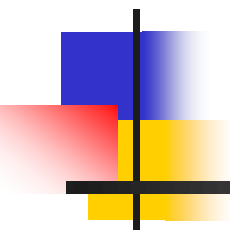


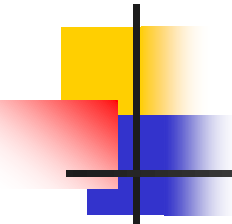
Programming Languages and Compilers (CS 421)



William Mansky

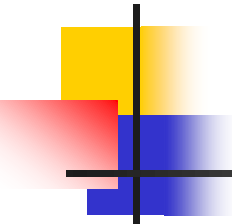
<http://courses.engr.illinois.edu/cs421/>

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



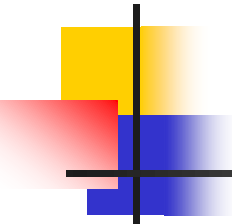
Untyped λ -Calculus

- Only three kinds of expressions:
 - Variables: $x, y, z, w, \dots$
 - Abstraction: $\lambda x. e$
 - Application: $e_1 e_2$
- What if we want something more complicated? bools? ints? ADTs?



Representing Data Structures (Enums)

- Suppose τ is an enumeration type with n constructors: $C_1, \dots, C_n$ (no arguments)
- Represent each term as an abstraction:
$$C_i \rightarrow \lambda x_1 \dots x_n. x_i$$
- Idea: take as arguments what to return in each case (as in match statement), and return the case for the i 'th constructor
- Functional (“extensional”) description of the enumeration

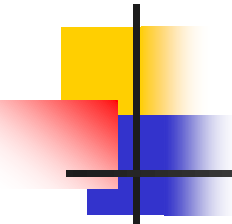


How to Represent Booleans

- `bool = True | False`
- $\text{True} \rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x. \lambda y. x$
- $\text{False} \rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x. \lambda y. y$

- Notes:

$\lambda x_1 \dots x_n. e$ means $\lambda x_1. \dots \lambda x_n. e$
application is left-associative



Functions over Enumeration Types

- Write a “match” function

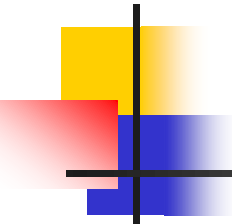
- match e with $C_1 \rightarrow x_1$

| ...

| $C_n \rightarrow x_n$

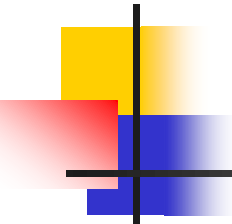
$\rightarrow \lambda x_1 \dots x_n e. e x_1 \dots x_n$

- Idea: take as arguments what to do in each case and a case, and return the result of applying that case



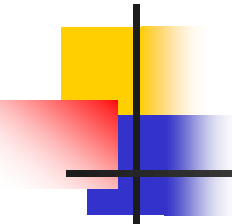
Functions over Enumeration Types

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



match for Booleans

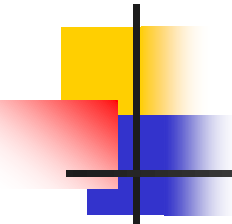
- $\text{bool} = \text{True} \mid \text{False}$
- $\text{True} \rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x y. x$
- $\text{False} \rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x y. y$
- $\text{match}_{\text{bool}} = ?$



match for Booleans

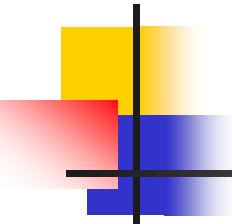
- $\text{bool} = \text{True} \mid \text{False}$
- $\text{True} \rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x y. x$
- $\text{False} \rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x y. y$

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



How to Write Functions over Booleans

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



Encoded Boolean Function

not b

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

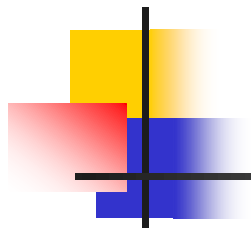
→ (match_{bool}) False True b

= (λ x₁ x₂ b . b x₁ x₂) (λ x y. y) (λ x y. x) b

= b (λ x y. y) (λ x y. x) ("b False True")

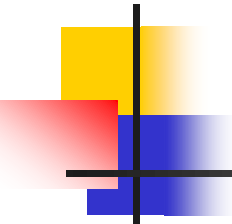
■ not ≡ λ b. b (λ x y. y) (λ x y. x)

■ How about and, or?



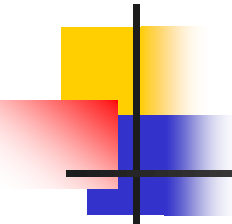
and

or



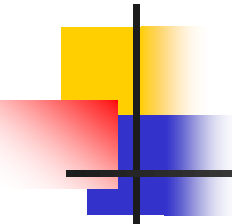
Representing Data Structures (Unions)

- Suppose τ is a type with n constructors:
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:
- $C_i t_{i1} \dots t_{ij} \rightarrow \lambda x_1 \dots x_n. x_i t_{i1} \dots t_{ij}$
- $C_i \rightarrow \lambda t_{i1} \dots t_{ij} x_1 \dots x_n. x_i t_{i1} \dots t_{ij}$
- Idea: you need to give each constructor its arguments first; then it acts like an enumeration



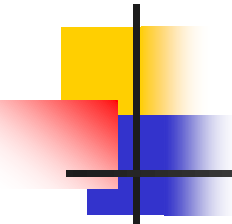
How to Represent Pairs

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



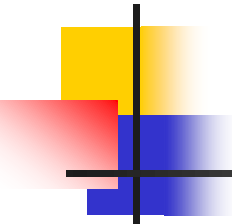
Functions over Union Types

- Write a “match” function
- match e with $C_1 y_1 \dots y_{m1} \rightarrow f_1 y_1 \dots y_{m1}$
| $\dots$
| $C_n y_1 \dots y_{mn} \rightarrow f_n y_1 \dots y_{mn}$
- $\text{match}_\tau \rightarrow \lambda f_1 \dots f_n e. e f_1 \dots f_n$
- Idea: take as arguments a function for each case and a case, and return the result of applying that case, i.e., applying the appropriate function to the data of the case



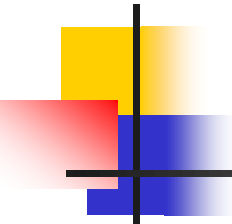
Functions over Pairs

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



Functions over Pairs

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



Functions over Pairs

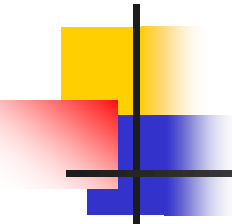
- $\text{match}_{\text{pair}} = \lambda f p. p f$
- $\text{fst } p = \text{match } p \text{ with } (x,y) \rightarrow x$
- $\text{fst} \rightarrow \lambda p. \text{match}_{\text{pair}} (\lambda x y. x)$
 $= (\lambda f p. p f) (\lambda x y. x) = \lambda p. p (\lambda x y. x)$
- $\text{snd} \rightarrow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

Representing Data Structures (Recursion)

- Suppose τ is a type with n constructors:

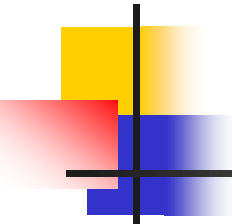
type $\tau = C_1 t_{11} \dots t_{1k} \mid \dots \mid C_n t_{n1} \dots t_{nm}$

- Suppose $t_{ih} = \tau$ (i.e. 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

- $\text{nat} \rightarrow \text{Suc nat} \mid 0$
 - $\text{Suc} \rightarrow \lambda n f x. f (n f x)$
 - $\text{Suc } n \rightarrow \lambda f x. f (n f x)$
 - $0 \rightarrow \lambda f x. x$
-
- This representation is called
Church Numerals



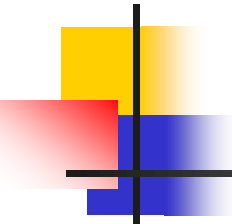
Some Church Numerals

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

$$\xrightarrow{\beta} \lambda f x. f ((\lambda f x. x) f x)$$

$$\xrightarrow{\beta} \lambda f x. f ((\lambda x. x) x) \xrightarrow{\beta} \lambda f x. f x$$

Apply a function to its argument once



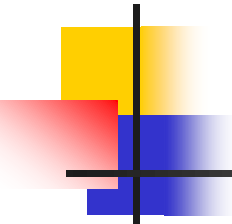
Some Church Numerals

$$\begin{aligned} \blacksquare \text{ Suc (Suc 0)} &\rightarrow (\lambda n f x. f (n f x)) (\text{Suc 0}) \\ &= (\lambda n f x. f (n f x)) (\lambda f x. f x) \\ &\xrightarrow{\beta} \lambda f x. f ((\lambda f x. f x) f x) \\ &\xrightarrow{\beta} \lambda f x. f ((\lambda x. f x) x) \\ &\xrightarrow{\beta} \lambda f x. f (f x) \quad \text{apply a function twice} \end{aligned}$$

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

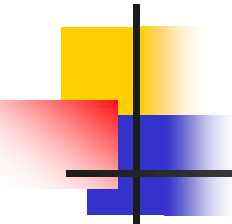
Primitive Recursive Functions

- 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 simple version of fold



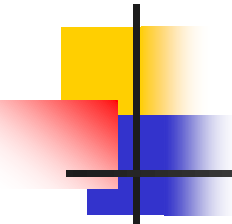
Primitive Recursion over Nat

- $\text{fold } f \ z \ n =$
 $\text{match } n \text{ with } 0 \rightarrow z$
 $| \text{ Suc } m \rightarrow f (\text{fold } f \ z \ m)$
- $\text{fold} \rightarrow \lambda f \ z \ n. n \ f \ z$
- $\text{is_zero } n \rightarrow \text{fold } (\lambda r. \text{False}) \ \text{True } n$



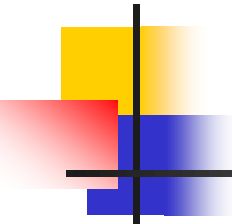
Primitive Recursion over Nat

- $\text{fold } f \ z \ n =$
 $\text{match } n \text{ with } 0 \rightarrow z$
 $| \text{ Suc } m \rightarrow f (\text{fold } f \ z \ m)$
- $\text{fold} \rightarrow \lambda f \ z \ n. n \ f \ z$
- $\text{is_zero } n \rightarrow \text{fold } (\lambda r. \text{False}) \ \text{True} \ n$
 $\xrightarrow{\beta} (\lambda f \ x. f^n \ x) (\lambda r. \text{False}) \ \text{True}$
 $\xrightarrow{\beta} ((\lambda r. \text{False})^n) \ \text{True}$



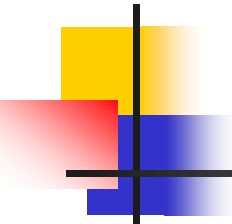
Primitive Recursion over Nat

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



Adding Church Numerals

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

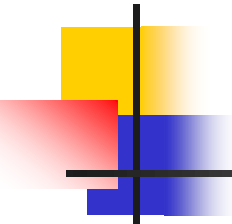


Multiplying Church Numerals

■ $n \rightarrow \lambda f x. f^n x$ and $m \rightarrow \lambda f x. f^m x$

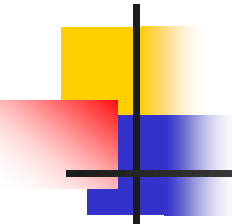
■ $n * m \rightarrow \lambda f x. (f^{n*m}) x = \lambda f x. (f^m)^n x$
 $= \lambda f x. n (m f) x$

$* \rightarrow \lambda n m f x. n (m f) x$



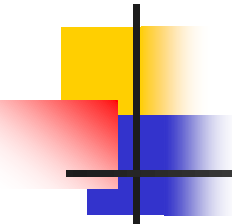
Predecessor

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



Recursion

- Want a λ -term Y such that for all terms R we have $Y R = R (Y R)$
- Y needs to have replication to “remember” a copy of R
- $Y = \lambda y. (\lambda x. y(x x)) (\lambda x. y(x x))$
- $Y R = (\lambda x. R(x x)) (\lambda x. R(x x))$
 $= R ((\lambda x. R(x x)) (\lambda x. R(x x)))$
- Note: requires lazy evaluation
- Called the *Y-combinator*



Factorial

■ Let $F = \lambda f n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f (n - 1)$

$Y F 3 = F (Y F) 3$

$= \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * ((Y F)(3 - 1))$

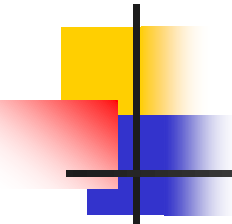
$= 3 * (Y F) 2 = 3 * (F(Y F) 2)$

$= 3 * (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * (Y F)(2 - 1))$

$= 3 * (2 * (Y F)(1)) = 3 * (2 * (F(Y F) 1)) = \dots$

$= 3 * 2 * 1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (Y F)(0 - 1))$

$= 3 * 2 * 1 * 1 = 6$



Y in OCaml

- Y is type-incorrect, but we can define its behavior using let rec:

```
# let rec y f = f (y f);;
```

```
val y : ('a -> 'a) -> 'a = <fun>
```

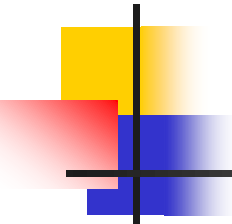
```
# let mk_fact =
```

```
  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>
```

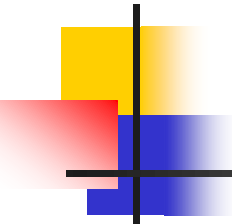
```
# y mk_fact;;
```

```
- : int -> int = <fun>
```

```
# y mk_fact 5;;
```

```
- : int = 120
```

■ Use recursion to get recursion



Some Other Combinators

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