

Programming Languages and Compilers (CS 421)

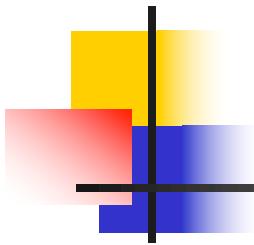


Elsa L Gunter

2112 SC, UIUC

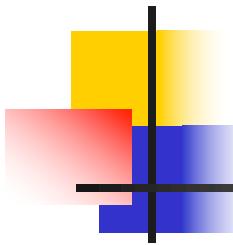
<http://courses.engr.illinois.edu/cs421>

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:
$$(\lambda x. e_1) e_2 \Rightarrow^* e_1 [e_2 / x]$$
- * Modulo all kinds of subtleties to avoid free variable capture



Transition Semantics for λ -Calculus

$$\frac{E \rightarrow E''}{EE' \rightarrow E''E'}$$

- Application (version 1 - Lazy Evaluation)

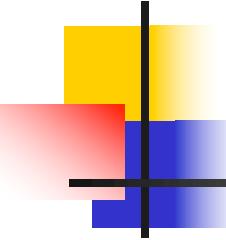
$$(\lambda x. E) E' \rightarrow E[E'/x]$$

- Application (version 2 - Eager Evaluation)

$$\frac{E' \rightarrow E''}{(\lambda x. E) E' \rightarrow (\lambda x. E) E''}$$

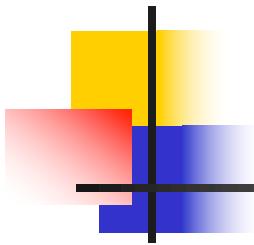
$$\overline{(\lambda x. E) V \rightarrow E[V/x]}$$

V - variable or abstraction (value)



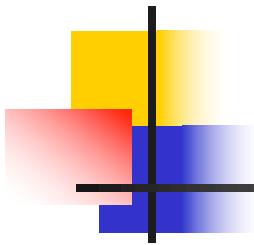
How Powerful is the Untyped λ -Calculus?

- 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

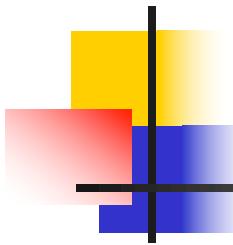
- α -conversion:

$$\lambda x. \exp \dashv\alpha\dashrightarrow \lambda y. (\exp [y/x])$$

- 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

$$\lambda x. x (\lambda y. x y) \text{ -X-} \rightarrow \lambda y. y(\lambda y. y y)$$



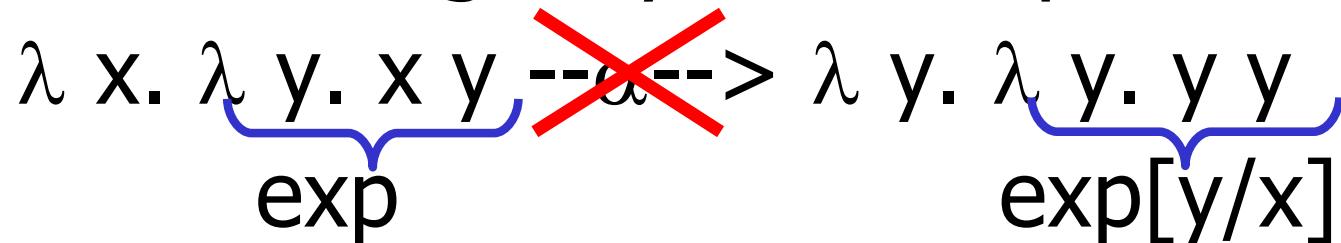
α Conversion Non-Examples

1. Error: y is not free in term second

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

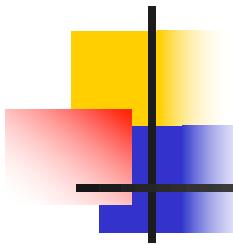
2. Error: free occurrence of x becomes bound in wrong way when replaced by y

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



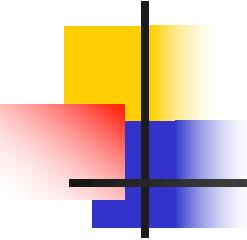
But $\lambda x. (\lambda y. y) x \text{ --a-->} \lambda y. (\lambda y. y) y$

And $\lambda y. (\lambda y. y) y \text{ --a-->} \lambda x. (\lambda y. y) x$



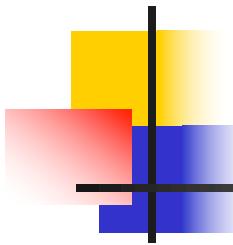
Congruence

- Let \sim be a relation on lambda terms. \sim 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

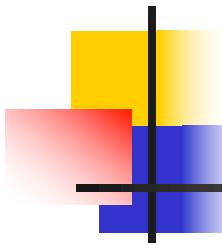
- α equivalence is the smallest congruence containing α conversion
- One usually treats α -equivalent terms as equal - i.e. use α equivalence classes of terms



Example

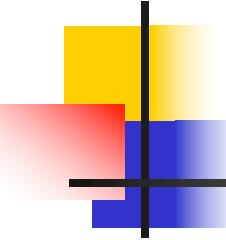
Show: $\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda y. (\lambda x. x y) y$

- $\lambda x. (\lambda y. y x) x \rightarrow_{\alpha} \lambda z. (\lambda y. y z) z$ so
 $\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda z. (\lambda y. y z) z$
- $(\lambda y. y z) \rightarrow_{\alpha} (\lambda x. x z)$ so
 $(\lambda y. y z) \sim_{\alpha} (\lambda x. x z)$ so
 $\lambda z. (\lambda y. y z) z \sim_{\alpha} \lambda z. (\lambda x. x z) z$
- $\lambda z. (\lambda x. x z) z \rightarrow_{\alpha} \lambda y. (\lambda x. x y) y$ so
 $\lambda z. (\lambda x. x z) z \sim_{\alpha} \lambda y. (\lambda x. x y) y$
- $\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda y. (\lambda x. x y) y$



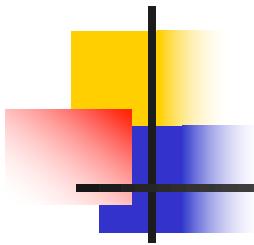
Substitution

- 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

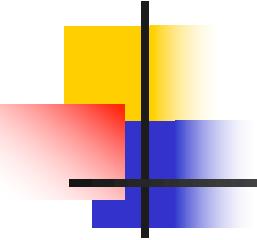
- $x [N / x] = N$
- $y [N / x] = y$ if $y \neq x$
- $(e_1 e_2) [N / x] = ((e_1 [N / x]) (e_2 [N / x]))$
- $(\lambda x. e) [N / x] = (\lambda x. e)$
- $(\lambda y. e) [N / x] = \lambda y. (e [N / x])$
provided $y \neq x$ and y not free in N
 - Rename y in redex if necessary



Example

$$(\lambda y. y z) [(\lambda x. x y) / z] = ?$$

- Problems?
 - z in redex in scope of y binding
 - y free in the residue
- $(\lambda y. y z) [(\lambda x. x y) / z] \text{ --}\alpha\text{--} >$
 $(\lambda w. w z) [(\lambda x. x y) / z] =$
 $\lambda w. w (\lambda x. x y)$

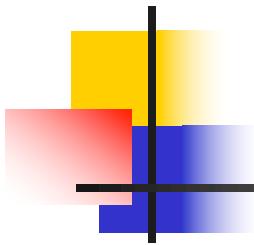


Example

- Only replace free occurrences
- $(\lambda y. y z (\lambda z. z)) [(\lambda x. x) / z] =$
 $\lambda y. y (\lambda x. x) (\lambda z. z)$

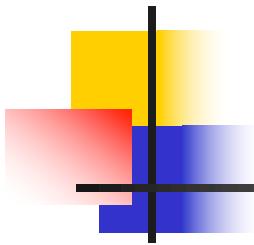
Not

$$\lambda y. y (\lambda x. x) (\lambda z. (\lambda x. x))$$



β reduction

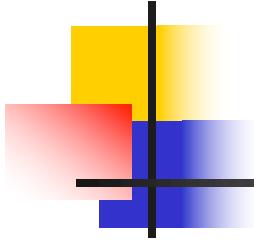
- β Rule: $(\lambda x. P) N \rightarrow \beta P [N/x]$
- Essence of computation in the lambda calculus
- Usually defined on α -equivalence classes of terms



Example

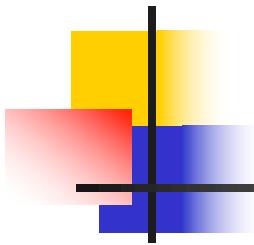
- $(\lambda z. (\lambda x. x y) z) (\lambda y. y z)$
-- β --> $(\lambda x. x y) (\lambda y. y z)$
-- β --> $(\lambda y. y z) y$ -- β --> $y z$

- $(\lambda x. x x) (\lambda x. x x)$
-- β --> $(\lambda x. x x) (\lambda x. x x)$
-- β --> $(\lambda x. x x) (\lambda x. x x)$ -- β -->



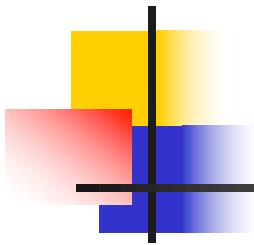
$\alpha \beta$ Equivalence

- $\alpha \beta$ 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



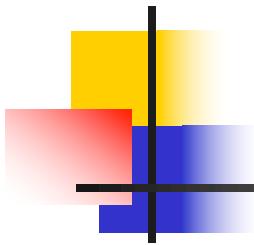
Order of Evaluation

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



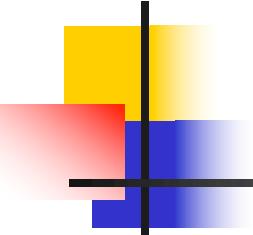
Lazy evaluation:

- 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



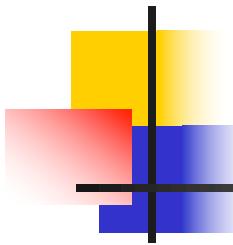
Example 1

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



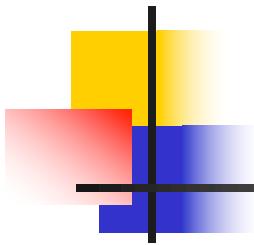
Eager evaluation

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



Example 1

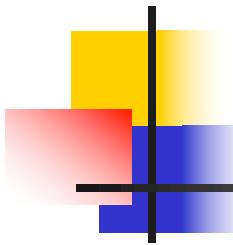
- $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
 - $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
 - β --> $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
 - β --> $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y)) \dots$



Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

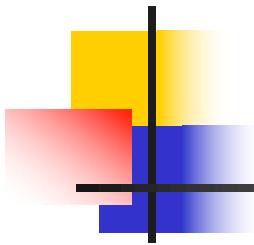


Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$

- Lazy evaluation:

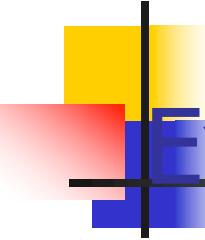
$(\lambda x. \boxed{x} \boxed{x})((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$



Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$$(\lambda x. \boxed{x} \boxed{x}) \underline{((\lambda y. y y) (\lambda z. z))} \text{--}\beta\text{--}>$$
$$\boxed{((\lambda y. y y) (\lambda z. z))} \quad \boxed{((\lambda y. y y) (\lambda z. z))}$$

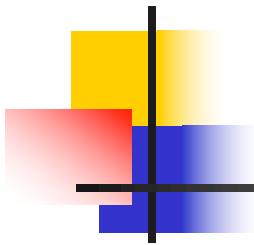


Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

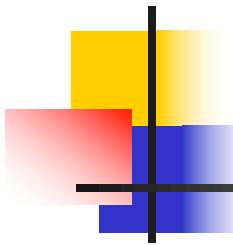
$((\lambda y. y y) (\lambda z. z))((\lambda y. y y) (\lambda z. z))$



Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{ --}\beta\text{-->} ((\lambda y. \boxed{y} \boxed{y}) \underline{(\lambda z. z)}) ((\lambda y. y y) (\lambda z. z))$$



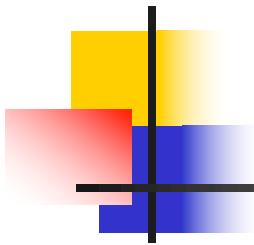
Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

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

$\text{--}\beta\text{--}> (\boxed{(\lambda z. z)} \boxed{(\lambda z. z)}) ((\lambda y. y y) (\lambda z. z))$



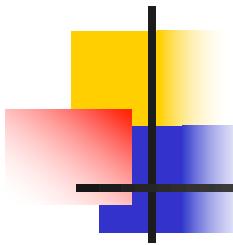
Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}> \boxed{((\lambda z. z) (\lambda z. z))} ((\lambda y. y y) (\lambda z. z))$



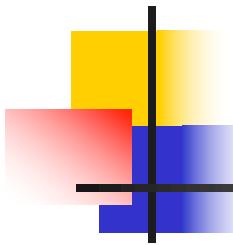
Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}> ((\lambda z. \boxed{z}) \underline{(\lambda z. z)})) ((\lambda y. y y) (\lambda z. z))$



Example 2

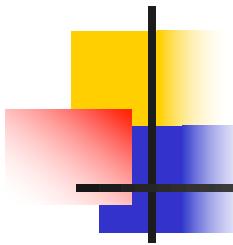
- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}> ((\lambda z. \boxed{z}) \underline{(\lambda z. z)}) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}> \boxed{(\lambda z. z)} ((\lambda y. y y) (\lambda z. z))$



Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$
 $((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$
 $\text{--}\beta\text{--}> ((\lambda z. z) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$
 $\text{--}\beta\text{--}> (\lambda z. \boxed{z}) \underline{((\lambda y. y y) (\lambda z. z))} \text{--}\beta\text{--}>$
 $(\lambda y. y y) (\lambda z. z)$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\xrightarrow{\beta} ((\lambda z. z) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}> (\lambda z. z) ((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$(\lambda y. y y) (\lambda z. z)$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\xrightarrow{\beta} ((\lambda z. z) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}> (\lambda z. z) ((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}>$

$(\lambda y. y y) (\lambda z. z) \sim \beta \sim \lambda z. z$

Example 2

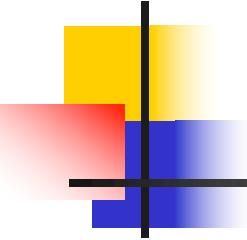
- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Eager evaluation:

$(\lambda x. x x) \boxed{((\lambda y. y y) (\lambda z. z))} \text{--}\beta\text{--}>$

$(\lambda x. x x) \boxed{((\lambda z. z) (\lambda z. z))} \text{--}\beta\text{--}>$

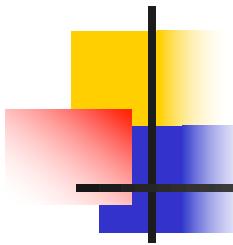
$(\lambda x. x x) \boxed{(\lambda z. z)} \text{--}\beta\text{--}>$

$(\lambda z. z) (\lambda z. z) \text{--}\beta\text{--}> \lambda z. z$



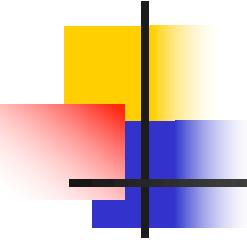
η (Eta) Reduction

- η Rule: $\lambda x. f x \rightsquigarrow f$ 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 $(\lambda x. f) x \rightarrow f$ (inst of β)
- Example: $\lambda x. (\lambda y. y) x \rightsquigarrow \lambda y. y$



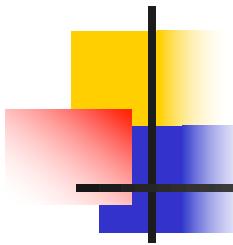
Untyped λ -Calculus

- Only three kinds of expressions:
 - Variables: x, y, z, w, \dots
 - Abstraction: $\lambda x. e$
(Function creation)
 - Application: $e_1 e_2$



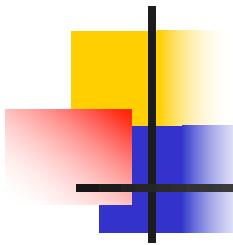
How to Represent (Free) Data Structures (First Pass - Enumeration Types)

- Suppose τ is a type with n constructors:
 C_1, \dots, C_n (no arguments)
- Represent each term as an abstraction:
- Let $C_i \rightarrow \lambda x_1 \dots x_n. x_i$
- 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

- $\text{bool} = \text{True} \mid \text{False}$
- $\text{True} \rightarrow \lambda x_1. \lambda x_2. x_1 \equiv_{\alpha} \lambda x. \lambda y. x$
- $\text{False} \rightarrow \lambda x_1. \lambda x_2. x_2 \equiv_{\alpha} \lambda x. \lambda y. y$
- Notation
 - Will write
$$\lambda x_1 \dots x_n. e \text{ for } \lambda x_1. \dots \lambda x_n. e$$
$$e_1 e_2 \dots e_n \text{ for } (\dots(e_1 e_2) \dots e_n)$$



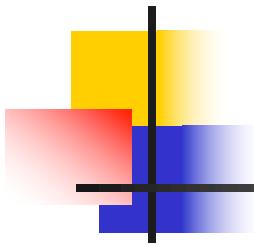
Functions over Enumeration Types

- Write a “match” function
- $\text{match } e \text{ with } C_1 \rightarrow x_1$

| ...
| $C_n \rightarrow x_n$

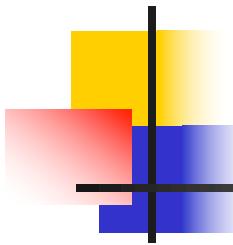
$\rightarrow \lambda x_1 \dots x_n. e. e x_1 \dots x_n$

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



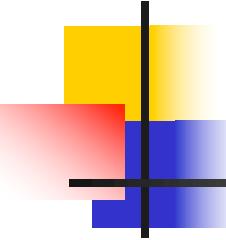
Functions over Enumeration Types

- type $\tau = C_1 \mid \dots \mid C_n$
- match e with $C_1 \rightarrow x_1$
 | ...
 | $C_n \rightarrow x_n$
- $\text{match } \tau = \lambda x_1 \dots x_n e. e \ x_1 \dots x_n$
- e = expression (single constructor)
 x_i is returned if $e = C_i$



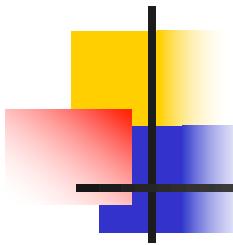
match for Booleans

- $\text{bool} = \text{True} \mid \text{False}$
- $\text{True} \rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x y. x$
- $\text{False} \rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x y. y$
- $\text{match}_{\text{bool}} = ?$



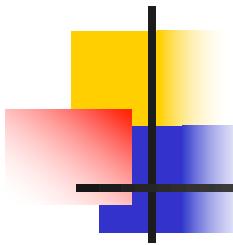
match for Booleans

- $\text{bool} = \text{True} \mid \text{False}$
- $\text{True} \rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x y. x$
- $\text{False} \rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x y. y$
- $\text{match}_{\text{bool}} = \lambda x_1 x_2 e. e x_1 x_2 \equiv_{\alpha} \lambda x y b. b x y$



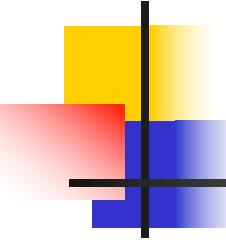
How to Write Functions over Booleans

- $\text{if } b \text{ then } x_1 \text{ else } x_2 \rightarrow$
- $\text{if_then_else } b \ x_1 \ x_2 = b \ x_1 \ x_2$
- $\text{if_then_else} \equiv \lambda \ b \ x_1 \ x_2 \cdot b \ x_1 \ x_2$



How to Write Functions over Booleans

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

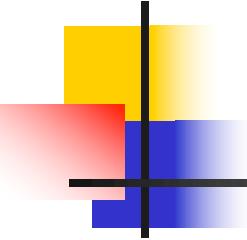


Example:

not b

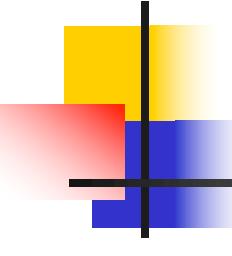
$$\begin{aligned} &= \text{match } b \text{ with True } \rightarrow \text{False} \mid \text{False } \rightarrow \text{True} \\ &\rightarrow (\text{match}_{\text{bool}}) \text{ False True } b \\ &= (\lambda x_1 x_2 b . b x_1 x_2) (\lambda x y. y) (\lambda x y. x) b \\ &= b (\lambda x y. y)(\lambda x y. x) \end{aligned}$$

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



and

or



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

- Suppose τ is a type with n constructors:

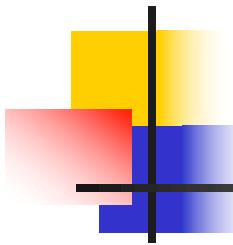
$$\text{type } \tau = C_1 t_{11} \dots t_{1k} \mid \dots \mid C_n t_{n1} \dots t_{nm},$$

- Represent each term as an abstraction:

- $C_i t_{i1} \dots t_{ij} \rightarrow \lambda x_1 \dots x_n. x_i t_{i1} \dots t_{ij},$

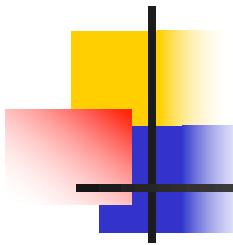
- $C_i \rightarrow \lambda t_{i1} \dots t_{ij}. x_1 \dots x_n. x_i t_{i1} \dots t_{ij},$

- Think: you need to give each constructor its arguments first



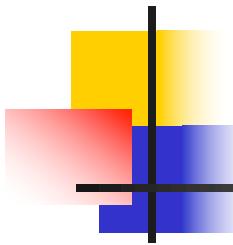
How to Represent Pairs

- Pair has one constructor (comma) that takes two arguments
- type $(\alpha, \beta)\text{pair} = (,) \alpha \beta$
- $(a, b) \rightarrow \lambda x. x a b$
- $(_, _) \rightarrow \lambda a b x. x a b$



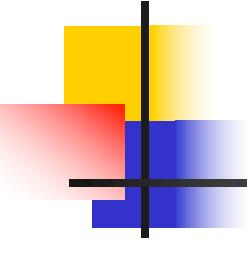
Functions over Union Types

- Write a “match” function
- $\text{match } e \text{ with } C_1 \ y_1 \dots y_{m1} \rightarrow f_1 \ y_1 \dots y_{m1}$
| ...
| $C_n \ y_1 \dots y_{mn} \rightarrow f_n \ y_1 \dots y_{mn}$
- $\text{match } \tau \rightarrow \lambda \ f_1 \dots f_n \ e. \ e \ f_1 \dots f_n$
- Think: give me a function for each case and give me a case, and I'll apply that case to the appropriate function with the data in that case



Functions over Pairs

- $\text{match}_{\text{pair}} = \lambda f p. p f$
- $\text{fst } p = \text{match } p \text{ with } (x,y) \rightarrow x$
- $\begin{aligned} \text{fst} &\rightarrow \lambda p. \text{match}_{\text{pair}} (\lambda x y. x) \\ &= (\lambda f p. p f) (\lambda x y. x) = \lambda p. p (\lambda x y. x) \end{aligned}$
- $\text{snd} \rightarrow \lambda p. p (\lambda x y. y)$

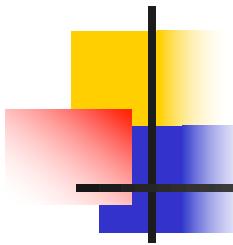


How to Represent (Free) Data Structures (Third Pass - Recursive Types)

- Suppose τ is a type with n constructors:

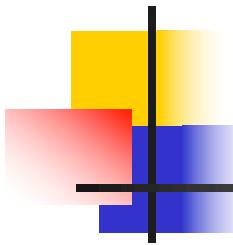
$\text{type } \tau = C_1 t_{11} \dots t_{1k} | \dots | C_n t_{n1} \dots t_{nm},$

- Suppose $t_{ih} : \tau$ (ie. is recursive)
- In place of a value t_{ih} have a function to compute the recursive value $r_{ih} x_1 \dots x_n$
- $C_i t_{i1} \dots r_{ih} \dots t_{ij} \rightarrow \lambda x_1 \dots x_n . x_i \ t_{i1} \dots (r_{ih} x_1 \dots x_n) \dots t_{ij}$
- $C_i \rightarrow \lambda t_{i1} \dots r_{ih} \dots t_{ij} \ x_1 \dots x_n . x_i \ t_{i1} \dots (r_{ih} x_1 \dots x_n) \dots t_{ij},$



How to Represent Natural Numbers

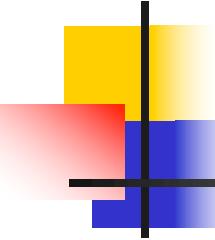
- $\text{nat} = \text{Suc nat} \mid 0$
- $\overline{\text{Suc}} = \lambda n f x. f(n f x)$
- $\overline{\text{Suc}\ n} = \lambda f x. f(n f x)$
- $\overline{0} = \lambda f x. x$
- Such representation called
Church Numerals



Some Church Numerals

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

Apply a function to its argument once

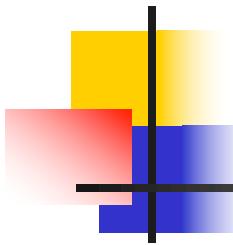


Some Church Numerals

■ $\overline{\text{Suc}(\text{Suc } 0)} = (\lambda n f x. f(n f x)) (\text{Suc } 0) \rightarrow$
 $(\lambda n f x. f(n f x)) (\lambda f x. f x) \rightarrow$
 $\lambda f x. f((\lambda f x. f x) f x) \rightarrow$
 $\lambda f x. f((\lambda x. f x) x) \rightarrow \lambda f x. f(f x)$

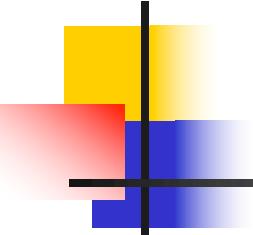
Apply a function twice

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



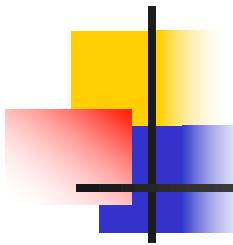
Primitive Recursive Functions

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



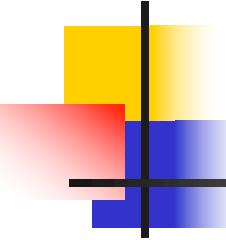
Primitive Recursion over Nat

- $\text{fold } f \ z \ n =$
- $\text{match } n \text{ with } 0 \rightarrow z$
- $\quad \quad \quad | \text{ Suc } m \rightarrow f(\text{fold } f \ z \ m)$
- $\overline{\text{fold}} \equiv \lambda f \ z \ n. \ n \ f \ z$
- $\overline{\overline{\text{is_zero}}} \ - \ \overline{\overline{\text{fold}}} (\lambda r. \text{ False}) \text{ True } n$
- $= (\lambda f x. f^n x) (\lambda r. \text{ False}) \text{ True}$
- $= ((\lambda r. \text{ False})^n) \text{ True}$
- $\equiv \text{if } n = 0 \text{ then True else False}$



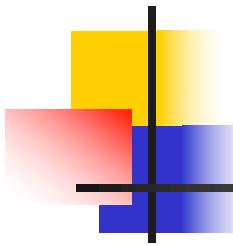
Adding Church Numerals

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



Multiplying Church Numerals

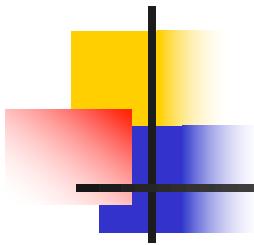
- $\bar{n} \equiv \lambda f x. f^n x$ and $m \equiv \lambda f x. f^m x$
- $\overline{n * m} = \lambda f x. (f^{n * m}) x = \lambda f x. (f^m)^n x$
 $= \lambda f x. \bar{n}(\overline{m} f) x$
- $\bar{*} \equiv \lambda n m f x. n(m f) x$



Predecessor

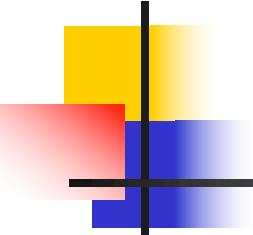
- let pred_aux n =
match n with 0 -> (0,0)
| Suc m
-> (Suc(fst(pred_aux m)), fst(pred_aux m))
= fold ($\lambda r. (\text{Suc}(\text{fst } r), \text{fst } r)$) (0,0) n

- $\text{pred} \equiv \lambda n. \text{snd} (\text{pred_aux } n)$ n =
 $\lambda n. \text{snd} (\text{fold} (\lambda r. (\text{Suc}(\text{fst } r), \text{fst } r)) (0,0) n)$



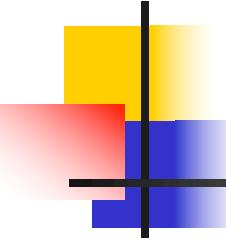
Recursion

- 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
- $Y = \lambda\ y.\ (\lambda\ x.\ y(x\ x))\ (\lambda\ x.\ y(x\ x))$
- $$\begin{aligned} Y\ R &= (\lambda\ x.\ R(x\ x))\ (\lambda\ x.\ R(x\ x)) \\ &= R\ ((\lambda\ x.\ R(x\ x))\ (\lambda\ x.\ R(x\ x))) \end{aligned}$$
- Notice: Requires lazy evaluation



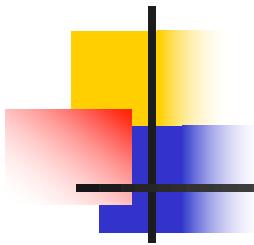
Factorial

- Let $F = \lambda f n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1)$
- $$\begin{aligned}Y F 3 &= F(Y F) 3 \\&= \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * ((Y F)(3 - 1)) \\&= 3 * (Y F) 2 = 3 * (F(Y F) 2) \\&= 3 * (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * (Y F)(2 - 1)) \\&= 3 * (2 * (Y F)(1)) = 3 * (2 * (F(Y F) 1)) = \dots \\&= 3 * 2 * 1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (Y F)(0 - 1)) \\&= 3 * 2 * 1 * 1 = 6\end{aligned}$$



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?).
```



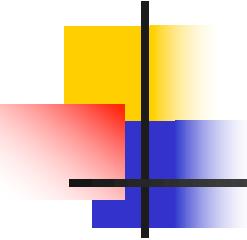
Eager Eval Y in Ocaml

```
# 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

- $I = \lambda x . x$
- $K = \lambda x. \lambda y. x$
- $K_* = \lambda x. \lambda y. y$
- $S = \lambda x. \lambda y. \lambda z. x z (y z)$