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

---

Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation
- λ−calculus is a theory of computation
- "The Lambda Calculus: Its Syntax and Semantics". H. P. Barendregt. North Holland, 1984

Lambda Calculus - Motivation

- All *sequential programs* may be viewed as functions from input (initial state and input values) to output (resulting state and output values).
- λ-calculus is a mathematical formalism of functions and functional computations
- Two flavors: typed and untyped

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation, think fun x -> e)
  - Application: e_1 e_2

Untyped λ-Calculus Grammar

- Formal BNF Grammar:
  - <expression> ::= <variable>
    | <abstraction>
    | <application>
    | (<expression>)
  - <abstraction>
    ::= λ<variable>.<expression>
  - <application>
    ::= <expression> <expression>

Untyped λ-Calculus Terminology

- Occurrence: a location of a subterm in a term
- Variable binding: λ x. e is a binding of x in e
- Bound occurrence: all occurrences of x in λ x. e
- Free occurrence: one that is not bound
- Scope of binding: in λ x. e, all occurrences in e not in a subterm of the form λ x. e' (same x)
- Free variables: all variables having free occurrences in a term

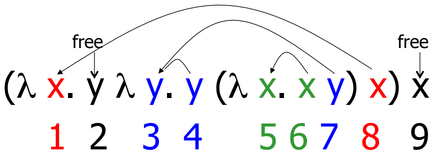
Example

- Label occurrences and scope:

(λ x. y λ y. y (λ x. x y) x) x
1 2 3 4 5 6 7 8 9

Example

- Label occurrences and scope:

- 

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

$$\frac{E \dashrightarrow E''}{E\ E' \dashrightarrow E''\ E'}$$

- Application (version 1 - Lazy Evaluation)

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

- Application (version 2 - Eager Evaluation)

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

$$\frac{}{(\lambda\ x\ .\ E)\ V \dashrightarrow E[V/x]}$$

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 recursion)

Uses of λ-Calculus

- Typed and untyped λ-calculus used for theoretical study of sequential programming languages
- Sequential programming languages are essentially the λ-calculus, extended with predefined constructs, constants, types, and syntactic sugar
- Ocaml is close to the λ-Calculus:

  fun x -> exp --> λ x. exp

  let x = e_1 in e_2 --> (λ x. e_2)e_1

α Conversion

- α-conversion:

  λ x. exp --α--> λ 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

---

α Conversion Non-Examples

1. Error: y is not free in termsecond

$$\lambda x.\ x\ y \ \cancel{\xrightarrow{\alpha}}\ \lambda y.\ y\ y$$

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

$$\lambda x.\ \underbrace{\lambda y.\ x\ y}_{\text{exp}} \ \cancel{\xrightarrow{\alpha}}\ \lambda y.\ \underbrace{\lambda y.\ y\ y}_{\text{exp[y/x]}}$$

But λ x. (λ y. y) x --α--> λ y. (λ y. y) y

And λ y. (λ y. y) y --α--> λ x. (λ y. y) x

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- it is an equivalence relation
- 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: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

- λ x. (λ y. y x) x --α--> λ z. (λ y. y z) z so
  λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z
- (λ y. y z) --α--> (λ x. x z) so
  (λ y. y z) ~α~ (λ x. x z) so
  λ z. (λ y. y z) z ~α~ λ z. (λ x. x z) z
- λ z. (λ x. x z) z --α--> λ y. (λ x. x y) y so
  λ z. (λ x. x z) z ~α~ λ y. (λ x. x y) y
- λ x. (λ y. y x) x ~α~ λ y. (λ 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: $(\lambda\, x.\ P)\ N \text{ --}\beta\text{--> } P\ [N /x]$
- Essence of computation in the lambda calculus
- Usually defined on α-equivalence classes of terms

Example

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

α β Equivalence

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

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

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

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

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z))

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z) ~β~ λ z. z

Example 2

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

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

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

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

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

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

$\rightarrow\ \ \lambda\ x_1\ \ldots\ x_n$ 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 $\tau\ = C_1|\ldots|C_n$
- match e with C_1 -> x_1
  | …
  | C_n -> x_n
- *match*$\tau = \lambda\ x_1\ \ldots\ x_n$ 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 $\rightarrow \lambda\, x_1\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\ y$

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

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

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 $\rightarrow$
  $\mathrm{match}_{\mathrm{bool}}\ x_1\ x_2\ b$ =
  $(\lambda\, x_1\, x_2\, b\ .\ b\, x_1\, x_2\ )\ x_1\, x_2\, b = b\, x_1\, x_2$
- if_then_else
  $\equiv \lambda\, b\, x_1\, x_2.\ (\mathrm{match}_{\mathrm{bool}}\ x_1\, x_2\, b)$
  $= \lambda\, b\, x_1\, x_2.\ (\lambda\, x_1\, x_2\, b\ .\ b\, x_1\, x_2\ )\ x_1\, x_2\, b$
  $= \lambda\, b\, x_1\, x_2.\ b\, x_1\, x_2$

Example:

not b
 = match b with True -> False | False -> True
 $\rightarrow (\mathrm{match}_{\mathrm{bool}})$ 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:
- $C_i\, t_{i1} \ldots t_{ij,} \rightarrow \lambda\, x_1 \ldots x_n.\ x_i\, t_{i1} \ldots t_{ij,}$
- $C_i \rightarrow \lambda\, t_{i1} \ldots t_{ij,}\, x_1 \ldots x_n .\ x_i\, t_{i1} \ldots t_{ij,}$
- 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 = (,) $\alpha\ \beta$
- (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

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

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

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

Apply a function to its argument once

Some Church Numerals

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

Apply a function twice

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

---

Primitive Recursive Functions

- Write a “fold” function
- fold $f_1 \ldots f_n$ = match e
  with $C_1\ y_1 \ldots y_{m1}$ -> $f_1\ y_1 \ldots y_{m1}$
  | …
  | $C_i\ y_1 \ldots r_{ij} \ldots y_{in}$ -> $f_n\ y_1 \ldots (\text{fold } f_1 \ldots f_n\ r_{ij}) \ldots y_{mn}$
  | …
  | $C_n\ y_1 \ldots y_{mn}$ -> $f_n\ y_1 \ldots y_{mn}$
- $\textit{fold}\tau \rightarrow\ \ \lambda\, f_1 \ldots f_n\, e.\ e\ f_1 \ldots f_n$
- 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)
- $\overline{\text{fold}} \equiv \lambda\, f\, z\, n.\ n\, f\, z$
- $\overline{\text{is\_zero}}\ \ \overline{n} = \overline{\text{fold}}\ (\lambda\, r.\ \overline{\text{False}})\ \overline{\text{True}}\ \overline{n}$
- $= (\lambda\, f\, x.\ f^{\,n}\, x)\ (\lambda\, r.\ \text{False})\ \text{True}$
- $= ((\lambda\, r.\ \text{False})^{\,n}\ )\ \text{True}$
- $\equiv$ if n = 0 then True else False

Adding Church Numerals

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

Multiplying Church Numerals

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

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

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 ≡ 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 = λ x . x
- K = λ x. λ y. x
- K_* = λ x. λ y. y
- S = λ x. λ y. λ z. x z (y z)