Programming Languages and Compilers (CS 421)

Sasa Misailovic
4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

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

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 deterministic *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 expression, 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:

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

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**)
  (λ x . E) E' --> 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 – Value = variable or abstraction

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is Turing Complete
  - Can express any deterministic sequential computation
- Problems:
  - How to express basic data: bools, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar (more on this later this week!)

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 aren't syntactic sugar! They disallow some terms
- <u>Simply typed</u> λ-calculus is less powerful than the untyped λ-Calculus:
  - NOT Turing Complete (no general recursion). See e.g.:
  - https://math.stackexchange.com/questions/1319149/what-breaks-the-turing-completeness-of-simply-typed-lambda-calculus
  - http://okmij.org/ftp/Computation/lambda-calc.html#predecessor

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 λ-Calculus:

  fun x -> exp == λ x. exp

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

α Conversion (aka Substitution)

- 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 the second term

   λ x. x y ~~--α-->~~ λ y. y y
2. Error: free occurrence of x becomes bound in wrong way when replaced by y

   λ x. λ y. x y ~~--α-->~~ λ y. λ y. y y

   (λ y. x y is exp; λ y. y y is 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 <u>lambda terms</u>. Then ~ is a congruence if:

- It is an equivalence relation
  - Reflexive, symmetric, transitive
- And if e_1 ~ e_2 then
  - (e e_1) ~ (e e_2) and (e_1 e) ~ (e_2 e)
  - λ x. e_1 ~ λ x. e_2

α Equivalence

- α equivalence is the smallest congruence containing α conversion
  - Notation: e_1 ~α~ e_2
- **One usually treats α-equivalent terms as equal** - i.e. use α equivalence classes of terms
  - "Equivalent up to renaming"

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
- Therefore: 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: Detailed Rules

P [N / x] means replace every free occurrence of variable x in redex P by residue N

- x [N / x] = N
- y [N / x] = y if y ≠ x
- (e_1 e_2) [N / x] = ((e_1 [N / x]) (e_2 [N / x]))
- (λ x. e) [N / x] = (λ x. e)
- (λ y. e) [N / x] = λ y. (e [N / x]) provided y ≠ x and y not free in N
  - Rename y in redex if necessary

Example

(λ y. y z) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] --α-->
- (λ w.w z) [(λ x. x y) / z] =
- λ w. w (λ x. x y)

Example

- Only replace free occurrences
- (λ y. y z (λ z. z)) [(λ x. x) / z] =

  λ y. y (λ x. x) (λ z. z)

Not

λ y. y (λ x. x) (λ z. (λ x. x))

β reduction

- β Rule: (λ x. P) N --β--> P [N /x]
- **Essence of computation** in the lambda calculus
- Usually defined on α-equivalence classes of terms

Example

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

- (λ x. x x) (λ x. x x)
 --β--> (λ x. x x) (λ x. x x)
 --β--> (λ x. x x) (λ x. x x) --β--> ….

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 αβ-equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

- Not all terms reduce to normal forms
  - Computations may be infinite
- Not all reduction strategies will produce a normal form if one exists
- We will explore two common reduction strategies next!

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

Eager evaluation

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then β-reduce the application

Example 1

- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))
- **Lazy** evaluation:
  - Reduce the left-most application:
- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))
  --β--> (λ x. x)

Example 1

- (λ z. (λ x. x))((λ y. y y) (λ y. y y))
- **Eager** evaluation:
  - Reduce the operator of the top-most application to an abstraction: Done.
  - Reduce the argument:
- (λ z. (λ x. x))((λ y. y y) (λ y. y y))

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

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

(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:**

(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))

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)

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))

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))

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) (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) (λ z. z) --β-->
(λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- **Eager 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))
- **Eager evaluation:**

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- **Eager evaluation:**

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

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- **Eager evaluation:**

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

Example 2

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

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

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e
  - Application: e_1 e_2
- Notation – will write:
  $\lambda x_1 \ldots x_n.$ e for $\lambda x_1. \lambda x_2. \ldots \lambda x_n.$ e
  $e_1\, e_2 \ldots e_n$ for $((\ldots((e_1\, e_2)\, e_3) \ldots e_{n-1})\ e_n$

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)
  - `type τ = C1 | … | Cn` (C_1 … C_n)
- 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 $\rightarrow$ $\lambda x_1. \lambda x_2. x_1 \equiv_\alpha \lambda x. \lambda y. x$
- False $\rightarrow$ $\lambda x_1. \lambda x_2. x_2 \equiv_\alpha \lambda x. \lambda y. y$

Functions over Enumeration Types

- Write a "match" function
- match e with C_1 -> x_1

```
                | …
                | Cn -> xn
```

$\rightarrow \lambda x_1 \ldots x_n\, e.\ e\, x_1 \ldots x_n$

- Think: give me what to do in each case and give the selector (the constructor expression), and I'll apply that case

Functions over Enumeration Types

```
type τ  = C1|…|Cn
match e with C1 -> x1
             | …
             | Cn -> xn
```

- Recall: $C_i \rightarrow \lambda x_1 \ldots x_n.\ x_i$
- Then: match $\tau = \lambda x_1 \ldots x_n\, e.\ e\, x_1 \ldots x_n$
- e = expression (single constructor instance). Then, "match C_i" selects x_i

match for Booleans

```
type τ  = C1|…|Cn
match e with C1 -> x1
           | …
           | Cn -> xn
```

- Recall: $C_i \to \lambda\, x_1 \ldots x_n.\, x_i$
- Then: match $\tau = \lambda\, x_1 \ldots x_n\, e.\ e\ x_1 \ldots x_n$

- bool = True | False
- True $\to \lambda\, x_1\, x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\ x$
- False $\to \lambda\, x_1\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\ y$

- match_{bool} = ?

match for Booleans

```
type τ  = C1|…|Cn
match e with C1 -> x1
           | …
           | Cn -> xn
```

- Recall: $C_i \to \lambda\, x_1 \ldots x_n.\, x_i$
- Then: match $\tau = \lambda\, x_1 \ldots x_n\, e.\ e\ x_1 \ldots x_n$

- bool = True | False
- True $\to \lambda\, x_1\, x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\ x$
- False $\to \lambda\, x_1\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\ y$

- $\text{match}_{bool} = \lambda\, x_1\, x_2\, e.\ e\ x_1\, x_2$

  $\equiv_\alpha \ \lambda\, x\, y\, b.\ b\ x\ y$

How to Write Functions over Booleans

- bool = True | False
- True $\to \lambda\, x_1\, x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\ x$
- False $\to \lambda\, x_1\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\ y$

- if b then x_1 else $x_2 \to$

  if_then_else b $x_1\ x_2$ = b $x_1\ x_2$

  if_then_else $\equiv_\alpha \lambda\, b\, x_1\, x_2\ .\ b\ x_1\ x_2$

How to Write Functions over Booleans

- bool = True | False
- True $\to \lambda\, x_1\, x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\ x$
- False $\to \lambda\, x_1\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\ y$

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

- Alternately:
- if b then x_1 else x_2 =

  match b with True -> x_1 | False -> x_2

$\to$

$\text{match}_{bool}\ x_1\ x_2\ b = (\lambda\, x_1\, x_2\, b\ .\ b\ x_1\ x_2\ )\ x_1\ x_2\ b$

$= b\ x_1\ x_2$

- if_then_else

  $\equiv_\alpha \lambda\, b\, x_1\, x_2.\ (\text{match}_{bool}\ x_1\ x_2\ b)$

  $= \ \lambda\, b\, x_1\, x_2.\ (\lambda\, x_1\, x_2\, b\ .\ b\ x_1\ x_2\ )\ x_1\ x_2\ b$

  $= \ \lambda\, b\, x_1\, x_2.\ b\ x_1\ x_2$

Example:

- bool = True | False
- True $\to \lambda\, x_1\, x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\ x$
- False $\to \lambda\, x_1\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\ y$

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

not b

= match b with True -> False

| False -> True

$\to (\text{match}_{bool})$ 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 (λ x y. y)(λ x y. x)**
- Try other operators: and, or, xor

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

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

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

Functions over Pairs

- (a , b) → λ x . x a b

- $match_{pair}$ = λ f p. p f
- `fst p = match p with (x,y) -> x`
- fst → λ p. $match_{pair}$ (λ x y. x)
  = (λ f p. p f) (λ x y. x)
  = λ p. p (λ x y. x)
- snd → λ p. p (λ x y. y)

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

- Suppose τ is a type with n constructors: type τ = $C_1\ t_{11} \ldots t_{1k} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nm}$,
- Represent each term as an abstraction:
- $C_i\ t_{i1} \ldots t_{ij}, \to \lambda\ x_1 \ldots x_n.\ x_i\ t_{i1} \ldots t_{ij},$
- $C_i \to \lambda\ t_{i1} \ldots t_{ij}, x_1 \ldots x_n .\ x_i\ t_{i1} \ldots t_{ij},$
- Think: you need to give each constructor its arguments first

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 τ → λ $f_1 \ldots f_n$ 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 function with the data in that case

How to Represent (Free) Data Structures (Third Pass - Recursive Types)

- Suppose τ is a type with n constructors:
  `type τ = C1 t11 … t1k | … |Cn tn1 … tnm,`
- Suppose t_{ih} : τ (i.e. is recursive)
- In place of a value t_{ih} have a function to compute the recursive value $r_{ih}\ x_1 \ldots x_n$
- Ci ti1 … **rih** …tij → λ x1 … xn . xi ti1 … **(rih x1 … xn)** … tij
- Ci → λ ti1 … **rih** …tij x1 … xn . xi ti1 … **(rih x1 … xn)** … tij

How to Represent Natural Numbers

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

Some Church Numerals

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

- $\overline{1}$
- $\overline{\textbf{Suc 0}}$ = (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

- "Do something (anything) once"

11/25/2018 67

Some Church Numerals

- $\overline{2}$
- $\overline{\textbf{Suc(Suc 0)}}$ = (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

- "Do something (anything) once"

In general $\overline{n}$ = λ f x. f (… (f x)…) with n applications of f (do "something" n times)

68

Some Church Numerals

- $\overline{0}$ = λ f x. x
- $\overline{1}$ = λ f x. f x
- $\overline{2}$ = λ f x. f f x
- $\overline{3}$ = λ f x. f f f x
- $\overline{4}$ = λ f x. f f f f x
- $\overline{5}$ = λ f x. f f f f f x
- ….
- $\overline{n}$ = λ f x. f^n x

11/26/2018 69

Primitive Recursive Functions

- Write a "fold" function
- fold f_1 … f_n = match e with

```
    C1 y1 … ym1 -> f1 y1 … ym1
  | …
  | Ci y1 … rij …yin -> fn y1 … (fold f1 … fn rij) …ymn
  | …
  | Cn y1 … ymn -> fn y1 … ymn
```

- fold τ → λ f_1 … f_n e. e f_1…f_n
- Match in non recursive case a degenerate version of fold

11/25/2018 70

Primitive Recursion over Nat

- $\overline{n}$ ≡ λ f x. f^n x

```
fold f z n =
   match n with 0 -> z
           | Suc m -> f (fold f z m)
```

- $\overline{\text{fold}}$ ≡ λ f z n. n f z
- $\overline{\text{is\_zero}}$ $\overline{n}$ = $\overline{\text{fold}}$ (λ r. $\overline{\text{False}}$) $\overline{\text{True}}$ $\overline{n}$
  = (λ f x. f^n x) (λ r. False) True
  = ((λ r. False) ^{n}) True
  ≡ if n = 0 then True else False

11/25/2018 71

Adding Church Numerals

- $\overline{n}$ ≡ λ f x. f^n x and $\overline{m}$ ≡ λ f x. f^m x
- $\overline{n + m}$ = λ f x. $f^{(n+m)}$ x
  = λ f x. f^n (f^m x) = λ f x. $\overline{n}$ f ($\overline{m}$ f x)
- $\overline{\textbf{+}}$ **≡ λ n m f x. n f (m f x)**
- Subtraction is harder
  (needs to refer to predecessors)

11/25/2018 72

How much is 2+2 ?

- $\overline{+} = \lambda\ n\ m\ f\ x.\ n\ f\ (m\ x)$
- $\overline{2} = \lambda\ f\ x.\ f\ (f\ x)$
- $\overline{2} = \lambda\ f\ x.\ f\ (f\ x)$
- So let's begin:

  $(\lambda\ \mathbf{n}\ \mathbf{m}\ f\ x.\ n\ f\ (m\ f\ x))\ \overline{2}\ \overline{2}$ --β-->

  $\lambda\ f\ x.\ (\lambda\ \mathbf{f\ x.\ f\ (f\ x)})\ f\ ((\lambda\ \mathbf{f\ x.\ f\ (f\ x)})\ f\ x)$ --β-->

  $\lambda\ f\ x.\ (\lambda\ \mathbf{f\ x.\ f\ (f\ x)})\ \mathbf{f}\ (\mathbf{f\ (f\ x)})$ --β-->

  $\lambda\ f\ x.\ \mathbf{f\ (f\ (f\ (f\ x)))} \equiv$

  $\overline{4}$

Multiplying Church Numerals

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

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

How much is $\overline{2}\ \overline{*}\ \overline{2}$?

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: Y-Combinator (the original one)

- 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 = 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**

  (see example 1 on eager vs lazy much earlier in this deck!)

Factorialb

- 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 Evaluation of 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

- More about Y-combinator:
  - https://mvanier.livejournal.com/2897.html
- 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)
  - https://en.wikipedia.org/wiki/SKI_combinator_calculus