

# Programming Languages and Compilers (CS 421)

---

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class  
/sp07/cs421/](http://www.cs.uiuc.edu/class/sp07/cs421/)

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

# How to Represent (Free) Data Structures (First Pass)

---

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

# How to Represent Booleans

---

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

# How to Write Functions over Data Structures

---

- Write a “match” or “case” function

- match  $c$  with  $C_1 \rightarrow x_1$

| ...

|  $C_n \rightarrow x_n$

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

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

# How to Write Functions over Booleans

---

- If b then c else d
- `if_then_else b x1 x2 = b x1 x2`
- `if_then_else ≡ λ b x1 x2 . b x1 x2`

Alternately,

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

# Example:

---

not b = if b then False else True

= if\_then\_else b False True

= ( $\lambda b c d. b c d$ ) b ( $\lambda x y. y$ )( $\lambda x y. x$ )

= 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

---

Elsa L. Gunter

# How to Represent (Free) Data Structures (Second Pass)

---

- Suppose  $\tau$  is a type with  $n$  constructors:  
 $C_1 t_{11} \dots t_{1k}, \dots, C_n t_{n1} \dots t_{nm},$
- 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 first

# How to Represent Pairs

---

- Pair has one constructor (comma) that takes two arguments
- $(a, b) \rightarrow \lambda x. x a b$
- $(\_, \_) \rightarrow \lambda a b x. x a b$

# Functions over Pairs

---

- $\text{fst} \equiv \lambda p. p (\lambda x y. x)$

$\text{fst } (u, v) \rightarrow$

$(\lambda p. p (\lambda x y. x))((\lambda a b x. x a b) u v) \rightarrow$

$(\lambda p. p (\lambda x y. x))(\lambda x. x u v) \rightarrow$

$(\lambda x. x u v) (\lambda x y. x) \rightarrow$

$(\lambda x y. x) u v \rightarrow (\lambda y. u) v \rightarrow u$

- $\text{snd} \equiv \lambda p. p (\lambda x y. y)$

# How to Represent (Free) Data Structures (Third Pass)

---

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

# How to Represent Natural Numbers

---

- $\text{nat} = \text{Suc nat} \mid 0$
  - $\underline{0} \equiv \lambda f x. x$
  - $\underline{\text{Suc } n} = \lambda f x. f (n f x)$
  - $\text{Suc} \equiv \lambda n f x. f (n f x)$
- 
- Such re presentation called Church Numerals

# Some Church Numerals

---

- 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

---

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

Apply a function twice

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

# Adding Church Numerals

---

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

# Multiplying Church Numerals

---

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

# Primitive Recursion over Nat

---

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

# 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
- 
- 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  $\lambda$ -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  $R$
- $Y = \lambda y. (\lambda x. y(x x)) (\lambda x. y(x x))$
- $Y R = (\lambda x. R(x x)) (\lambda x. R(x x))$
- $= R ((\lambda x. R(x x)) (\lambda x. R(x x)))$
- Notice: Requires lazy evaluation

# 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

---

```
# 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
```

- Used recursion to get recursion

# Some Other Combinators

---

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