universität innsbruck

Interactive Theorem Proving

Lecture & Exercises Week 4

Cezary Kaliszyk

Summary

Previous Lecture

- HOL exercises
- (untyped) λ-calculus

Today

- λ-calculus vs functional programming
- Typing

β-Reduction (reminder)

Definition (β-step)

if exist context C and terms s, u, and v such that

$$s = C[(\lambda x.u)\ v]$$

then

$$s \quad \rightarrow_\beta \quad C[u\{x/v\}]$$

is a β-step with redex $(\lambda x.u)\ v$ and contractum $u\{x/v\}$

β-Reduction (reminder)

Definition (β-step)

if exist context C and terms s, u, and v such that

$$s = C[(\lambda x.u)\ v]$$

then

$$s \quad \to_\beta \quad C[u\{x/v\}]$$

is a β-step with redex $(\lambda x.u)\ v$ and contractum $u\{x/v\}$

β-Reduction (reminder)

Definition (β-step)

if exist context C and terms s, u, and v such that

$$s = C[(\lambda x.u)\ v]$$

then

$$s \quad \to_\beta \quad C[u\{x/v\}]$$

is a β-step with redex $(\lambda x.u)\ v$ and contractum $u\{x/v\}$

β-Reduction (reminder)

Definition (β-step)

if exist context C and terms s, u, and v such that

$$s = C[(\lambda x.u)\ v]$$

then

$$s \quad \to_\beta \quad C[u\{x/v\}]$$

is a β-step with redex $(\lambda x.u)\ v$ and contractum $u\{x/v\}$

β-Reduction (reminder)

Definition (β-step)

if exist context C and terms s, u, and v such that

$$s = C[(\lambda x.u)\ v]$$

then

$$s \quad \to_\beta \quad C[u\{x/v\}]$$

is a β-step with redex $(\lambda x.u)\ v$ and contractum $u\{x/v\}$

- $s \to_\beta^+ t$ denotes sequence $s = t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_n = t$ with $n > 0$

β-Reduction (reminder)

Definition (β-step)

if exist context C and terms s, u, and v such that

$$s = C[(\lambda x.u)\ v]$$

then

$$s \quad \to_\beta \quad C[u\{x/v\}]$$

is a β-step with redex $(\lambda x.u)\ v$ and contractum $u\{x/v\}$

- $s \to_\beta^+ t$ denotes sequence $s = t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_n = t$ with $n > 0$
- $s \to_\beta^* t$ is sequence with $n \geq 0$ (s β-reduces to t)

Exercises

- Beta-reduce the term $(\lambda x.\lambda y.y\ x)\ (\lambda x.\lambda y.x\ (x\ y))\ (\lambda x.\lambda y.x\ (x\ y))$ as many times as possible (note, minimally different!)
- (LATER) Implement a minimal λ-calculus together with a beta-reduction step

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$K_*\ \Omega$

$K_*\ \Omega$

$I_2\ I_2$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$K_*\ \Omega \rightarrow_\beta K_*\ \Omega$

$K_*\ \Omega$

$I_2\ I_2$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \to_\beta K_*\ \Omega \to_\beta \cdots$$
$$K_*\ \Omega$$
$$I_2\ I_2$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2 = (\lambda xy.x\ y)\ (\lambda xy.x\ y)$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2 = (\lambda xy.x\ y)\ (\lambda xy.x\ y) \rightarrow_\beta \lambda y.(\lambda xy.x\ y)\ y$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2 = (\lambda xy.x\ y)\ (\lambda xy.x\ y) \rightarrow_\beta \lambda y.(\lambda xy.x\ y)\ y \equiv \lambda y.(\lambda xy'.x\ y')\ y$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2 = (\lambda xy.x\ y)\ (\lambda xy.x\ y) \rightarrow_\beta \lambda y.(\lambda xy.x\ y)\ y \equiv \lambda y.(\lambda xy'.x\ y')\ y$$
$$\rightarrow_\beta \lambda y.(\lambda y'.y\ y')$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2 = (\lambda xy.x\ y)\ (\lambda xy.x\ y) \rightarrow_\beta \lambda y.(\lambda xy.x\ y)\ y \equiv \lambda y.(\lambda xy'.x\ y')\ y$$
$$\rightarrow_\beta \lambda y.(\lambda y'.y\ y') = \lambda yy'.y\ y'$$

Example

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$
$$K_* = \lambda xy.y$$
$$I_2 = \lambda xy.x\ y$$

$$K_*\ \Omega \rightarrow_\beta K_*\ \Omega \rightarrow_\beta \cdots$$
$$K_*\ \Omega \rightarrow_\beta \lambda y.y$$
$$I_2\ I_2 = (\lambda xy.x\ y)\ (\lambda xy.x\ y) \rightarrow_\beta \lambda y.(\lambda xy.x\ y)\ y \equiv \lambda y.(\lambda xy'.x\ y')\ y$$
$$\rightarrow_\beta \lambda y.(\lambda y'.y\ y') = \lambda yy'.y\ y' \equiv I_2$$

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

Definition (Normal form)

$t \in \mathcal{T}(\mathcal{V})$ is in normal form (NF) if no β-step possible

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

Definition (Normal form)

$t \in \mathcal{T}(\mathcal{V})$ is in normal form (NF) if no β-step possible

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

Definition (Normal form)

$t \in \mathcal{T}(\mathcal{V})$ is in normal form (NF) if no β-step possible

Example

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

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

Definition (Normal form)

$t \in \mathcal{T}(\mathcal{V})$ is in normal form (NF) if no β-step possible

Example

$$\begin{array}{rl} \lambda x.x & \text{NF} \\ (\lambda x.x)\ y & \end{array}$$

What Are the Results of Computations?

Idea

- only terms in λ-calculus
- express functions and values through λ-terms

Definition (Normal form)

$t \in \mathcal{T}(\mathcal{V})$ is in normal form (NF) if no β-step possible

Example

$$\begin{array}{rl} \lambda x.x & \text{NF} \\ (\lambda x.x)\; y & \text{not NF} \end{array}$$

Booleans and Conditionals

OCaml

- true
- false
- $\mathrm{if}\ b\ \mathrm{then}\ t\ \mathrm{else}\ e$

Booleans and Conditionals

OCaml

- true
- false
- $\mathrm{if}\ b\ \mathrm{then}\ t\ \mathrm{else}\ e$

λ-Calculus

Booleans and Conditionals

OCaml

- $\mathtt{true}$
- $\mathtt{false}$
- $\mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ e$

λ-Calculus

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

Booleans and Conditionals

OCaml

- $\mathtt{true}$
- $\mathtt{false}$
- $\mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ e$

λ-Calculus

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

Booleans and Conditionals

OCaml

- $\mathtt{true}$
- $\mathtt{false}$
- $\mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ e$

λ-Calculus

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

Booleans and Conditionals

OCaml

- $\texttt{true}$
- $\texttt{false}$
- $\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ e$

λ-Calculus

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

Example

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

Booleans and Conditionals

OCaml

- $\texttt{true}$
- $\texttt{false}$
- $\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ e$

λ-Calculus

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

Example

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

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

Booleans and Conditionals

OCaml

- $\mathtt{true}$
- $\mathtt{false}$
- $\mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ e$

λ-Calculus

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

Example

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

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

Booleans and Conditionals

OCaml

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

λ-Calculus

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

Example

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

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

Booleans and Conditionals

OCaml

- $\texttt{true}$
- $\texttt{false}$
- $\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ e$

λ-Calculus

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

Example

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

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

Natural Numbers (Church Numerals)

Definition

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

OCaml vs. λ-Calculus

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

Natural Numbers (Church Numerals)

Definition

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

OCaml vs. λ-Calculus

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

Natural Numbers (Church Numerals)

Definition

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

OCaml vs. λ-Calculus

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

Natural Numbers (Church Numerals)

Definition

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

OCaml vs. λ-Calculus

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

Natural Numbers (Church Numerals)

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

Natural Numbers (Church Numerals)

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 (Church Numerals)

Definition

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

OCaml vs. λ-Calculus

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

Natural Numbers (Church Numerals)

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

Example

$$\text{add}\ \overline{1}\ \overline{1} \rightarrow_\beta^*$$

Natural Numbers (Church Numerals)

Definition

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

OCaml vs. λ-Calculus

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

Example

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

Pairs

OCaml vs. λ-Calculus

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

Pairs

OCaml vs. λ-Calculus

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

Pairs

OCaml vs. λ-Calculus

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

Pairs

OCaml vs. λ-Calculus

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

Pairs

OCaml vs. λ-Calculus

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

Example

$$\text{fst}\ (\text{pair}\ \overline{\text{m}}\ \overline{\text{n}}) \rightarrow^{*}_{\beta}$$

Pairs

OCaml vs. λ-Calculus

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

Example

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Lists

OCaml vs. λ-Calculus

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

Example

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

Lists

OCaml vs. λ-Calculus

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

Example

$$\text{null nil} \rightarrow_{\beta}^{*} \text{true}$$

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Recursion

OCaml

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

λ-Calculus

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

Definition (Y-combinator)

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

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

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

Definition (Y-combinator)

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

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

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

Example

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

```
d (d 2)
```

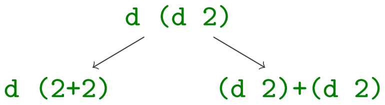
Example

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

```
d (d 2)
  ↙
d (2+2)
```

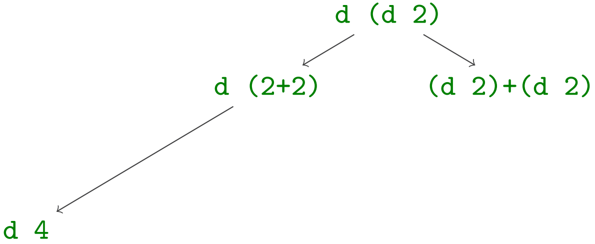
Example

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



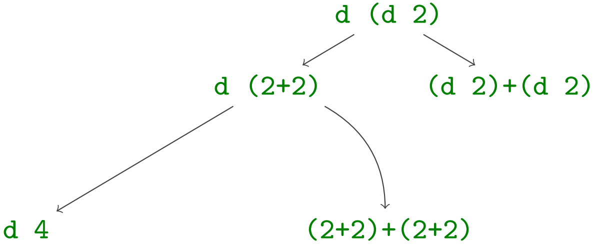
Example

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



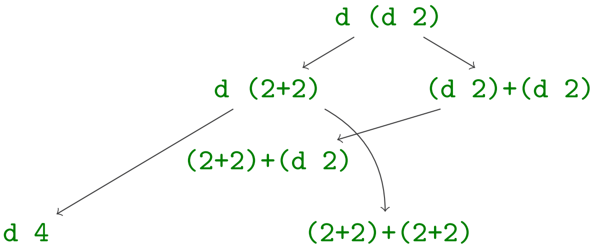
Example

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



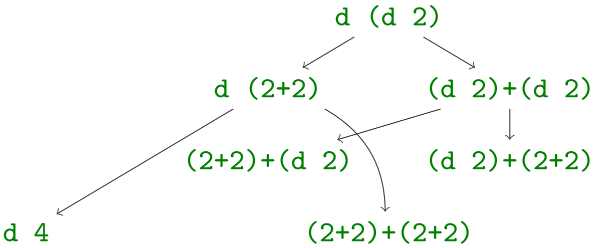
Example

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



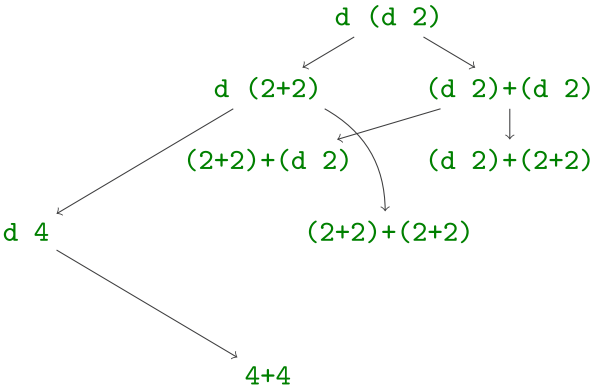
Example

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



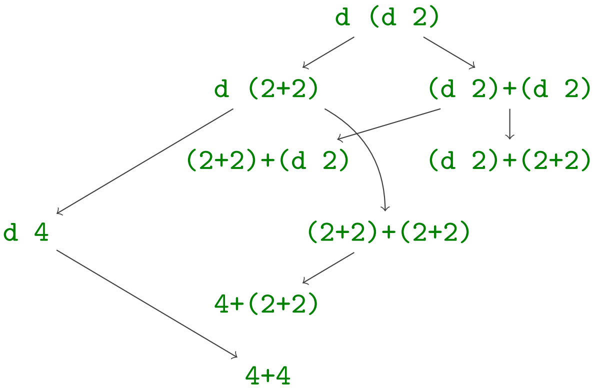
Example

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



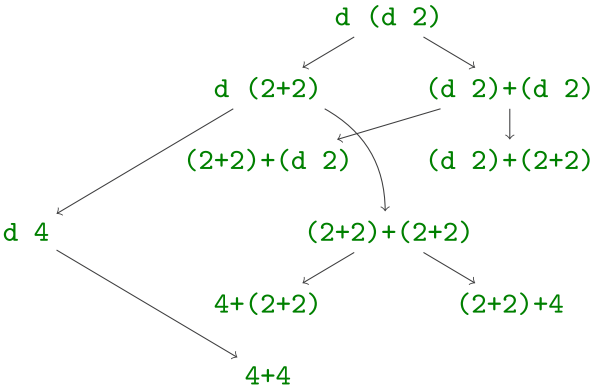
Example

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



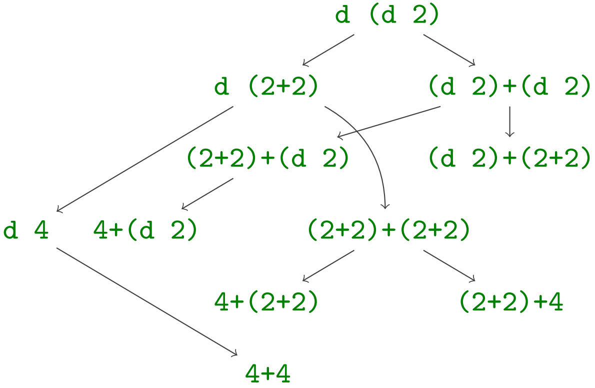
Example

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



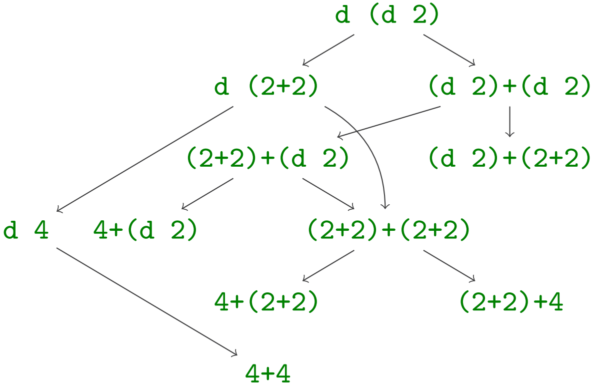
Example

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



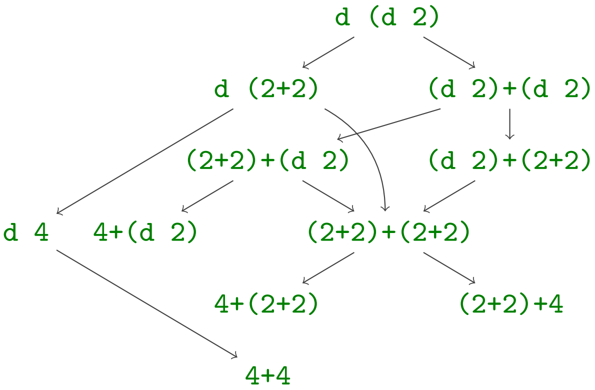
Example

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



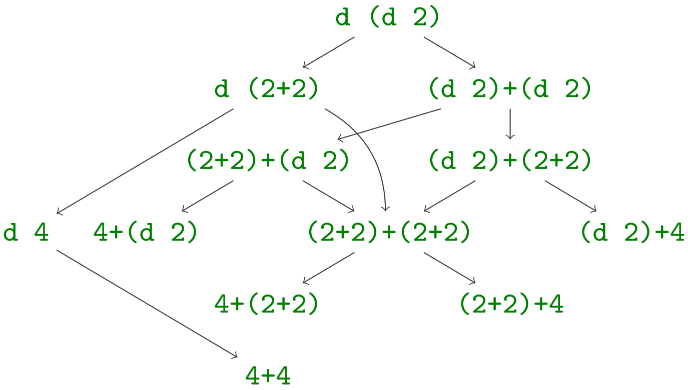
Example

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



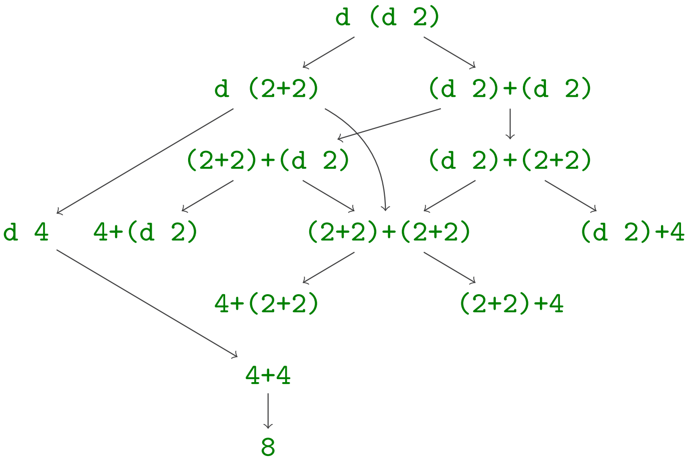
Example

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



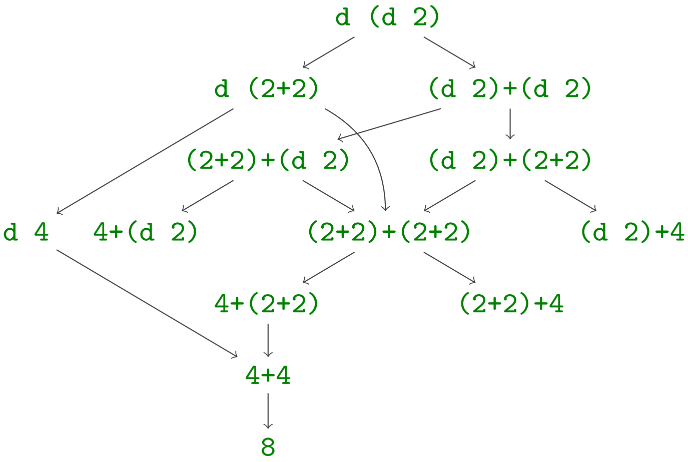
Example

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



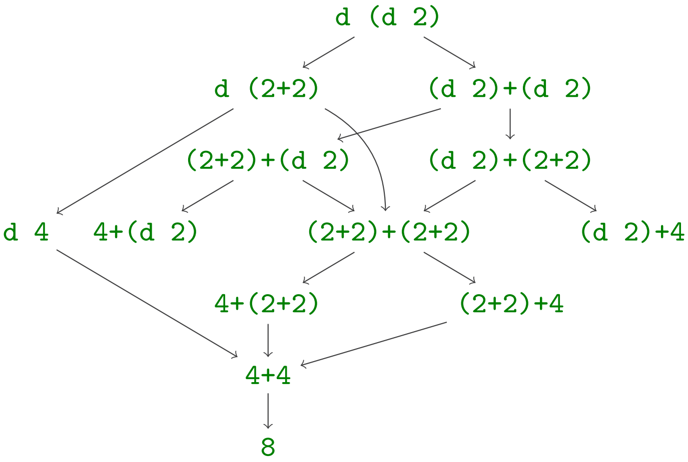
Example

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



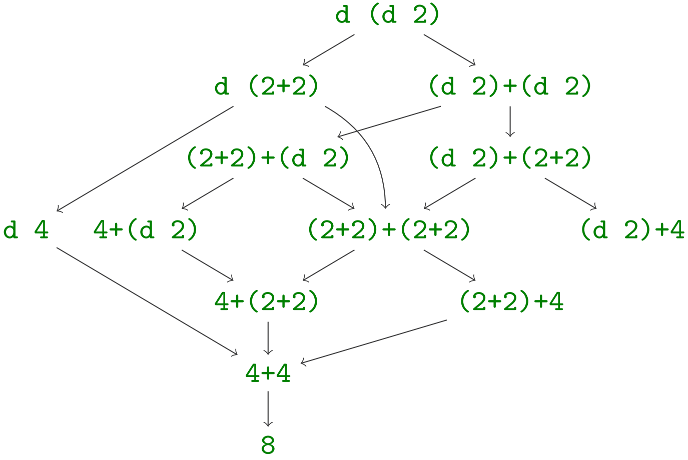
Example

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



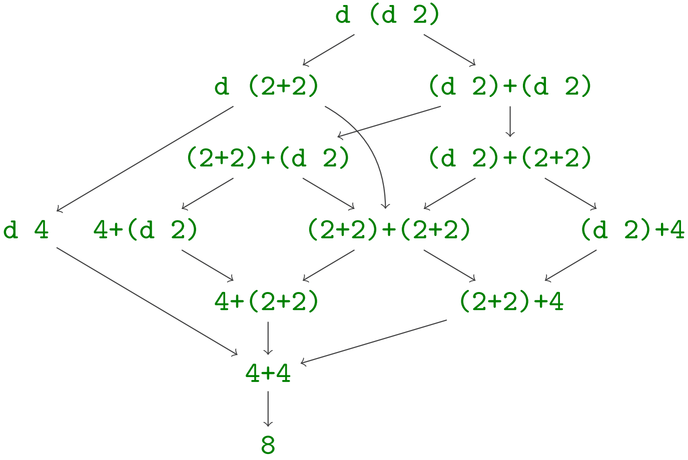
Example

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



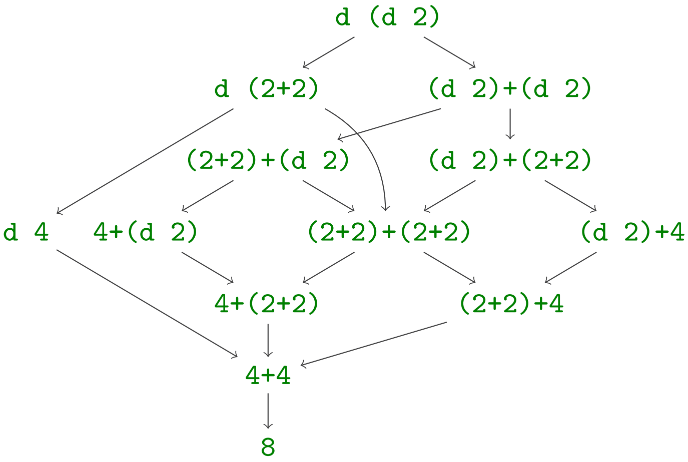
Example

- 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 (9 possibilities)



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.,

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

Example

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

(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.,

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

Example

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

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

Calling

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 (not applications)

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

Calling

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 (not applications)

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

Which one is better?

Exercise

Does your minimal interpreter do inttermost or outermost? Add the other one!

Core ML

Definition (Expressions)

$$e ::= x \mid e\ e \mid \lambda x.e \mid \quad c \quad \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

Core ML

Definition (Expressions)

$$e ::= \overbrace{x \mid e\ e \mid \lambda x.e}^{\lambda\text{-Calculus}} \mid \quad c \quad \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

Core ML

Definition (Expressions)

$$e ::= x \mid e\ e \mid \lambda x.e \mid \underbrace{c}_{\text{primitives/constants}} \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

Core ML

Definition (Expressions)

$$e ::= x \mid e\ e \mid \lambda x.e \mid \quad c \quad \mid \underbrace{\textbf{let}\ x = e\ \textbf{in}\ e}_{\text{let binding}} \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

Core ML

Definition (Expressions)

$$e ::= x \mid e\ e \mid \lambda x.e \mid \quad c \quad \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \underbrace{\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e}_{\text{conditional}}$$

Core ML

Definition (Expressions)

$$e ::= x \mid e\ e \mid \lambda x.e \mid \quad c \quad \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

Primitives

Boolean: true, false, $<$, $>$, ...

Arithmetic: $\times$, $+$, $\div$, $-$, 0, 1, ...

Tuples: pair, fst, snd

Lists: nil, cons, hd, tl

Homework Exercises

- Implement lambda-calculus interpreter that supports an innermost and a outermost strategy.