Functional Programming

WS 2016/17

Cezary Kaliszyk (VO+PS)
Bertram Felgenhauer (PS)
Sebastiaan Joosten (PS)
Akihisa Yamada (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 06

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t\ \ t)$$

λ-Calculus

λ-Terms

$$t ::= \overbrace{x}^{\text{Variable}} \mid (\lambda x.t) \mid (t\ t)$$

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid \underbrace{(\lambda x.t)}_{\text{Abstraction}} \mid (t\ \ t)$$

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid \overbrace{(t\ \ t)}^{\text{Application}}$$

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \ \ t)$$

Example

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

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \;\; t)$$

Example

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

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t\ t)$$

Example

$x\ y$	$(x\ y)$	"x applied to y"
$\lambda x.x$		
$\lambda xy.x$		
$\lambda x.x\ x$		
$(\lambda x.x)\ x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \ \ t)$$

Example

$x \ y$	$(x \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	
$\lambda xy.x$		
$\lambda x.x \ x$		
$(\lambda x.x) \ x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \;\, t)$$

Example

$x \; y$	$(x \; y)$	*“x applied to y”*
$\lambda x.x$	$(\lambda x.x)$	*“lambda x dot x”*
$\lambda xy.x$		
$\lambda x.x \; x$		
$(\lambda x.x) \; x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t\ \ t)$$

Example

$x\ \ y$	$(x\ \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x dot x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	
$\lambda x.x\ \ x$		
$(\lambda x.x)\ \ x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \ t)$$

Example

$x \ y$	$(x \ y)$	*“x applied to y”*
$\lambda x.x$	$(\lambda x.x)$	*“lambda x dot x”*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*“lambda x y dot x”*
$\lambda x.x \ x$		
$(\lambda x.x) \ x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \ \ t)$$

Example

$x \ \ y$	$(x \ \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x dot x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y dot x"*
$\lambda x.x \ x$	$(\lambda x.(x \ \ x))$	
$(\lambda x.x) \ x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \ t)$$

Example

$x \ y$	$(x \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x dot x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y dot x"*
$\lambda x.x \ x$	$(\lambda x.(x \ x))$	*"lambda x dot (x applied to x)"*
$(\lambda x.x) \ x$		

λ-Calculus

λ-Terms

$$t ::= \quad x \quad | \, (\lambda x.t) \, | \, (t \ \ t)$$

Example

$x \ y$	$(x \ y)$	*"x applied to y"*
$\lambda x.x$	$(\lambda x.x)$	*"lambda x dot x"*
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	*"lambda x y dot x"*
$\lambda x.x \ x$	$(\lambda x.(x \ x))$	*"lambda x dot (x applied to x)"*
$(\lambda x.x) \ x$	$((\lambda x.x) \ x)$	

λ-Calculus

λ-Terms

$$t ::= \quad x \quad \mid (\lambda x.t) \mid (t \;\; t)$$

Example

$x \;\; y$	$(x \;\; y)$	“x applied to y”
$\lambda x.x$	$(\lambda x.x)$	“lambda x dot x”
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	“lambda x y dot x”
$\lambda x.x \;\; x$	$(\lambda x.(x \;\; x))$	“lambda x dot (x applied to x)”
$(\lambda x.x) \;\; x$	$((\lambda x.x) \;\; x)$	“(lambda x dot x) applied to x”

λ-Calculus (cont'd)

β-Reduction

the term s (β-)reduces to the term t in one step, i.e.,

$$s \rightarrow_\beta t$$

iff there exist context C and terms u, v s.t.

$$s = C[(\lambda x.u)\ v] \qquad \text{and} \qquad t = C[u\{x/v\}]$$

λ-Calculus (cont'd)

β-Reduction

the term s (β-)reduces to the term t in one step, i.e.,

$$\overbrace{s \rightarrow_\beta t}^{(\beta\text{-})\text{step}}$$

iff there exist context C and terms u, v s.t.

$$s = C[(\lambda x.u)\ v] \qquad \text{and} \qquad t = C[u\{x/v\}]$$

λ-Calculus (cont'd)

β-Reduction

the term s (β-)reduces to the term t in one step, i.e.,

$$s \to_\beta t$$

iff there exist context C and terms u, v s.t.

$$s = C[(\lambda x.u)\ v] \qquad \text{and} \qquad t = C[u\{x/v\}]$$

Example

$$K \overset{\text{def}}{=} \lambda xy.x$$

$$I \overset{\text{def}}{=} \lambda x.x$$

$$\Omega \overset{\text{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction,
reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing,

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, dependent types, monads

Overview

- Week 6 - λ Calculus, Evaluation Strategies
    - Summary of Week 5
    - λ-Calculus - Data Types
    - Evaluation Strategies

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$
- if $\overset{\text{def}}{=} \lambda xyz.x\ y\ z$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$
- if $\overset{\text{def}}{=} \lambda xyz.x\ y\ z$

Example

$$\text{if true } t\ e \rightarrow_\beta^+$$

$$\text{if false } t\ e \rightarrow_\beta^+$$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$
- if $\overset{\text{def}}{=} \lambda xyz.x\ y\ z$

Example

$$\text{if true } t\ e \to_\beta^+ \text{true } t\ e$$

$$\text{if false } t\ e \to_\beta^+$$

Booleans and Conditionals

OCaml

- `true`
- `false`
- `if` b `then` t `else` e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$
- if $\overset{\text{def}}{=} \lambda xyz.x\ y\ z$

Example

$$\text{if true } t\ e \rightarrow_\beta^+ \text{true } t\ e \rightarrow_\beta^+ t$$

$$\text{if false } t\ e \rightarrow_\beta^+$$

Booleans and Conditionals

OCaml

- `true`
- `false`
- **`if`** b **`then`** t **`else`** e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$
- if $\overset{\text{def}}{=} \lambda xyz.x\ y\ z$

Example

$$\text{if true } t\ e \to_\beta^+ \text{true } t\ e \to_\beta^+ t$$
$$\text{if false } t\ e \to_\beta^+ \text{false } t\ e$$

Booleans and Conditionals

OCaml

- **`true`**
- **`false`**
- **`if`** b **`then`** t **`else`** e

λ-Calculus

- true $\overset{\text{def}}{=} \lambda xy.x$
- false $\overset{\text{def}}{=} \lambda xy.y$
- if $\overset{\text{def}}{=} \lambda xyz.x\ y\ z$

Example

$$\text{if true } t\ e \rightarrow_\beta^+ \text{true } t\ e \rightarrow_\beta^+ t$$

$$\text{if false } t\ e \rightarrow_\beta^+ \text{false } t\ e \rightarrow_\beta^+ e$$

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

```
0
1
n
( + )
( * )
( ** )
```

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	
`n`	
`( + )`	
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	
`( + )`	
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\mathsf{n}} \overset{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{n} \overset{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\text{add} \overset{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\mathrm{n}} \overset{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\mathrm{add} \overset{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\mathrm{mul} \overset{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\mathrm{n}} \overset{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\mathrm{add} \overset{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\mathrm{mul} \overset{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	$\mathrm{exp} \overset{\text{def}}{=} \lambda mn.n\ m$

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\mathrm{n}} \overset{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\mathrm{add} \overset{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\mathrm{mul} \overset{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	$\mathrm{exp} \overset{\text{def}}{=} \lambda mn.n\ m$

Example

$$\mathrm{add}\ \overline{1}\ \overline{1} \to_\beta^*$$

Natural Numbers

Definition

$$s^0\ t \overset{\text{def}}{=} t \qquad\qquad s^{n+1}\ t \overset{\text{def}}{=} s\ (s^n\ t)$$

OCaml vs. λ-Calculus

`0`	$\overline{0} \overset{\text{def}}{=} \lambda fx.x$
`1`	$\overline{1} \overset{\text{def}}{=} \lambda fx.f\ x$
`n`	$\overline{\mathsf{n}} \overset{\text{def}}{=} \lambda fx.f^n\ x$
`( + )`	$\mathsf{add} \overset{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x)$
`( * )`	$\mathsf{mul} \overset{\text{def}}{=} \lambda mnf.m\ (n\ f)$
`( ** )`	$\mathsf{exp} \overset{\text{def}}{=} \lambda mn.n\ m$

Example

$$\mathsf{add}\ \overline{1}\ \overline{1} \rightarrow_\beta^* \overline{2}$$

Pairs

OCaml vs. λ-Calculus

```
fun x y -> (x,y)
fst
snd
```

Pairs

OCaml vs. λ-Calculus

`fun x y -> (x,y)`	pair $\overset{\text{def}}{=} \lambda xyf.f\ x\ y$
`fst`	
`snd`	

Pairs

OCaml vs. λ-Calculus

`fun` `x y -> (x,y)`	pair $\overset{\text{def}}{=} \lambda xyf.f\ x\ y$
`fst`	fst $\overset{\text{def}}{=} \lambda p.p$ true
`snd`	

Pairs

OCaml vs. λ-Calculus

`fun x y -> (x,y)`	$\mathsf{pair} \overset{\text{def}}{=} \lambda xyf.f\ x\ y$
`fst`	$\mathsf{fst} \overset{\text{def}}{=} \lambda p.p\ \mathsf{true}$
`snd`	$\mathsf{snd} \overset{\text{def}}{=} \lambda p.p\ \mathsf{false}$

Pairs

OCaml vs. λ-Calculus

`fun x y -> (x,y)`	pair $\overset{\text{def}}{=} \lambda xyf.f\ x\ y$
`fst`	fst $\overset{\text{def}}{=} \lambda p.p$ true
`snd`	snd $\overset{\text{def}}{=} \lambda p.p$ false

Example

$$\mathsf{fst}\ (\mathsf{pair}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}}) \rightarrow_{\beta}^{*}$$

Pairs

OCaml vs. λ-Calculus

`fun x y -> (x,y)`	$\mathsf{pair} \overset{\text{def}}{=} \lambda xyf.f\ x\ y$
`fst`	$\mathsf{fst} \overset{\text{def}}{=} \lambda p.p\ \mathsf{true}$
`snd`	$\mathsf{snd} \overset{\text{def}}{=} \lambda p.p\ \mathsf{false}$

Example

$$\mathsf{fst}\ (\mathsf{pair}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}}) \to_\beta^* \overline{\mathsf{m}}$$

Lists

OCaml vs. λ-Calculus

```
::
hd
tl
[]
fun x -> x = []
```

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\text{def}}{=} \lambda xy.\ \ \mathsf{pair}\ x\ y$
`hd`	
`tl`	
`[]`	
`fun x -> x = []`	

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\mathsf{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	
`tl`	
`[]`	
`fun x -> x = []`	

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\text{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	$\mathsf{hd} \overset{\text{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$
`tl`	
`[]`	
`fun x -> x = []`	

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\text{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	$\mathsf{hd} \overset{\text{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$
`tl`	$\mathsf{tl} \overset{\text{def}}{=} \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z)$
`[]`	
`fun x -> x = []`	

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\text{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	$\mathsf{hd} \overset{\text{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$
`tl`	$\mathsf{tl} \overset{\text{def}}{=} \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z)$
`[]`	$\mathsf{nil} \overset{\text{def}}{=} \lambda x.x$
`fun x -> x = []`	

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\mathsf{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	$\mathsf{hd} \overset{\mathsf{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$
`tl`	$\mathsf{tl} \overset{\mathsf{def}}{=} \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z)$
`[]`	$\mathsf{nil} \overset{\mathsf{def}}{=} \lambda x.x$
`fun x -> x = []`	$\mathsf{null} \overset{\mathsf{def}}{=} \mathsf{fst}$

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\mathsf{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	$\mathsf{hd} \overset{\mathsf{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$
`tl`	$\mathsf{tl} \overset{\mathsf{def}}{=} \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z)$
`[]`	$\mathsf{nil} \overset{\mathsf{def}}{=} \lambda x.x$
`fun x -> x = []`	$\mathsf{null} \overset{\mathsf{def}}{=} \mathsf{fst}$

Example

$$\mathsf{null}\ \mathsf{nil} \to_\beta^*$$

Lists

OCaml vs. λ-Calculus

`::`	$\mathsf{cons} \overset{\text{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$
`hd`	$\mathsf{hd} \overset{\text{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$
`tl`	$\mathsf{tl} \overset{\text{def}}{=} \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z)$
`[]`	$\mathsf{nil} \overset{\text{def}}{=} \lambda x.x$
`fun x -> x = []`	$\mathsf{null} \overset{\text{def}}{=} \mathsf{fst}$

Example

$$\mathsf{null}\ \mathsf{nil} \to_{\beta}^{*} \mathsf{true}$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                              else 1 + length(tl x)
```

λ-Calculus

$$\text{length} \overset{\text{def}}{=} \lambda x.\text{if } (\text{null } x)\ \overline{0}\ (\text{add } \overline{1}\ (\text{length } (\text{tl } x)))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                                 else 1 + length(tl x)
```

λ-Calculus

$$\mathsf{length} \overset{\text{def}}{=} \lambda x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (\mathsf{length}\ (\mathsf{tl}\ x)))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                              else 1 + length(tl x)
```

λ-Calculus

$$\mathsf{length} \overset{\text{def}}{=} \lambda f x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x)))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                                else 1 + length(tl x)
```

λ-Calculus

$$\mathsf{length} \overset{\text{def}}{=} \mathsf{Y}\ (\lambda f x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x))))$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                                 else 1 + length(tl x)
```

λ-Calculus

$$\text{length} \overset{\text{def}}{=} \mathsf{Y}\ (\lambda f x.\text{if}\ (\text{null}\ x)\ \overline{0}\ (\text{add}\ \overline{1}\ (f\ (\text{tl}\ x))))$$

Definition (Y-combinator)

$$\mathsf{Y} \overset{\text{def}}{=} \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Y has fixed point property, i.e., for all $t \in \mathcal{T}(\mathcal{V})$

$$\mathsf{Y}\ t \leftrightarrow^* t\ (\mathsf{Y}\ t)$$

Recursion

OCaml

```
let rec length x = if x = [] then 0
                               else 1 + length(tl x)
```

λ-Calculus

$$\mathsf{length} \overset{\text{def}}{=} \mathsf{Y}\ (\lambda f x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x))))$$

Definition (Y-combinator)

$$\mathsf{Y} \overset{\text{def}}{=} \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Y has fixed point property, i.e., for all $t \in \mathcal{T}(\mathcal{V})$

$$\mathsf{Y}\ t \leftrightarrow^* t\ (\mathsf{Y}\ t)$$

Overview

- Week 6 - λ Calculus, Evaluation Strategies
  - Summary of Week 5
  - λ-Calculus - Data Types
  - Evaluation Strategies

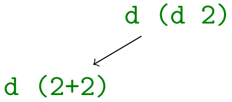
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows

```
d (d 2)
```

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows

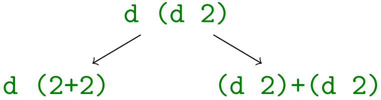


`d (d 2)`

`d (2+2)`

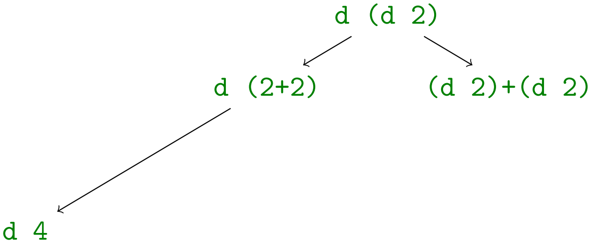
Example

- consider **`let`** `d x = x + x`
- the term `d (d 2)` can be evaluated as follows



Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

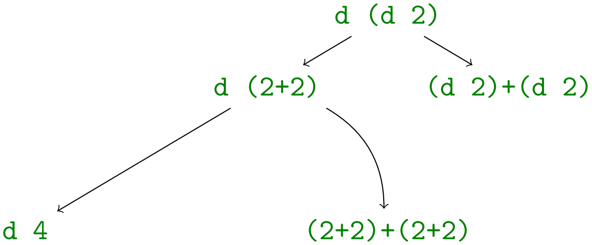
`d (2+2)`

`(d 2)+(d 2)`

`d 4`

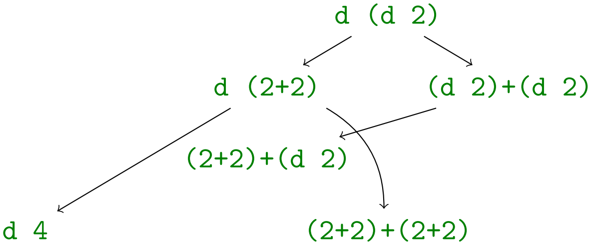
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



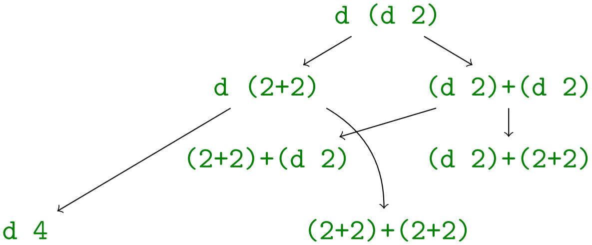
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



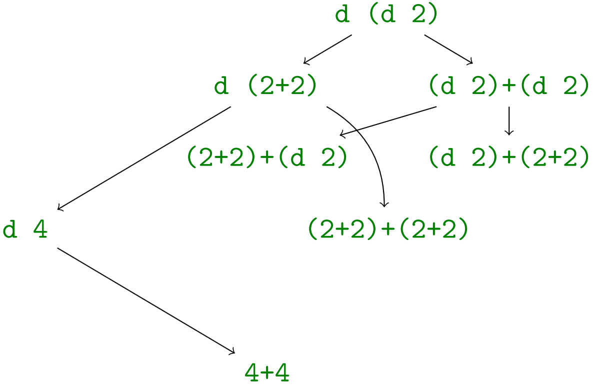
Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

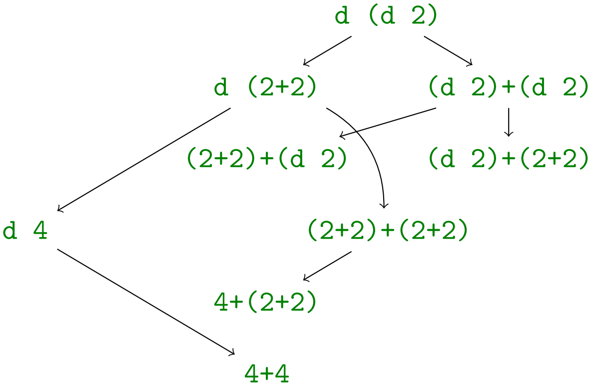
`(2+2)+(d 2)` `(d 2)+(2+2)`

`d 4` `(2+2)+(2+2)`

`4+4`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)`

`(d 2)+(d 2)`

`(2+2)+(d 2)`

`(d 2)+(2+2)`

`d 4`

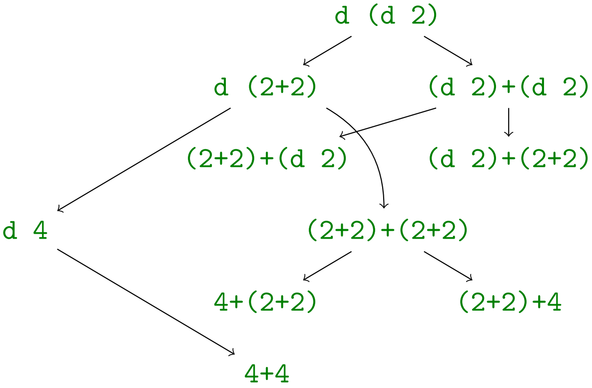
`(2+2)+(2+2)`

`4+(2+2)`

`4+4`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

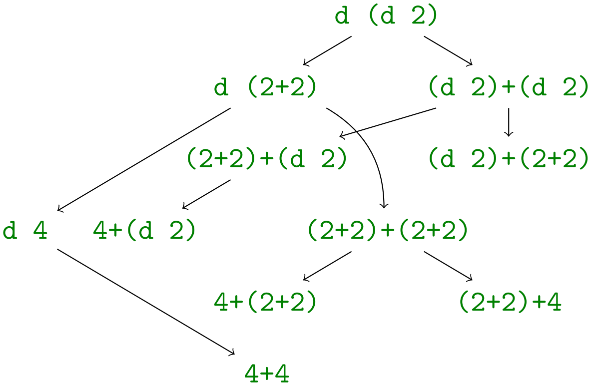
`d 4` `(2+2)+(2+2)`

`4+(2+2)` `(2+2)+4`

`4+4`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)`

`(d 2)+(d 2)`

`(2+2)+(d 2)`

`(d 2)+(2+2)`

`d 4`

`4+(d 2)`

`(2+2)+(2+2)`

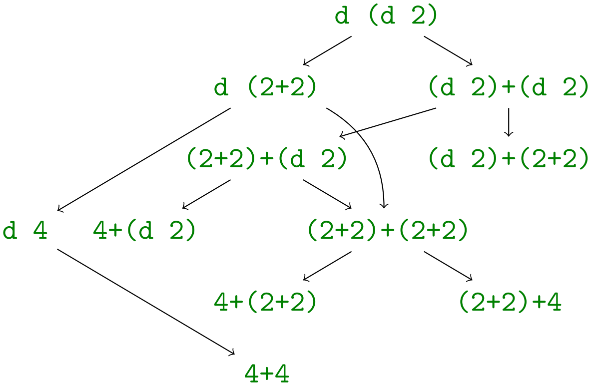
`4+(2+2)`

`(2+2)+4`

`4+4`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



d (d 2)

d (2+2)

(d 2)+(d 2)

(2+2)+(d 2)

(d 2)+(2+2)

d 4

4+(d 2)

(2+2)+(2+2)

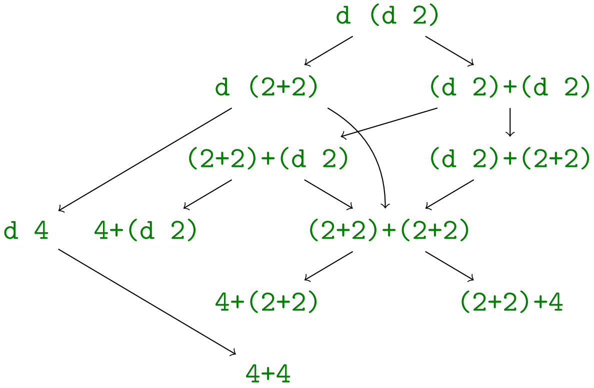
4+(2+2)

(2+2)+4

4+4

Example

- consider **`let`** `d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

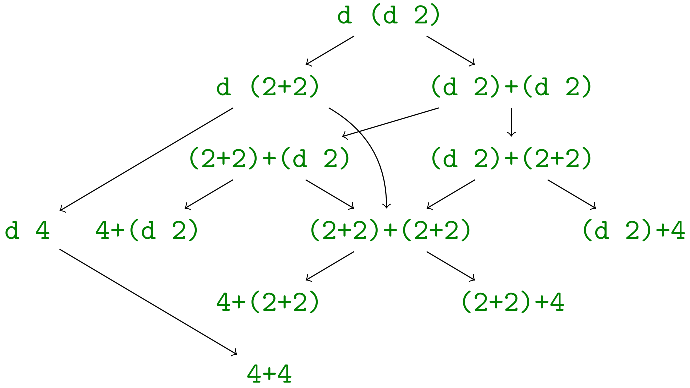
`d 4` `4+(d 2)` `(2+2)+(2+2)`

`4+(2+2)` `(2+2)+4`

`4+4`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

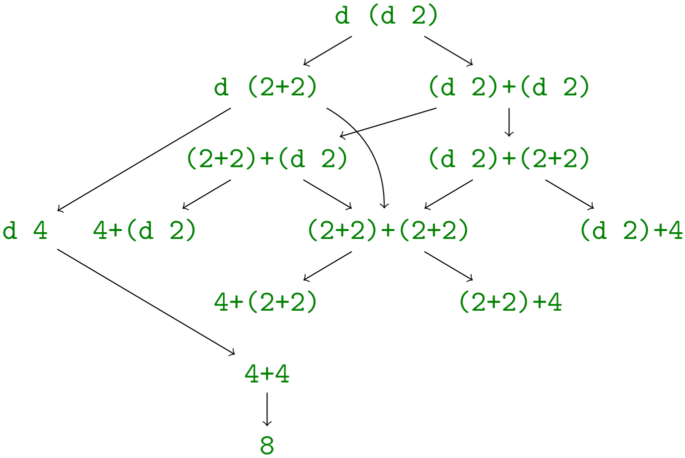
`d 4` `4+(d 2)` `(2+2)+(2+2)` `(d 2)+4`

`4+(2+2)` `(2+2)+4`

`4+4`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

`d 4` `4+(d 2)` `(2+2)+(2+2)` `(d 2)+4`

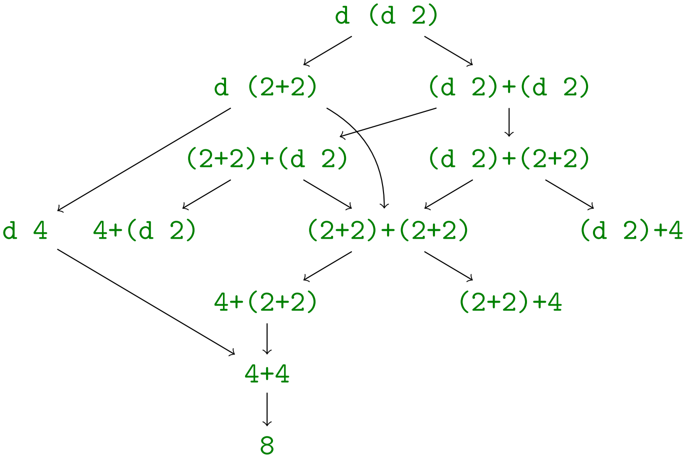
`4+(2+2)` `(2+2)+4`

`4+4`

`8`

Example

- consider **`let`** `d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

`d 4` `4+(d 2)` `(2+2)+(2+2)` `(d 2)+4`

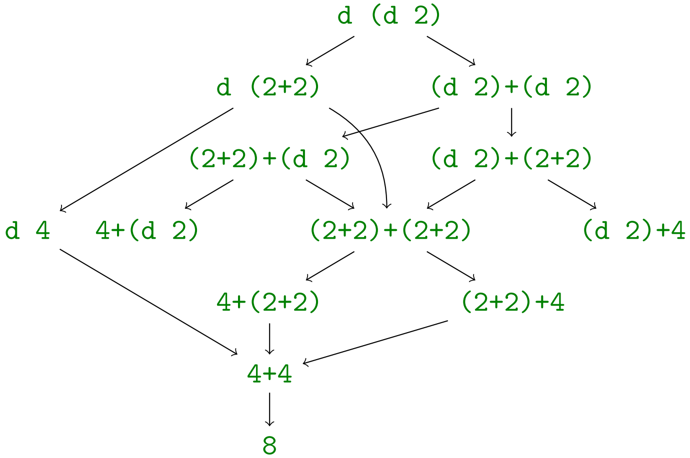
`4+(2+2)` `(2+2)+4`

`4+4`

`8`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

`d 4` `4+(d 2)` `(2+2)+(2+2)` `(d 2)+4`

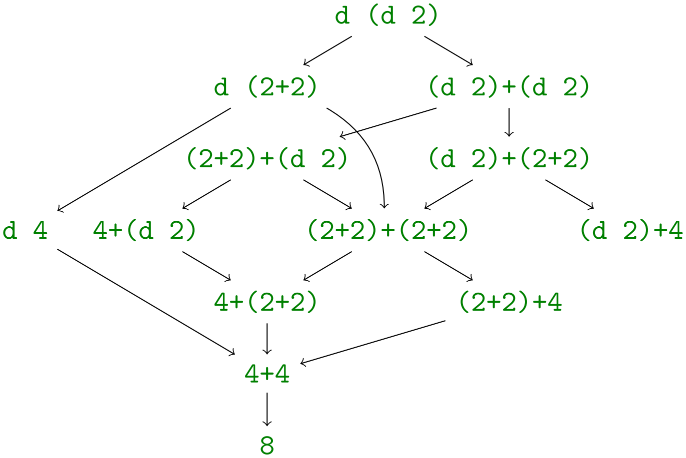
`4+(2+2)` `(2+2)+4`

`4+4`

`8`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows



d (d 2)

d (2+2)

(d 2)+(d 2)

(2+2)+(d 2)

(d 2)+(2+2)

d 4

4+(d 2)

(2+2)+(2+2)

(d 2)+4

4+(2+2)

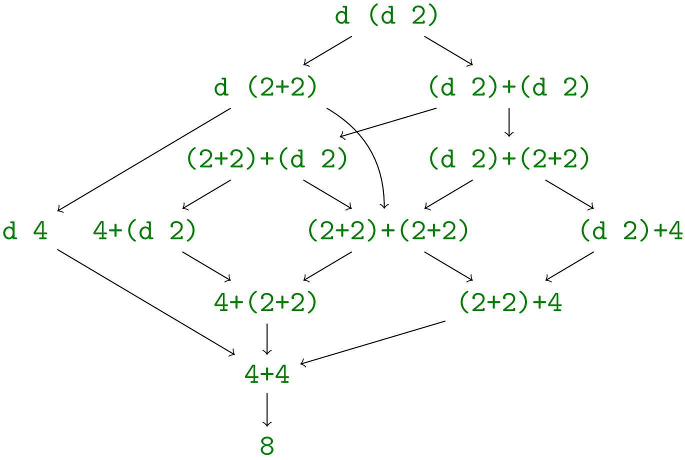
(2+2)+4

4+4

8

Example

- consider **`let`** `d x = x + x`
- the term `d (d 2)` can be evaluated as follows



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

`d 4` `4+(d 2)` `(2+2)+(2+2)` `(d 2)+4`

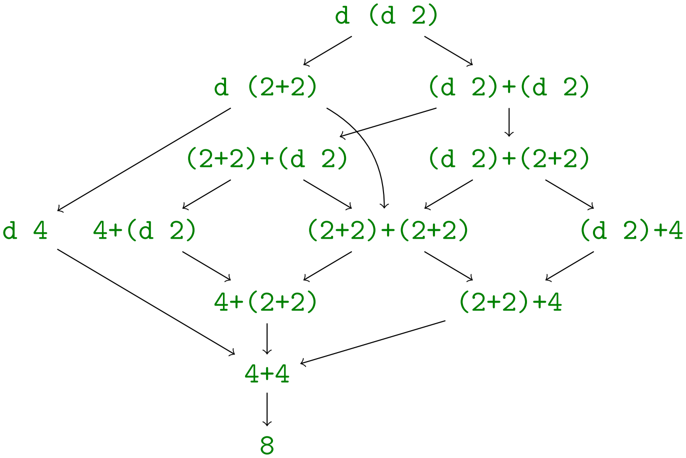
`4+(2+2)` `(2+2)+4`

`4+4`

`8`

Example

- consider `let d x = x + x`
- the term `d (d 2)` can be evaluated as follows (9 possibilities)



`d (d 2)`

`d (2+2)` `(d 2)+(d 2)`

`(2+2)+(d 2)` `(d 2)+(2+2)`

`d 4` `4+(d 2)` `(2+2)+(2+2)` `(d 2)+4`

`4+(2+2)` `(2+2)+4`

`4+4`

`8`

Strategies

Strategy

- fixes evaluation order
- examples: call-by-value and call-by-name

Example

```
let d x = x + x
```

- call-by-value:

$$\begin{aligned} \texttt{d (d 2)} &\rightarrow \texttt{d (2+2)} \\ &\rightarrow \texttt{d 4} \\ &\rightarrow \texttt{4 + 4} \\ &\rightarrow \texttt{8} \end{aligned}$$

- call-by-name:

$$\begin{aligned} \texttt{d (d 2)} &\rightarrow \texttt{(d 2)+(d 2)} \\ &\rightarrow \texttt{(2+2)+(d 2)} \\ &\rightarrow \texttt{4+(d 2)} \\ &\rightarrow \texttt{4+(2+2)} \\ &\rightarrow \texttt{4+4} \\ &\rightarrow \texttt{8} \end{aligned}$$

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is innermost if it does not contain a redex as proper subterm, i.e.,

$$\nexists s \in \mathcal{S}\mathrm{ub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is innermost if it does not contain a redex as proper subterm, i.e.,

$$\nexists s \in \mathcal{S}\mathrm{ub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

Example

Consider $t = \underline{(\lambda x.\underline{(\lambda y.y)\ x})\ z}$

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is innermost if it does not contain a redex as proper subterm, i.e.,

$$\nexists s \in \mathcal{S}\text{ub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

Example

Consider $t = \underline{(\lambda x.\underline{(\lambda y.y)\ x})\ z}$

- $(\lambda y.y)\ x$ is innermost redex

(Leftmost) Innermost Reduction

- always reduce (leftmost) innermost redex

Definition

redex t of term u is innermost if it does not contain a redex as proper subterm, i.e.,

$$\nexists s \in \mathcal{S}\mathrm{ub}(t) \text{ s.t. } s \neq t \text{ and } s \text{ is a redex}$$

Example

Consider $t = \underline{(\lambda x.\underline{(\lambda y.y)\ x})\ z}$

- $(\lambda y.y)\ x$ is innermost redex
- $(\lambda x.(\lambda y.y)\ x)\ z$ is redex, but not innermost

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is outermost if it is not a proper subterm of some other redex in u, i.e.,

$$\nexists s \in \mathcal{S}\text{ub}(u) \text{ s.t. } s \text{ is a redex and } t \in \mathcal{S}\text{ub}(s) \text{ and } s \neq t$$

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is outermost if it is not a proper subterm of some other redex in u, i.e.,

$$\nexists s \in \mathcal{S}\mathrm{ub}(u) \text{ s.t. } s \text{ is a redex and } t \in \mathcal{S}\mathrm{ub}(s) \text{ and } s \neq t$$

Example

Consider $t = \underline{(\lambda x.\underline{(\lambda y.y)}\ x)\ z}$

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is outermost if it is not a proper subterm of some other redex in u, i.e.,

$$\nexists s \in \mathcal{S}\text{ub}(u) \text{ s.t. } s \text{ is a redex and } t \in \mathcal{S}\text{ub}(s) \text{ and } s \neq t$$

Example

Consider $t = \underline{(\lambda x.\underline{(\lambda y.y)\ x})\ z}$

- $(\lambda x.(\lambda y.y)\ x)\ z$ is outermost redex

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex

Definition

redex t of term u is outermost if it is not a proper subterm of some other redex in u, i.e.,

$$\nexists s \in \mathcal{S}\mathrm{ub}(u) \text{ s.t. } s \text{ is a redex and } t \in \mathcal{S}\mathrm{ub}(s) \text{ and } s \neq t$$

Example

Consider $t = \underline{(\lambda x.\underline{(\lambda y.y)\ x})\ z}$

- $(\lambda x.(\lambda y.y)\ x)\ z$ is outermost redex
- $(\lambda y.y)\ x$ is redex, but not outermost

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form ($WHNF$) iff

$$t \neq u\ v$$

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form (*WHNF*) iff

$$t \neq u\ v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form ($WHNF$) iff

$$t \neq u\ v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form (*WHNF*) iff

$$t \neq u\ v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form (*WHNF*) iff

$$t \neq u\ v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form (*WHNF*) iff

$$t \neq u\ v$$

Example (WHNF)

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

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation, e.g., OCaml
- slight modification: only reduce terms that are not in WHNF

Definition (Weak head normal form)

term t is in weak head normal form (*WHNF*) iff

$$t \neq u \ v$$

Example (WHNF)

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

Call-by-Name

- use outermost reduction
- corresponds to lazy evaluation (without memoization), e.g., Haskell
- slight modification: only reduce terms that are not in WHNF