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

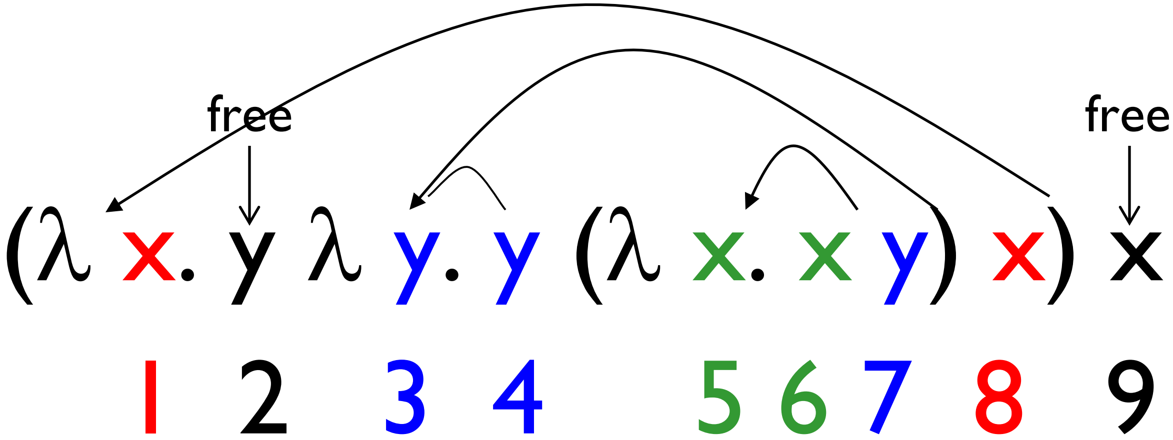
Example

- Label occurrences and scope:

$$(\lambda\ \underset{1}{x}.\ \underset{2}{y}\ \lambda\ \underset{3}{y}.\ \underset{4}{y}\ (\lambda\ \underset{5}{x}.\ \underset{6}{x}\ \underset{7}{y})\ \underset{8}{x})\ \underset{9}{x}$$

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

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

$$\frac{E \text{ --> } E''}{E\ E' \text{ --> } E''\ E'}$$

- Application (version 1 - **Lazy Evaluation**)

$$(\lambda\, x\, .\, E)\ E' \text{ --> } E[E'/x]$$

- Application (version 2 - **Eager Evaluation**)

$$\frac{E' \text{ --> } E''}{(\lambda\, x\, .\, E)\ E' \text{ --> } (\lambda\, x\, .\, E)\ E''}$$

$$\frac{}{(\lambda\, x\, .\, E)\ V \text{ --> } E[V/x]}$$

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
- Simply typed 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:

$$\text{fun } x \text{ -> exp} \quad == \quad \lambda\, x.\ \text{exp}$$

$$\text{let } x = e_1 \text{ in } e_2 \quad == \quad (\lambda\, x.\ e_2)\ e_1$$

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

$$\lambda x.\ x\ y \ \cancel{\text{--}\alpha\text{-->}}\ \lambda y.\ y\ y$$

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

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

But $\lambda x.\ (\lambda y.\ y)\ x \text{ --}\alpha\text{--> } \lambda y.\ (\lambda y.\ y)\ y$

And $\lambda y.\ (\lambda y.\ y)\ y \text{ --}\alpha\text{--> } \lambda x.\ (\lambda y.\ y)\ x$

Congruence

Let ~ be a relation on lambda terms. Then ~ is a congruence if:

- It is an equivalence relation
  - Reflexive, symmetric, transitive
- And 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
  - Notation: $e_1 \sim\alpha\sim e_2$
- **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 $y \neq x$
- $(e_1\ e_2)\ [N / x] = ((e_1\ [N / x]\ )\ (e_2\ [N / x]\ ))$
- $(\lambda\ x.\ e)\ [N / x] = (\lambda\ x.\ e)$
- $(\lambda\ y.\ e)\ [N / x] = \lambda\ y.\ (e\ [N / x]\ )$ 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
- $(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z]$ --α-->
- $(\lambda\ w.w\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] =$
- $\lambda\ w.\ w\ (\lambda\ x.\ x\ y)$

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: (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 <u>top application</u> 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

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

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

Example 2

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

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

Example 2

- $(\lambda\, x.\, x\ x)((\lambda\, y.\, y\ y)\ (\lambda\, z.\, z))$
- **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

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

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --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))

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

Example 2

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

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

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

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

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-->

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

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

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

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))

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

--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-->

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

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

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

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

Example 2

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

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

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

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

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

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

$(\lambda\ z.\ z)\ (\lambda\ z.\ z)$

Example 2

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

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))\ --\beta-->$

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

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

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

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

$(\lambda\ z.\ z)\ (\lambda\ z.\ z)\ --\beta-->$

$(\lambda\ z.\ z)$

Example 2

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

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

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-->

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

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

- (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-->

$\boxed{\lambda\ z.\ z}$

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 $((\ldots((e_1\ e_2)\ e_3)\ \ldots\ e_{n-1})\ \ e_n$

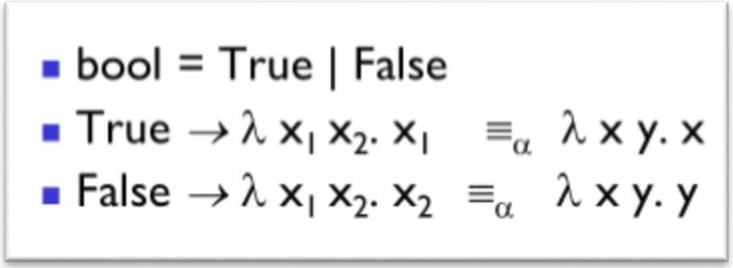
How to Represent (Free) Data Structures (First Pass - Enumeration Types)

- Suppose τ is a type with n constructors: $C_1, \ldots, C_n$ (no arguments)
  - `type τ  = C₁ | … | Cₙ`
- 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 → $\lambda\, x_1.\ \lambda\, x_2.\ x_1 \quad \equiv_\alpha \quad \lambda\, x.\ \lambda\, y.\ x$
- False → $\lambda\, x_1.\ \lambda\, x_2.\ x_2 \quad \equiv_\alpha \quad \lambda\, x.\ \lambda\, y.\ y$

How to Write Functions over Booleans

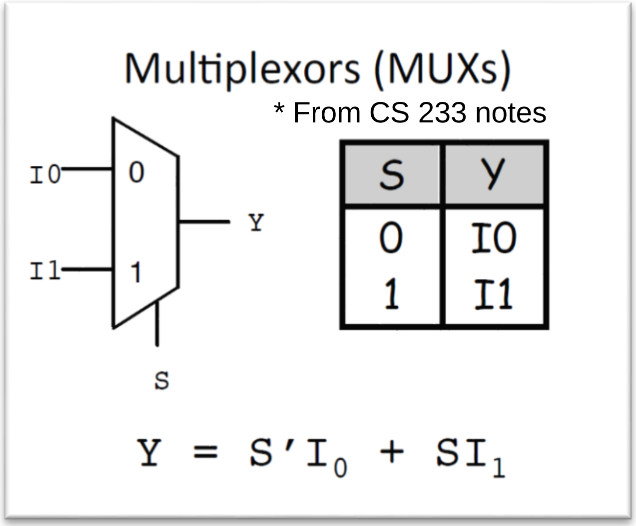


- bool = True | False
- True $\rightarrow \lambda\, x_1\, x_2.\, x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\, x$
- False $\rightarrow \lambda\, x_1\, x_2.\, x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\, y$

- if b then x_1 else x_2 $\rightarrow$

  $$\text{if\_then\_else}\ b\ x_1\ x_2 = b\ x_1\ x_2$$

  $$\text{if\_then\_else} \equiv_\alpha \lambda\, b\, x_1\, x_2\, .\, b\, x_1\, x_2$$



Multiplexors (MUXs)

* From CS 233 notes

S	Y
0	I0
1	I1

$$Y = S'I_0 + SI_1$$

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

$\texttt{type}\ \tau\ =\ C_1|\ldots|C_n$

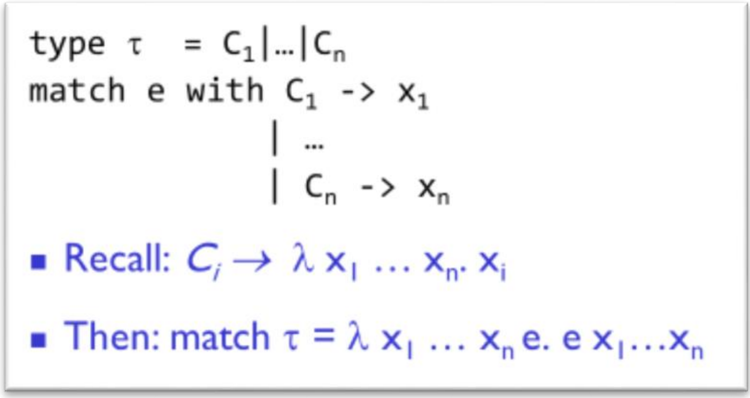
$\texttt{match e with}\ C_1\ \texttt{->}\ x_1$

$\qquad\qquad\qquad\quad |\ \ldots$

$\qquad\qquad\qquad\quad |\ C_n\ \texttt{->}\ x_n$

- 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



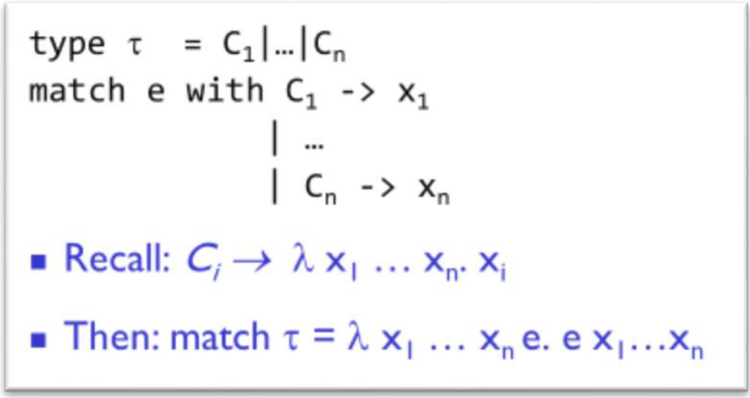
```
type τ  = C₁|…|Cₙ
match e with C₁ -> x₁
           | …
           | Cₙ -> xₙ
```

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

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

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

match for Booleans



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

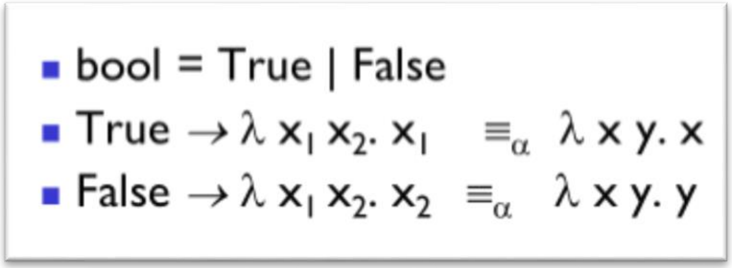
- Recall: $C_i \rightarrow \lambda\, x_1 \ldots x_n.\, x_i$
- Then: match $\tau = \lambda\, x_1 \ldots x_n\, e.\; e\, x_1 \ldots x_n$

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

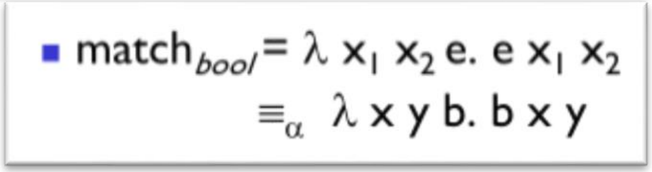
- $\text{match}_{bool} = \lambda\, x_1\, x_2\, e.\; e\, x_1\, x_2$

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

How to Write Functions over Booleans



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



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

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

- Alternately:
- if b then x_1 else x_2 =

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

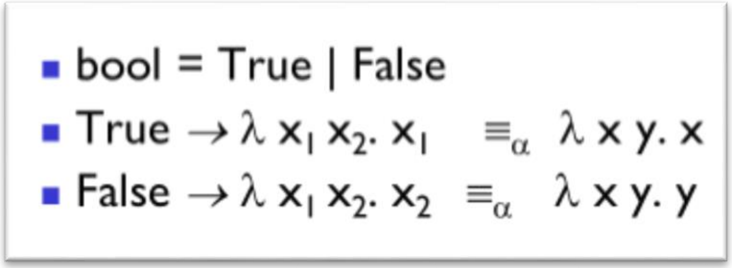
$\rightarrow$

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

- if_then_else

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

Example:



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

not b

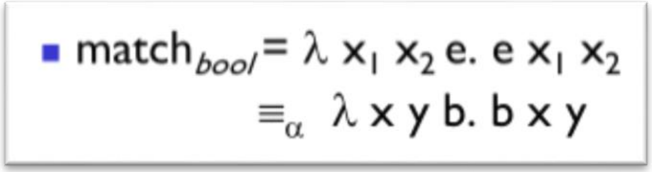
= match b with True -> False

| False -> True

$\rightarrow$ $(\text{match}_{\text{bool}})$ False True b

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

$= b\, (\lambda\, x\, y.\, y)\, (\lambda\, x\, y.\, x)$



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

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

- **not $\equiv \lambda$ b. b (λ x y. y)(λ x y. x)**
- Try other operators: and, or, xor

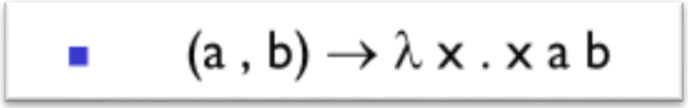
How to Represent (Free) Data Structures (Second Pass - Union Types)

- Suppose τ is a type with n constructors: type $\tau = C_1\ t_{11} \ldots t_{1k} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nm,}$
- 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 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



- $(a\ ,\ b) \rightarrow \lambda\ x\ .\ x\ a\ b$

- $\text{match}_{\text{pair}} = \lambda\ f\ p.\ p\ f$

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

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

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

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

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 first

Functions over Union Types

- Write a “match” function
- $$\begin{array}{ll}\texttt{match e with} & C_1\ y_1 \dots y_{m1}\ \texttt{->}\ f_1\ y_1 \dots y_{m1} \\ & \texttt{|}\ \dots \\ & \texttt{|}\ C_n\ y_1 \dots y_{mn}\ \texttt{->}\ f_n\ y_1 \dots y_{mn}\end{array}$$

- match $\tau \to \lambda f_1 \dots f_n\, e.\ e\ f_1 \dots f_n$

- 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,`

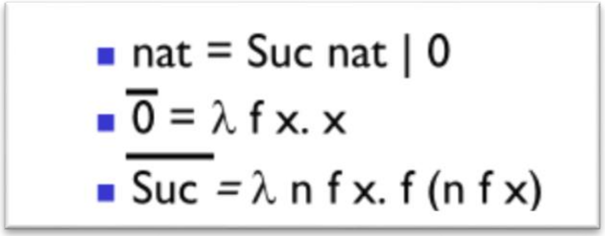
- Suppose $t_{ih} : \tau$ (i.e. is recursive)

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

Some Church Numerals



- nat = Suc nat | 0
- $\overline{0}$ = λ f x. x
- $\overline{\text{Suc}}$ = λ 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"

Some Church Numerals

- $\overline{2}$
- **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)

Some Church Numerals

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

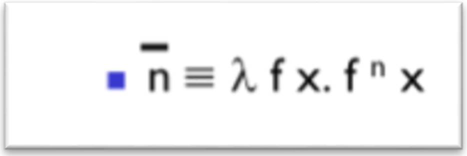
Primitive Recursive Functions

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

    C_1 y_1 … y_{m1} -> f_1 y_1 … y_{m1}
  | …
  | Ci y_1 … r_{ij} …y_{in} -> f_n y_1 … (fold f_1 … f_n r_{ij}) …y_{mn}
  | …
  | C_n y_1 … y_{mn} -> f_n y_1 … y_{mn}

- fold $\tau \rightarrow$ λ f_1 … f_n e. e f_1…f_n
- Match in non recursive case a degenerate version of fold

Primitive Recursion over Nat



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

```
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 $\overline{m} \equiv \lambda\, f\, x.\, f^{\,m}\, x$

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

- $\overline{\mathbf{+}} \equiv \boldsymbol{\lambda}\, \mathbf{n\ m\ f\ x.\ n\ f\ (m\ f\ x)}$

- Subtraction is harder (e.g. has to refer to predecessors)

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:

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

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

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

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

$\overline{\mathbf{4}}$

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 $\overline{\mathbf{2}}\ \overline{*}\ \overline{\mathbf{2}}$?

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!)

Factorial (Lazy): **Y R = R (Y R)**

- Let R = λ f n. if n = 0 then 1 else n * f (n - 1)

Y R 3 = **R** (Y R) 3
= **if 3 = 0 then 1 else 3** * ((Y R)(3 - 1))
= **3** * (Y R) 2
= 3 * (**R(Y R)** 2)
= 3 * (**if 2 = 0 then 1 else 2 * (Y R)(2 - 1)**)
= 3 * (**2 * (Y R)(1)**)
= 3 * (2 * **(F(Y R) 1)**) =…
= 3 * 2 * 1 * (**if 0 = 0 then 1 else 0*(Y R)(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:
  - $I = \lambda\, x\, .\, x$
  - $K = \lambda\, x.\ \lambda\, y.\ x$
  - $K_* = \lambda\, x.\ \lambda\, y.\ y$
  - $S = \lambda\, x.\ \lambda\, y.\ \lambda\, z.\ x\ z\ (y\ z)$
  - https://en.wikipedia.org/wiki/SKI_combinator_calculus