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

- 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*τ = $\lambda\ x_1 \ldots x_n\ e.\ e\ x_1 \ldots x_n$

- 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 $\rightarrow \lambda\, x_1\, x_2.\, x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\, y$

- $\text{match}_{\mathit{bool}}$ = ?

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 $\rightarrow$
  $match_{bool}\ x_1\ x_2\ b$ =
  $(\lambda\ x_1\ x_2\ b\ .\ b\ x_1\ x_2\ )\ x_1\ x_2\ b = b\ x_1\ x_2$
- if_then_else
  $\equiv \lambda\ b\ x_1\ x_2.\ (match_{bool}\ x_1\ x_2\ b)$
  $= \lambda\ b\ x_1\ x_2.\ (\lambda\ x_1\ x_2\ b\ .\ b\ x_1\ x_2\ )\ x_1\ x_2\ b$
  $= \lambda\ b\ x_1\ x_2.\ b\ x_1\ x_2$

Example:

not b

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

$\rightarrow$ (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

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

- Suppose τ is a type with *n* constructors:
  type $\tau \;= C_1\, t_{11} \ldots t_{1k} \mid \ldots \mid C_n\, t_{n1} \ldots t_{nm},$
- Represent each term as an abstraction:
- $C_i\, t_{i1} \ldots t_{ij,} \rightarrow \lambda\, \mathrm{x}_1 \ldots \mathrm{x}_n.\ \mathrm{x}_i\, t_{i1} \ldots t_{ij,}$
- $C_i \rightarrow \lambda\, t_{i1} \ldots t_{ij,}\, \mathrm{x}_1 \ldots \mathrm{x}_n\, .\ \mathrm{x}_i\, t_{i1} \ldots t_{ij,}$
- 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 $C_1\ y_1 \ldots y_{m1}$ -> $f_1\ y_1 \ldots y_{m1}$
  | …
  | $C_n\ y_1 \ldots y_{mn}$ -> $f_n\ y_1 \ldots y_{mn}$
- $match_\tau \rightarrow \lambda f_1 \ldots f_n e.\ e\ f_1 \ldots f_n$
- 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

- $match_{pair} = \lambda\, f\, p.\ p\, f$

- fst p = match p with (x,y) -> x
- fst $\rightarrow \lambda\, p.\ match_{pair}\ (\lambda\, x\, y.\ x)$

  $= (\lambda\, f\, p.\ p\, f)\ (\lambda\, x\, y.\ x) = \lambda\, p.\ p\ (\lambda\, x\, y.\ x)$

- 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:

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

How to Represent Natural Numbers

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

Some Church Numerals

- $\overline{\text{Suc 0}}$ = (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 $C_1\ y_1 \ldots y_{m1}$ -> $f_1\ y_1 \ldots y_{m1}$
  | …
  | $Ci\ y_1 \ldots r_{ij} \ldots y_{in}$ -> $f_n\ y_1 \ldots (\text{fold } f_1 \ldots f_n\ r_{ij}) \ldots y_{mn}$
  | …
  | $C_n\ y_1 \ldots y_{mn}$ -> $f_n\ y_1 \ldots y_{mn}$
- *fold*$\tau \rightarrow \lambda\, f_1 \ldots f_n\, e.\ e\ f_1 \ldots f_n$
- 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)
- $\overline{\text{fold}} \equiv \lambda$ 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) (λ r. False) True
- $= ((\lambda$ r. False$)^{n}$) True
- $\equiv$ if n = 0 then True else False

Adding Church Numerals

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

Multiplying Church Numerals

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

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

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

Predecessor

- let pred_aux n =
 match n with 0 -> (0,0)
 | Suc m
-> (Suc(fst(pred_aux m)), fst(pred_aux m)
= fold (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

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