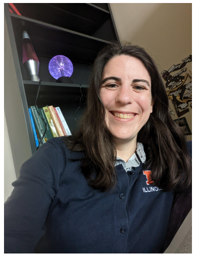
Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC



https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Lambda Calculus, Continued

Questions before we start?

Does every term have a normal form?

Try to normalize:

$(\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

Evaluation

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists**

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists**

Cannot normalize:

$(\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists**

Lazy vs. Eager

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists**

Lazy vs. Eager

Order of Evaluation

Lazy Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Order of Evaluation

Lazy Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Reduce leftmost applications first

Order of Evaluation

Lazy Evaluation

$$\frac{E \rightarrow E''}{\mathbf{E}\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Reduce leftmost applications first

Order of Evaluation

Lazy Evaluation

$$\frac{\mathbf{E} \rightarrow \mathbf{E''}}{\mathbf{E}\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Reduce leftmost applications first

Order of Evaluation

Lazy Evaluation

$$\frac{\mathbf{E} \rightarrow \mathbf{E''}}{\mathbf{E}\ E' \rightarrow \mathbf{E''}\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Reduce leftmost applications first

Order of Evaluation

Lazy Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Delay computation of function arguments

Order of Evaluation

Lazy Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\, x\, .\, E)\ \mathbf{E'} \rightarrow E[E'/x]}\ \textbf{L-App}$$

Delay computation of function arguments

Order of Evaluation

Lazy Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ \mathbf{E'} \rightarrow E[\mathbf{E'}/x]}\ \textbf{L-App}$$

Delay computation of function arguments

Order of Evaluation

Lazy Evaluation

$$\frac{E \to E''}{E\ E' \to E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\, x\, .\, E)\ E' \to E[E'/x]}\ \textbf{L-App}$$

Stop when term is **not an application**, or **leftmost application** is not an **application of an abstraction** (that is, a lambda term)

Example 1

Lazy Evaluation

L-App

$$\frac{}{(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))\ -\beta\to\ ??}\ \textbf{L-App}$$

What does this evaluate to?

Example 1

Lazy Evaluation

$$\frac{}{(\lambda\, z.\, (\lambda\, x.\, x))\ ((\lambda\, y.\, y\ y)\ (\lambda\, y.\, y\ y))\ -\beta\!\!\rightarrow\ ??}\ \textbf{L-App}$$

Subtitute for z

Example 1

Lazy Evaluation

$$\frac{}{(\lambda\, z.\, (\lambda\, x.\, x))\ ((\lambda\, y.\, y\, y)\ (\lambda\, y.\, y\, y)) \ -\beta\!\!\to\ (\lambda\, x.\, x)}\ \textbf{L-App}$$

There is no z in the body

Example 1

Lazy Evaluation

L-App

$$\frac{}{(\lambda\, \mathbf{z}.\ (\lambda\, x.\ x))\ ((\lambda\, \mathbf{y.\ y\ y})\ (\lambda\, \mathbf{y.\ y\ y})) \ -\beta\!\!\rightarrow\ (\lambda\, x.\ x)}$$

We didn't ever need to compute the argument. Which is good here …

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists**

Lazy vs. Eager

Order of Evaluation

Eager Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{E' \rightarrow E''}{(\lambda\ x\ .\ E)\ E' \rightarrow (\lambda\ x\ .\ E)\ E''}\ \textbf{E-App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \rightarrow E[V/x]}\ \textbf{V-App}$$

Order of Evaluation

Eager Evaluation

$$\frac{\mathbf{E} \rightarrow \mathbf{E''}}{\mathbf{E}\ E' \rightarrow \mathbf{E''}\ E'}\ \textbf{App}$$

$$\frac{E' \rightarrow E''}{(\lambda\ x\ .\ E)\ E' \rightarrow (\lambda\ x\ .\ E)\ E''}\ \textbf{E-App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \rightarrow E[V/x]}\ \textbf{V-App}$$

(Eagerly) reduce leftmost applications first

Order of Evaluation

Eager Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{\mathbf{E'} \rightarrow \mathbf{E''}}{(\lambda\ x\ .\ E)\ \mathbf{E'} \rightarrow (\lambda\ x\ .\ E)\ \mathbf{E''}}\ \textbf{E-App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \rightarrow E[V/x]}\ \textbf{V-App}$$

Then (eagerly) reduce argument

Order of Evaluation

Eager Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{E' \rightarrow E''}{(\lambda\ x\ .\ E)\ E' \rightarrow (\lambda\ x\ .\ E)\ E''}\ \textbf{E-App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \rightarrow E[V/x]}\ \textbf{V-App}$$

Once you have abstraction and value, substitute

Example 1

Eager Evaluation

$$\frac{\textbf{??}}{(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))\ -\beta\!\!\to\ \textbf{??}}\ \textbf{E-App}$$

What does this evaluate to?

Example 1

Eager Evaluation

$$\frac{(\lambda\, y.\ y\ y)\ (\lambda\, y.\ y\ y)\ \text{–}\beta\text{–>}\ ??}{(\lambda\, z.\ (\lambda\, x.\ x))\ ((\lambda\, y.\ y\ y)\ (\lambda\, y.\ y\ y))\ \text{–}\beta\text{–>}\ ??}\ \text{E-App}$$

Reduce the argument first …

Example 1

Eager Evaluation

$$\frac{(\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y) \ \text{–}\beta\text{–>}\ (\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y)}{(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y)) \ \text{–}\beta\text{–>}\ ??}\ \textbf{E-App}$$

Uh oh ...

Example 1

Eager Evaluation

$$\frac{(\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y)\ \text{–}\beta\text{–>}\ (\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y)}{(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))\ \text{–}\beta\text{–>}}\ \textbf{E-App}$$

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

Uh...

Example 1

Eager Evaluation

(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))

yo guys

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists**

Lazy vs. Eager

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists** (in general)

Lazy vs. Eager

Order of Evaluation

- **Not all terms reduce** to normal forms
- **Not all reduction strategies** will **produce** a normal form **if one exists** (in general) (for some terms, may not matter)

Lazy vs. Eager

Example 2

Lazy Evaluation

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

??

Example 2

Lazy Evaluation

(λ **x**. **x x**) **((λ y. y y) (λ z. z))** –β–>
??

Substitute for x

Example 2

Lazy Evaluation

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

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

Substitute for x

Example 2

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–>

??

Evaluate leftmost

Example 2

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–>
??

Substitute for y

Example 2

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))

Substitute for y

Example 2

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))

Evaluate leftmost

Example 2

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–>
??

Substitute for z

Example 2

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))

Substitute for z

Example 2

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–>
??

Substitute for z

Example 2

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)

Substitute for z

Example 2

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–>
??

Substitute for y

Example 2

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)

Substitute for y

Example 2

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–>

??

Substitute for z

Example 2

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)

Substitute for z

Example 2

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)

Done

Example 2

Eager Evaluation

(λ x. x x) ((λ y. y y) (λ z. z)) –β–>

??

Example 2

Eager Evaluation

(λ x. x x) **((λ y. y y) (λ z. z))** –β–>
??

Eagerly evaluate argument

Example 2

Eager Evaluation

(λ x. x x) ((λ **y**. **y** **y**) **(λ z. z)**) –β–>

??

Substitute for y

Example 2

Eager Evaluation

(λ x. x x) ((λ **y**. **y y**) **(λ z. z)**) –β–>
(λ x. x x) (**(λ z. z) (λ z. z)**)

Substitute for y

Example 2

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–>
??

Eagerly evaluate argument

Example 2

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–>
??

Substitute for z

Example 2

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)**

Substitute for z

Example 2

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–>
??

Substitute for x

Example 2

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)

Substitute for x

Example 2

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–>
??

Substitute for z

Example 2

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−>
(λ z. z)

Substitute for z

Example 2

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–>
(λ z. z)

Done

Example 2

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)

Done

Example 2

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–>
(λ z. z)

Done

Questions so far?

η (Eta) Reduction/Expansion

- **η Rule:** λ x. f x –η–> f if x not free in f
  - Can be useful in each direction
  - Not valid in OCaml
    - Interacts poorly with side effects
  - **Not** equivalent to (λ x. f) x –> f (inst. of β)
- Example: λ x. (λ y. y) x –η–> λ y. y

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is **Turing Complete**
  - Can express any sequential computation
  - Yes, in that few computation rules
- But it'd suck to use as-is:
  - How to express basic **data**: booleans, integers, etc?
  - How to express **recursion**?
  - What about **constants**? **If-then-else**?
    - "Just" a convenience—can be added as syntactic sugar

Evaluation

Bonus:
Representing Data Structures

First Pass - **Enumeration** Types

- Suppose T is a type with n constructors: $C_1, \ldots, C_n$ (no arguments)
- Represent each constructor 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 ith constructor

Example: Booleans

bool = True | False

True $\rightarrow \lambda x_1. \lambda x_2. x_1 \equiv_\alpha \lambda x. \lambda y. x$

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

How to Represent Booleans

bool = True | False

True $\rightarrow \lambda\, x_1.\ \lambda\, x_2.\ x_1 \equiv_\alpha \lambda\, \mathbf{x}.\ \lambda\, y.\ \mathbf{x}$

False $\rightarrow \lambda\, x_1.\ \lambda\, x_2.\ x_2 \equiv_\alpha \lambda\, x.\ \lambda\, \mathbf{y}.\ \mathbf{y}$

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

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 $(\ldots(e_1\, e_2) \ldots e_n)$

How to Represent Booleans

bool = True | False

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

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

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

- For type T $= C_1 \mid \ldots \mid C_n$
- Write a "match" function:

  match e with
  | C_1 -> x_1
  | …
  | C_n -> x_n

  as:

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

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

Functions over Enumeration Types

- For type $T = C_1 \mid \ldots \mid C_n$
- Write a "match" function:

  match e with
  $\mid C_1 \text{ -> } x_1$
  $\mid \ldots$
  $\mid C_n \text{ -> } x_n$

  as:

  $\lambda\, x_1 \ldots x_n\, e \,.\, e\; x_1 \ldots x_n$
- **Think**: give me what to do in each case and give me a case, and I'll apply that case

match for Booleans

bool = True | False

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

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

match_{bool} = **??**

match for Booleans

bool = True | False

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

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

$match_{bool} = \lambda\, x_1\, x_2\, e\, .\, e\, x_1\, x_2$

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

match for Booleans

bool = True | False

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

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

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

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

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

match for Booleans

bool = True | False

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

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

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

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

(λ x y **b** . **b** x y) x y **True**

match for Booleans

bool = True | False

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

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

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

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

(**True** x y)

match for Booleans

bool = True | False

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

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

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

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

$((\lambda\, x\, y .\, x)\; x\; y)$

match for Booleans

bool = True | False

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

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

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

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

$((\lambda\, \mathbf{x}\, \mathbf{y} .\, \mathbf{x})\, \mathbf{x}\, \mathbf{y})$

match for Booleans

bool = True | False

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

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

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

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

-

match for Booleans

bool = True | False

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

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

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

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

$(\lambda\, x\, y\, \mathbf{b} .\, \mathbf{b}\, x\, y)\, x\, y$ **False**

match for Booleans

bool = True | False

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

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

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

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

$((\lambda\, \mathbf{x}\, \mathbf{y} .\, \mathbf{y})\, \mathbf{x}\, \mathbf{y})$

match for Booleans

bool = True | False

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

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

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

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

y

How to Write Functions over Booleans

if b then x else y =

$match_{bool}$ x y b =

(λ x y b . b x y) x y b =

b x y

How to Write Functions over Booleans

if b then x else y =

$match_{bool}$ x y b =

(λ x y b . b x y) x y b =

b x y

How to Write Functions over Booleans

if b then x else y =

$\text{match}_{\text{bool}}$ x y b =

(λ x y b . b x y) x y b =

b x y

How to Write Functions over Booleans

if b then x else y =

$match_{bool}$ x y b =

(λ x y b . b x y) x y b =

b x y

How to Write Functions over Booleans

if b then x else y =

match_{bool} x y b =

(λ x y b . b x y) x y b =

b x y

if_then_else

≡ λ b x y . (match_{bool} x y b)

= λ b x y . (λ x y b . b x y) x y b

= λ b x y . b x y

Example:

not b

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

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

= (λ x y b . b x y) (λ x y . y) (λ x y . x) b

= b (λ x y. y) (λ x y. x)

not ≡ λ b. b (λ x y. y) (λ x y. x)

Example:

not b

= match b with

| True -> False

| False -> True

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

= (λ x y b . b x y) (λ x y . y) (λ x y . x) b

= b (λ x y. y) (λ x y. x)

not ≡ λ b. b (λ x y. y) (λ x y. x)

Example:

not b

= match b with

| True -> False

| False -> True

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

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

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

$\text{not} \equiv \lambda\ b.\ b\ (\lambda\ x\ y.\ y)\ (\lambda\ x\ y.\ x)$

Example:

not b

= match b with

| True -> False

| False -> True

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

= (λ x y b . b x y) (λ x y . y) (λ x y . x) b

= b (λ x y. y) (λ x y. x)

not ≡ λ b. b (λ x y. y) (λ x y. x)

Example:

not b

= match b with

| True -> False

| False -> True

$\rightarrow$ (match_{bool}) False True b

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

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

not $\equiv \lambda\ b.\ b\ (\lambda\ x\ y.\ y)\ (\lambda\ x\ y.\ x)$

Example:

not b

= match b with

| True -> False

| False -> True

$\rightarrow$ (match_{bool}) False True b

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

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

not $\equiv \lambda\ b.\ b\ (\lambda\ x\ y.\ y)\ (\lambda\ x\ y.\ x)$

Example:

and b1 b2

= **??**

Let's do this together

Example:

```
and b1 b2
 = match b1 with
    | True -> b2
    | False -> False
```

One way to do this

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

$\rightarrow$ $(\text{match}_{\text{bool}})$ b2 False b1   **Using match_bool**

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

$\rightarrow$ $(\text{match}_{\text{bool}})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

Expanding

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

$\rightarrow$ $(\text{match}_{\text{bool}})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

= b1 b2 (λ x y . y) **Reducing**

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

→ $(\text{match}_{\text{bool}})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

= b1 b2 (λ x y . y)

and ≡ λ b1 b2 . b1 b2 (λ x y. y) **Done**

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

→ $(match_{bool})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

= b1 b2 (λ x y . y)

and ≡ λ b1 b2 . b1 b2 (λ x y. y) **Done**

and True True ≡ (λ x y. x) (λ x y. x) (λ x y. y)

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

$\rightarrow$ $(\text{match}_{\text{bool}})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

= b1 b2 (λ x y . y)

and ≡ λ b1 b2 . b1 b2 (λ x y. y) **Done**

and True True ≡ (λ **x** y. **x**) **(λ x y. x)** (λ x y. y)

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

→ $(\text{match}_{\text{bool}})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

= b1 b2 (λ x y . y)

and ≡ λ b1 b2 . b1 b2 (λ x y. y) **Done**

and True True ≡ **(λ x y. x)**

Example:

and b1 b2

= match b1 with

| True -> b2

| False -> False

$\rightarrow$ $(\text{match}_{\text{bool}})$ b2 False b1

= (λ x y b . b x y) b2 (λ x y . y) b1

= b1 b2 (λ x y . y)

and ≡ λ b1 b2 . b1 b2 (λ x y. y) **Done**

and True True ≡ **True**

Try on your own:

or b1 b2

= **??**

More in Appendix

Questions?

Next Class: Axiomatic Semantics

Next Class

- **WA10 due Thursday**
- **MP11 due next Tuesday**
- **Email me about retakes!**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help

Appendix A: More Data Structures

Second Pass - **Union** Types

- Suppose T is a type with n constructors:

  type T =
  | $C_1\ t_{11} \ldots t_{1k}$
  | …
  | $C_n\ t_{n1} \ldots t_{nm}$

- Represent each term as an abstraction:

  $C_i \rightarrow \lambda\ t_{i1} \ldots t_{ij,}\ x_1 \ldots x_n\ .\ x_i\ t_{i1} \ldots t_{ij}$

- **Think**: you need to give each constructor its arguments first

Appendix A

Second Pass - **Union** Types

- Suppose T is a type with n constructors:

  $$\begin{aligned}&\text{type } T =\\ &\mid C_1\ t_{11} \ldots t_{1k}\\ &\mid \ldots\\ &\mid C_n\ t_{n1} \ldots t_{nm}\end{aligned}$$

- Represent each term as an abstraction:

  $$C_i \rightarrow \lambda\, t_{i1} \ldots t_{ij,}\ x_1 \ldots x_n\,.\ x_i\ t_{i1} \ldots t_{ij}$$

- **Think**: you need to give each constructor its arguments first

Appendix A

Example: How to Represent Pairs

Pair has **one constructor** taking **two arguments**:

```
type ('a, 'b) pair =
| (,) 'a 'b
```

(a , b) → λ x . x a b

(,) → λ a b x . x a b

Appendix A

Example: How to Represent Pairs

Pair has **one constructor** taking **two arguments**:

```
type ('a, 'b) pair =
| (,) 'a 'b
```

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

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

Example: How to Represent Pairs

Pair has **one constructor** taking **two arguments**:

type ('a, 'b) pair =

| (,) 'a 'b

(a , b) → λ x . x a b

(,) → λ a b x . x a b

Appendix A

Functions over Union Types

- Write a "match" function:

  match e with
  | C_1 y_1 … y_{m1} -> f_1 y_1 … y_{m1}
  | …
  | C_n y_1 … y_{mn} -> f_n y_1 … y_{mn}

- match_T $\rightarrow \lambda f_1 \ldots f_n\, e.\ e\ f_1 \ldots f_n$
- **Think:** give me a function for each case and give me a case, and I'll apply that case to the appropriate function with the data in that case

Appendix A

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_T $\rightarrow \lambda\ f_1 \ldots f_n\ e.\ e\ f_1 \ldots f_n$
- **Think:** give me a function for each case and give me a case, and I’ll apply that case to the appropriate function with the data in that case

Appendix A

Functions over Union Types

- Write a "match" function:

  match e with

  | C_1 y_1 … y_{m1} -> $\mathbf{f_1}$ y_1 … y_{m1}

  | …

  | C_n y_1 … y_{mn} -> $\mathbf{f_n}$ y_1 … y_{mn}
- match_T → λ $\mathbf{f_1}$ … $\mathbf{f_n}$ e. e $\mathbf{f_1}$…$\mathbf{f_n}$
- **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

Appendix A

Functions over Union Types

- Write a "match" function:

  match e with
  | $C_1\ y_1 \ldots y_{m1}$ -> $\mathbf{f_1}\ y_1 \ldots y_{m1}$
  | …
  | $C_n\ y_1 \ldots y_{mn}$ -> $\mathbf{f_n}\ y_1 \ldots y_{mn}$

- match_T → $\lambda\ \mathbf{f_1} \ldots \mathbf{f_n}\ e.\ e\ \mathbf{f_1} \ldots \mathbf{f_n}$
- **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

Appendix A

Example: Functions over Pairs

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

fst p = match p with (x, y) -> x

fst $\rightarrow \lambda$ p. $match_{pair}$ (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)

Appendix A

Example: Functions over Pairs

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

fst p = match p with (x, y) -> x

$fst \rightarrow \lambda\ p.\ match_{pair}\ (\lambda\ x\ y.\ x)$

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

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

$snd \rightarrow \lambda\ p.\ p\ (\lambda\ x\ y.\ y)$

Example: Functions over Pairs

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

fst p = match p with (x, y) -> x

fst $\to$ 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 $\to$ λ p. p (λ x y. y)

Appendix A

Example: Functions over Pairs

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

fst p = match p with (x, y) -> x

$fst \rightarrow \lambda\ p.\ match_{pair}\ (\lambda\ x\ y.\ x)$

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

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

$snd \rightarrow \lambda\ p.\ p\ (\lambda\ x\ y.\ y)$

Example: Functions over Pairs

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

fst p = match p with (x, y) -> x

$fst \rightarrow \lambda\ p.\ match_{pair}\ (\lambda\ x\ y.\ x)$

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

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

$snd \rightarrow \lambda\ p.\ p\ (\lambda\ x\ y.\ y)$

Example: Functions over Pairs

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

fst p = match p with (x, y) -> x

$fst \rightarrow \lambda\ p.\ match_{pair}\ (\lambda\ x\ y.\ x)$

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

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

$snd \rightarrow \lambda\ p.\ p\ (\lambda\ x\ y.\ y)$

Appendix A

Third Pass - **Recursive** Types

- Suppose T is a type with n constructors:

  type T =
  | $C_1\ t_{11} \ldots t_{1k}$
  | …
  |$C_n\ t_{n1} \ldots t_{nm}$
- Suppose $t_{ih} : T$ (i.e., it is a **recursive reference**)
- In place of a value t_{ih} have a **function** to compute the **recursive value** $r_{ih}\ x_1 \ldots x_n$

  $C_i \rightarrow \lambda\, t_{i1} \ldots r_{ih} \ldots t_{ij}\ x_1 \ldots x_n . x_i\ t_{i1} \ldots (r_{ih}\ x_1 \ldots x_n) \ldots t_{ij}$

Appendix A

Third Pass - **Recursive** Types

- Suppose T is a type with n constructors:

  type T =
  | $C_1\ t_{11} \ldots t_{1k}$
  | …
  | $C_n\ t_{n1} \ldots t_{nm}$
- Suppose $t_{ih} : T$ (i.e., it is a **recursive reference**)
- In place of a value t_{ih} have a **function** to compute the **recursive value** $r_{ih}\ x_1 \ldots x_n$

  $C_i \rightarrow \lambda\ t_{i1} \ldots r_{ih} \ldots t_{ij}\ x_1 \ldots x_n . x_i\ t_{i1} \ldots (r_{ih}\ x_1 \ldots x_n) \ldots t_{ij}$

Appendix A

Example: Natural Numbers

nat =
| 0
| Suc nat

$0 = \lambda\, f\, x.\ x$

$Suc = \lambda\, n\, f\, x.\ f\ (n\ f\ x)$

$Suc\ n = \lambda\, f\, x.\ f\ (n\ f\ x)$

Such representation called **Church Numerals**

Appendix A

Example: Natural Numbers

nat =
| 0
| Suc nat

$0 = \lambda f\, x.\ x$

$Suc = \lambda n\, f\, x.\ f\ (n\ f\ x)$

$Suc\ n = \lambda f\, x.\ f\ (n\ f\ x)$

Such representation called **Church Numerals**

Appendix A

Example: Natural Numbers

nat =
| 0
| Suc nat

0 = λ f x. x

Suc = λ n f x. f (n f x)

Suc n = λ f x. f (n f x)

Such representation called **Church Numerals**

Appendix A

Example: Natural Numbers

nat =
| 0
| Suc nat

$0 = \lambda\, f\, x.\, x$

$Suc = \lambda\, n\, f\, x.\, f\, (n\, f\, x)$

$Suc\ n = \lambda\, f\, x.\, f\, (n\, f\, x)$

Such representation called **Church Numerals**

Appendix A

Example: Some Church Numerals

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

Appendix A

Example: Some Church Numerals

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

Appendix A

Example: Some Church Numerals

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

Appendix A

Example: Some Church Numerals

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

Appendix A

Example: Some Church Numerals

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 n = λ f x. f (… (f x)…) with n applications of f

Appendix A

Example: Some Church Numerals

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 n = λ f x. f (… (f x)…) with n applications of f

Appendix A

Appendix: Recursion

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_T $\rightarrow$ λ f_1 … f_n e. e f_1…f_n
- Match in nonrec. case a degenerate version of fold

Appendix B

Primitive Recursive Functions

- Write a "fold" function

  $\text{fold } f_1 \ldots f_n =$
  $\quad \text{match e with}$
  $\quad | \; C_1 \; y_1 \ldots y_{m1} \text{ -> } f_1 \; y_1 \ldots y_{m1}$
  $\quad | \ldots$
  $\quad | \; Ci \; y_1 \ldots r_{ij} \ldots y_{in} \text{ -> } f_n \; y_1 \ldots (\text{fold } f_1 \ldots f_n \; r_{ij}) \ldots y_{mn}$
  $\quad | \ldots$
  $\quad | \; C_n \; y_1 \ldots y_{mn} \text{ -> } f_n \; y_1 \ldots y_{mn}$

- $\text{fold\_T} \rightarrow \lambda \, f_1 \ldots f_n \, e. \; e \; f_1 \ldots f_n$
- Match in nonrec. case a degenerate version of fold

Appendix B

Primitive Recursion over Nat

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

fold ≡ λ f z n. n f z

is_zero n = fold (λ r. False) True n =

(λ f x. f^{n} x) (λ r. False) True =

((λ r. False) ^{n}) True ≡

if n = 0 then True else False

Appendix B

Primitive Recursion over Nat

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

fold ≡ λ f z n. n f z

is_zero n = fold (λ r. False) True n =

(λ f x. f^n x) (λ r. False) True =

((λ r. False) ^{n}) True ≡

if n = 0 then True else False

Appendix B

Adding Church Numerals

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

$n + m$

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

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

$= \lambda\, f\, x.\, n\, f\, (m\, f\, x)$

$+ \equiv \lambda\, n\, m\, f\, x.\, n\, f\, (m\, f\, x)$

(Subtraction is harder.)

Appendix B

Adding Church Numerals

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

$n + m$

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

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

$= \lambda\, f\, x.\ n\ f\ (m\ f\ x)$

$+ \equiv \lambda\, n\ m\ f\ x.\ n\ f\ (m\ f\ x)$

(Subtraction is harder.)

Appendix B

Multiplying Church Numerals

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

$n * m$

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

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

$= \lambda\, f\, x.\ n\ (m\ f)\ x$

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

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)

Appendix B

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)
```

Appendix B

Recursion

- Want a λ-term Y such that for all terms R,
  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

Appendix B

Recursion

- Want a λ-term Y such that for all terms R, 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

Appendix B

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

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

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

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?).
```

Appendix B

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**

Appendix B

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**

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**

Appendix B

Some Other Combinators

- For your general exposure
- $I = \lambda\, x\, .\, x$
- $K = \lambda\, x\, y\, .\, x$
- $K_* = \lambda\, x\, y\, .\, y$
- $S = \lambda\, x\, y\, z\, .\, x\, z\, (y\, z)$
- See "SKI combinator calculus"