x_1 \ldots x_n\ e.\ e\ x_1 \ldots x_n$

- Think: give me what to do in each case and give me a case, and I' ll apply that case



Functions over Enumeration Types

- type τ = $C_1|\ldots|C_n$
- match e with C_1 -> x_1

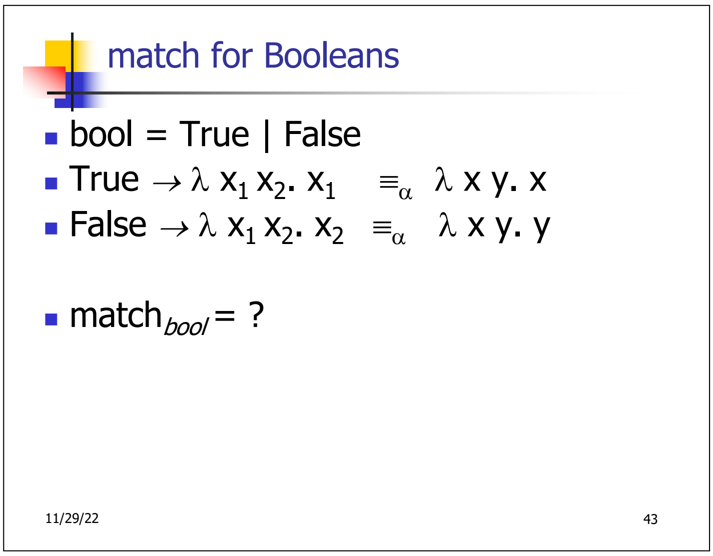
  | …

  | C_n -> x_n

- $match\tau = \lambda\ x_1 \ldots x_n\ e.\ e\ x_1 \ldots x_n$

- e = expression (single constructor)

  x_i is returned if e = C_i



match for Booleans

- bool = True | False
- True $\rightarrow \lambda\ x_1\ x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\ x\ y.\ x$
- False $\rightarrow \lambda\ x_1\ x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\ x\ y.\ y$

- $match_{bool}$ = ?

match for Booleans

- bool = True | False
- True $\rightarrow \lambda\, x_1\, x_2.\, x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\, x$
- False $\rightarrow \lambda\, x_1\, x_2.\, x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\, y$

- $\text{match}_{\mathit{bool}} = \lambda\, x_1\, x_2\, e.\ e\, x_1\, x_2$
  $\equiv_\alpha \quad \lambda\, x\, y\, b.\ b\, x\, y$

How to Write Functions over Booleans

- if b then x_1 else $x_2 \rightarrow$
- if_then_else b $x_1\, x_2$ = b $x_1\, x_2$
- if_then_else $\equiv \lambda\, b\, x_1\, x_2\, .\ b\, x_1\, x_2$

How to Write Functions over Booleans

- Alternately:
- if b then x_1 else x_2 =
  match b with True -> x_1 | False -> x_2 $\rightarrow$
  $\text{match}_{\text{bool}}\, x_1\, x_2\, b$ =
  $(\lambda\, x_1\, x_2\, b\, .\ b\, x_1\, x_2\,)\, x_1\, x_2\, b = b\, x_1\, x_2$
- if_then_else
  $\equiv \lambda\, b\, x_1\, x_2.\ (\text{match}_{\text{bool}}\, x_1\, x_2\, b)$
  $= \lambda\, b\, x_1\, x_2.\ (\lambda\, x_1\, x_2\, b\, .\ b\, x_1\, x_2\,)\, x_1\, x_2\, b$
  $= \lambda\, b\, x_1\, x_2.\ b\, x_1\, x_2$

Example:

not b

= match b with True -> False | False -> True

$\rightarrow (\text{match}_{\text{bool}})$ False True b

$= (\lambda\, x_1\, x_2\, b\, .\ b\, x_1\, x_2\,)\ (\lambda\, x\, y.\, y)\ (\lambda\, x\, y.\, x)\ b$

$= b\ (\lambda\, x\, y.\, y)(\lambda\, x\, y.\, x)$

- not $\equiv \lambda\, b.\ b\ (\lambda\, x\, y.\, y)(\lambda\, x\, y.\, x)$
- Try and, or

and or

How to Represent (Free) Data Structures (Second Pass - Union Types)

- Suppose τ is a type with *n* constructors:
  type $\tau = C_1\, t_{11} \dots t_{1k}\, | \dots |C_n\, t_{n1} \dots t_{nm,}$
- Represent each term as an abstraction:
- $C_i\, t_{i1} \dots t_{ij,} \rightarrow \lambda\, x_1 \dots x_n.\ x_i\, t_{i1} \dots t_{ij,}$
- $C_i \rightarrow \lambda\, t_{i1} \dots t_{ij,}\, x_1 \dots x_n\, .\ x_i\, t_{i1} \dots t_{ij,}$
- Think: you need to give each constructor its arguments fisrt

How to Represent Pairs

- Pair has one constructor (comma) that takes two arguments
- type (α,β)pair = (,) α β
- (a , b) --> λ x . x a b
- (_ , _) --> λ a b x . x a b

Functions over Union Types

- Write a “match” function
- match e with C_1 $y_1 \ldots y_{m1}$ -> f_1 $y_1 \ldots y_{m1}$
  | …
  | C_n $y_1 \ldots y_{mn}$ -> f_n $y_1 \ldots y_{mn}$
- *match*$\tau \rightarrow$ λ $f_1 \ldots f_n$ e. e $f_1 \ldots f_n$
- Think: give me a function for each case and give me a case, and I’ll apply that case to the appropriate fucntion with the data in that case

Functions over Pairs

- $\text{match}_{\text{pair} =}$ λ f p. p f
- fst p = match p with (x,y) -> x
- fst $\rightarrow \lambda$ p. $\text{match}_{\text{pair}}$ (λ x y. x)
  = (λ f p. p f) (λ x y. x) = λ p. p (λ x y. x)
- snd $\rightarrow \lambda$ p. p (λ x y. y)

How to Represent (Free) Data Structures (Third Pass - Recursive Types)

- Suppose τ is a type with *n* constructors:
  type τ = $C_1\, t_{11} \ldots t_{1k}$ | … | $C_n\, t_{n1} \ldots t_{nm,}$
- Suppose $t_{ih} : \tau$ (ie. is recursive)
- In place of a value t_{ih} have a function to compute the recursive value $r_{ih}\, x_1 \ldots x_n$
- $C_i\, t_{i1} \ldots r_{ih} \ldots t_{ij} \rightarrow \lambda\, x_1 \ldots x_n .\ x_i\, t_{i1} \ldots (r_{ih}\, x_1 \ldots x_n) \ldots t_{ij}$
- $C_i \rightarrow \lambda\, t_{i1} \ldots r_{ih} \ldots t_{ij}\, x_1 \ldots x_n . x_i\, t_{i1} \ldots (r_{ih}\, x_1 \ldots x_n) \ldots t_{ij,}$

How to Represent Natural Numbers

- nat = Suc nat | 0
- $\overline{\text{Suc}}$ = λ n f x. f (n f x)
- $\overline{\text{Suc n}}$ = λ f x. f (n f x)
- $\overline{0}$ = λ f x. x
- Such representation called *Church Numerals*

Some Church Numerals

- $\overline{\text{Suc 0}}$ = (λ n f x. f (n f x)) (λ f x. x) -->
  λ f x. f ((λ f x. x) f x) -->
  λ f x. f ((λ x. x) x) --> λ f x. f x

Apply a function to its argument once

Some Church Numerals

- $\overline{\text{Suc(Suc 0)}}$ = (λ n f x. f (n f x)) (Suc 0) -->
(λ n f x. f (n f x)) (λ f x. f x) -->
λ f x. f ((λ f x. f x) f x)) -->
λ f x. f ((λ x. f x) x)) --> λ f x. f (f x)
Apply a function twice

In general $\overline{n}$ = λ f x. f (… (f x)…) with n applications of f

Primitive Recursive Functions

- Write a “fold” function
- fold f_1 … f_n = match e
  with C_1 y_1 … y_{m1} -> f_1 y_1 … y_{m1}
  | …
  | C_i y_1 … r_{ij} …y_{in} -> f_n y_1 … (fold f_1 … f_n r_{ij}) …y_{mn}
  | …
  | C_n y_1 … y_{mn} -> f_n y_1 … y_{mn}
- *$fold_\tau \to$* $\lambda\ f_1 \ldots f_n\ e.\ e\ f_1 \ldots f_n$
- Match in non recursive case a degenerate version of fold

Primitive Recursion over Nat

- fold f z n=
- match n with 0 -> z
- | Suc m -> f (fold f z m)
- $\overline{\text{fold}}$ ≡ λ f z n. n f z
- $\overline{\text{is\_zero}}$ $\overline{n}$ = $\overline{\text{fold}}$ (λ r. $\overline{\text{False}}$) $\overline{\text{True}}$ $\overline{n}$
- = (λ f x. f^{n} x) (λ r. False) True
- = ((λ r. False) ^{n}) True
- ≡ if n = 0 then True else False

Adding Church Numerals

- $\overline{n}$ ≡ λ f x. f^{n} x and m ≡ λ f x. f^{m} x
- $\overline{n + m}$ = λ f x. $f^{(n+m)}$ x
  = λ f x. f^{n} (f^{m} x) = λ f x. $\overline{n}$ f ($\overline{m}$ f x)
- $\overline{+}$ ≡ λ n m f x. n f (m f x)
- Subtraction is harder

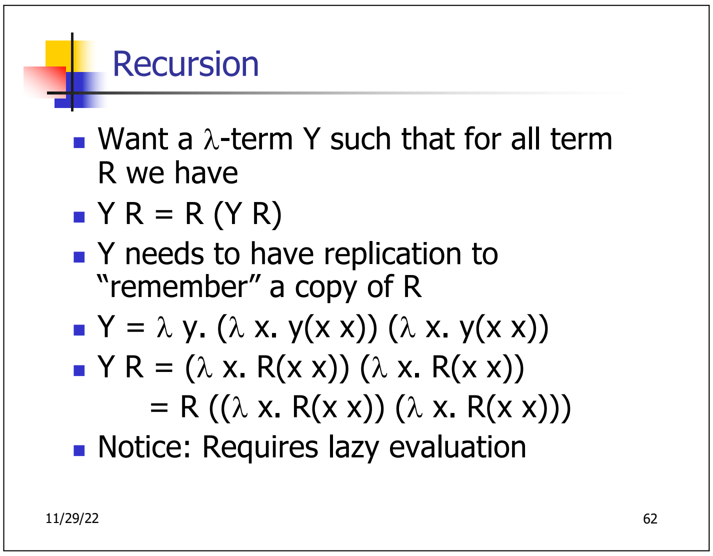
Multiplying Church Numerals

- $\overline{n}$ ≡ λ f x. f^{n} x and m ≡ λ f x. f^{m} x
- $\overline{n * m}$ = λ f x. $(f^{n * m})$ x = λ f x. $(f^{m})^{n}$ x
  = λ f x. $\overline{n}$ ($\overline{m}$ f) x

$\overline{*}$ ≡ λ n m f x. n (m f) x

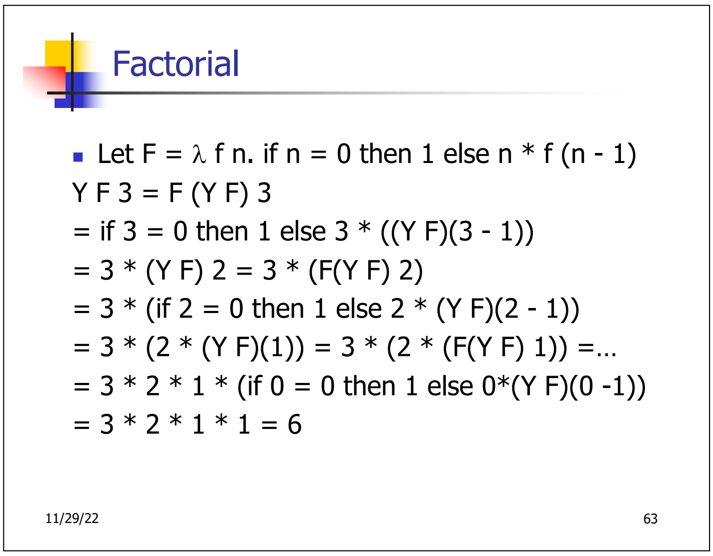
Predecessor

- let pred_aux n =
  match n with 0 -> (0,0)
  | Suc m
  -> (Suc(fst(pred_aux m)), fst(pred_aux m)
  = fold (λ r. (Suc(fst r), fst r)) (0,0) n
- pred ≡ λ n. snd (pred_aux n) n =
  λ n. snd (fold (λ r.(Suc(fst r), fst r)) (0,0) n)



Recursion

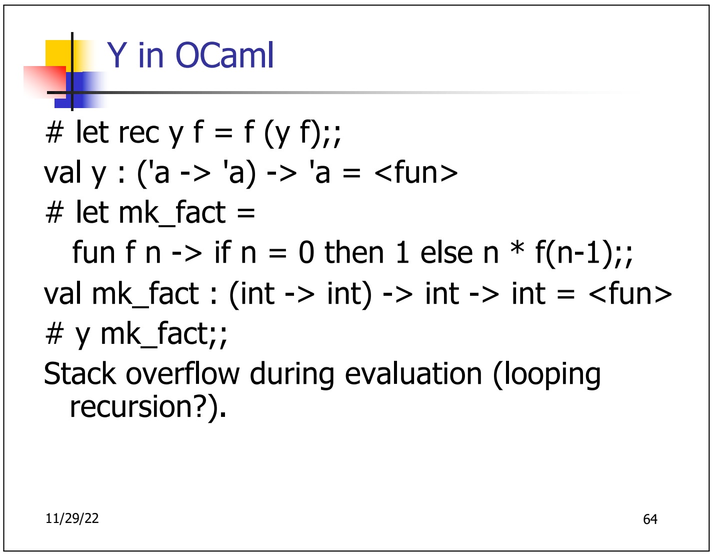
- Want a λ-term Y such that for all term R we have
- Y R = R (Y R)
- Y needs to have replication to "remember" a copy of R
- Y = λ y. (λ x. y(x x)) (λ x. y(x x))
- Y R = (λ x. R(x x)) (λ x. R(x x))
  = R ((λ x. R(x x)) (λ x. R(x x)))
- Notice: Requires lazy evaluation



Factorial

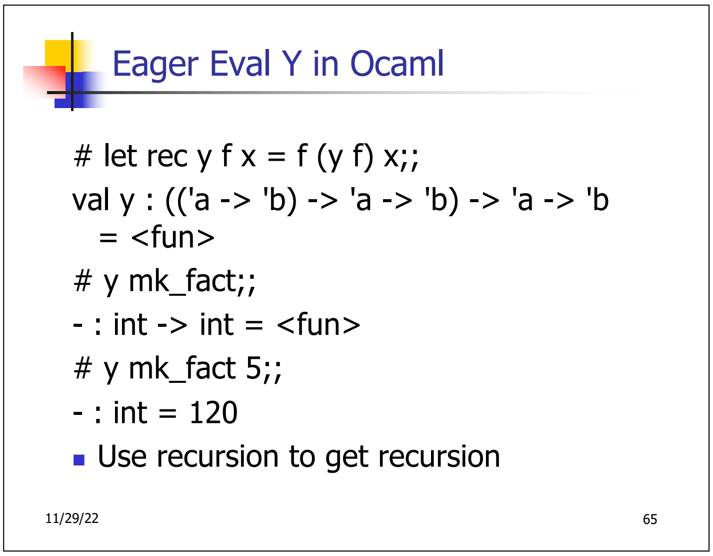
- Let F = λ f n. if n = 0 then 1 else n * f (n - 1)

Y F 3 = F (Y F) 3
= if 3 = 0 then 1 else 3 * ((Y F)(3 - 1))
= 3 * (Y F) 2 = 3 * (F(Y F) 2)
= 3 * (if 2 = 0 then 1 else 2 * (Y F)(2 - 1))
= 3 * (2 * (Y F)(1)) = 3 * (2 * (F(Y F) 1)) =…
= 3 * 2 * 1 * (if 0 = 0 then 1 else 0*(Y F)(0 -1))
= 3 * 2 * 1 * 1 = 6



Y in OCaml

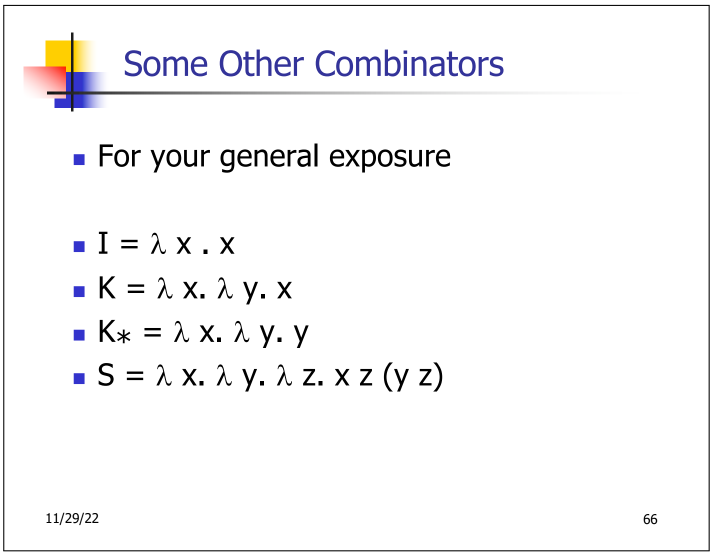
```
# let rec y f = f (y f);;
val y : ('a -> 'a) -> 'a = <fun>
# let mk_fact =
  fun f n -> if n = 0 then 1 else n * f(n-1);;
val mk_fact : (int -> int) -> int -> int = <fun>
# y mk_fact;;
Stack overflow during evaluation (looping
  recursion?).
```



Eager Eval Y in Ocaml

```
# let rec y f x = f (y f) x;;
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
   = <fun>
# y mk_fact;;
- : int -> int = <fun>
# y mk_fact 5;;
- : int = 120
```

- Use recursion to get recursion



Some Other Combinators

- For your general exposure
- I = λ x . x
- K = λ x. λ y. x
- K_* = λ x. λ y. y
- S = λ x. λ y. λ z. x z (y z)