Programming Languages and Compilers (CS 421)



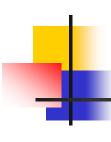
Reza Zamani

http://www.cs.illinois.edu/class/cs421/

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



- Only three kinds of expressions:
 - Variables: x, y, z, w, ...
 - Abstraction: λ x. e
 (Function creation)
 - Application: e₁ e₂



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 /th constructor

How to Represent Booleans

- bool = True | False
- True $\rightarrow \lambda x_1$. λx_2 . $x_1 \equiv_{\alpha} \lambda x$. λy . x
- False $\rightarrow \lambda x_1$. λx_2 . $x_2 \equiv_{\alpha} \lambda x$. λy . y
- Notation
 - Will write

$$\lambda x_1 \dots x_n$$
. e for $\lambda x_1 \dots \lambda x_n$. e $e_1 e_2 e_3 \dots e_n$ for $((...((e_1 e_2) e_3).....)e_n)$



Functions over Enumeration Types

- Write a "match" function
- match e with $C_1 \rightarrow x_1$

$$\mid ...$$
 $\mid C_n -> X_n$

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

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 $\tau = C_1 | ... | C_n$
- match e with $C_1 \rightarrow x_1$

$$\mid ...$$

 $\mid C_n -> X_n$

- $match\tau = \lambda x_1 ... x_n e. e x_1...x_n$
- e = expression (single constructor)
 x_i is returned if e = C_i



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

■ match_{bool} = ?

match for Booleans

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

■ match_{bool} = $\lambda x_1 x_2$ e. e $x_1 x_2$ ≡_α $\lambda x y$ b. b x y



How to Write Functions over Booleans

- if b then x_1 else $x_2 \rightarrow$
- if_then_else b x₁ x₂ = b x₁ x₂
- 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 match_{bool} x_1 x_2 b = $(\lambda x_1 x_2 b \cdot b \cdot b \cdot x_1 x_2) x_1 x_2 b = b \cdot x_1 x_2$
- if_then_else
 - $\equiv \lambda b x_1 x_2$. (match_{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
- → (match_{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

7/25/2011 12

4

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

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



How to Represent Pairs

- Pair has one constructor (comma) that takes two arguments
- type (α, β) pair = (,) $\alpha \beta$
- $(a, b) --> \lambda x \cdot x \cdot a b$
- $(_,_) --> \lambda a b x . x a b$



Functions over Union Types

- Write a "match" function
- match e with $C_1 y_1 ... y_{m1} -> f_1 y_1 ... y_{m1}$ | ... | $C_n y_1 ... y_{mn} -> f_n y_1 ... y_{mn}$
- $match\tau \rightarrow \lambda f_1 ... f_n e. e f_1...f_n$
- 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

- match_{pair} = λ f p. p f
- fst p = match p with (x,y) -> x
- fst $\rightarrow \lambda$ p. match_{pair} (λ x y. x) p = λ p.(λ f p. p f) (λ x y. x)p = 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 $\tau = C_1 t_{11} \dots t_{1k} | \dots | C_n t_{n1} \dots t_{nm}$
- Suppose t_{ih} : τ (ie. is recursive)
- In place of a value t_{ih} have a function to compute the recursive value $r_{ih} x_1 \dots x_n$
- $C_i t_{i1} \dots t_{ih} \dots t_{ij} \rightarrow \lambda x_1 \dots x_n \cdot x_i t_{i1} \dots (r_{ih} x_1 \dots x_n) \dots t_{ij}$
- $C_i \rightarrow \lambda \ t_{i1} \dots r_{ih} \dots t_{ij} \ X_1 \dots \ X_n . X_i \ t_{i1} \dots \ (r_{ih} X_1 \dots X_n) \dots \ t_{ij}$



How to Represent Natural Numbers

- nat = Suc nat | 0
- Suc = λ n f x. f (n f x)
- Suc $n = \lambda f x$. f(n f x)
- $\mathbf{0} = \lambda f \mathbf{x} \cdot \mathbf{x}$
- Such representation called Church Numerals



Some Church Numerals

• Suc $0 = (\lambda n f x. f (n f x)) (\lambda f x. x) --> \lambda f x. f ((\lambda f x. x) f x) --> \lambda f x. f ((\lambda x. x) x) --> \lambda f x. f x.$

Apply a function to its argument once



Some Church Numerals

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

Primitive Recursive Functions

- Write a "fold" function
- fold $f_1 ext{ ... } f_n e = match e$ with $C_1 ext{ y}_1 ext{ ... } ext{ y}_{m1} ext{ ... } ext{ y}_{m1} ext{ ... } ext{ } ext{$
- $fold\tau \rightarrow \lambda f_1 \dots f_n e. e f_1 \dots f_n$
- Match in non recursive case a degenerate version of fold



Primitive Recursion over Nat

- fold f z n=
- match n with 0 -> z
- Suc m -> f (fold f z m)
- fold $\equiv \lambda$ f z n. n f z
- is_zero n = fold (λ r. False) True n
- = (λ f x. f ⁿ x) (λ r. False) True
- = = ((λ r. False) ⁿ) True
- \blacksquare if n = 0 then True else False



Adding Church Numerals

$$\overline{n} \equiv \lambda f x. f^n x$$
 and $m \equiv \lambda f x. f^m x$

•
$$n + m = \lambda f x. f^{(n+m)} x$$

= $\lambda f x. f^{n} (f^{m} x) = \lambda f x. \overline{n} f (\overline{m} f x)$

= + $\equiv \lambda$ n m f x. n f (m f x)

Subtraction is harder



Multiplying Church Numerals

 $\overline{n} \equiv \lambda f x. f^n x$ and $m \equiv \lambda f x. f^m x$

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

 $\overline{*} \equiv \lambda n m f x. n (m f) x$

Predecessor

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

7/25/2011 25

Recursion

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

Factorial

• Let $F = \lambda f n$. 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?).
```

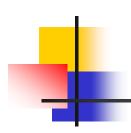
7/25/2011 28

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
```

7/25/2011 29

Use recursion to get recursion



Some Other Combinators

For your general exposure

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