

# Functional Programming

WS 2008/09

Christian Sternagel (VO)<sup>1</sup> Friedrich Neurauter (PS)<sup>2</sup>

Computational Logic  
 Institute of Computer Science  
 University of Innsbruck

7 November 2008

<sup>1</sup>christian.sternagel@uibk.ac.at

<sup>2</sup>friedrich.neurauter@uibk.ac.at

## $\lambda$ -Calculus

### $\lambda$ -Terms

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

### Example

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

## $\lambda$ -Calculus (cont'd)

### $\beta$ -Reduction

the term  $s$  ( $\beta$ -)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.

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

### Example

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

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

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

## 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, monads, ...

# Sets

- ▶ order of elements not important
- ▶ no duplicates

## Example

$$\{1, 2, 3, 5\} = \{5, 1, 3, 2\}$$

$$\{1, 1, 2, 2\} = \{1, 2\}$$

# Set Operations

description	notation	OCaml
empty set	$\emptyset$	<code>empty : 'a set</code>
membership	$e \in S$	<code>mem : 'a -&gt; 'a set -&gt; bool</code>
union	$S \cup T$	<code>union : 'a set -&gt; 'a set -&gt; 'a set</code>
difference	$S \setminus T$	<code>diff : 'a set -&gt; 'a set -&gt; 'a set</code>

# OCaml Datatype for Sets

## Idea

- ▶ use binary search tree
- ▶ easy to implement
- ▶ (potentially) efficient lookup and insertion

## Type

```
type 'a t = 'a BinTree.t
```

## Empty set

```
let empty = Empty
```

# Membership Test: $e \in S$

```
let rec mem x = function  
  | Empty          -> false  
  | Node(_,v,_)   when x = v -> true  
  | Node(l,v,_)   when x < v -> mem x l  
  | Node(_,v,r)   when x > v -> mem x r
```

Union:  $S \cup T$ 

```

let rec union set = function
| Empty      -> set
| Node(l,v,r) -> union (union (insert v set) l) r

```

with

```

let rec insert x = function
| Empty      -> singleton x
| Node(l,y,r) as n -> if x = y then n else (
  if x < y then Node(insert x l,y,r)
  else Node(l,y,insert x r)
)

```

Difference:  $S \setminus T$ 

```

let rec remove_max = function
| Empty      -> failwith "empty_tree"
| Node(l,v,Empty) -> (v,l)
| Node(l,v,r) ->
  let (m,r) = remove_max r in (m,Node(l,v,r))

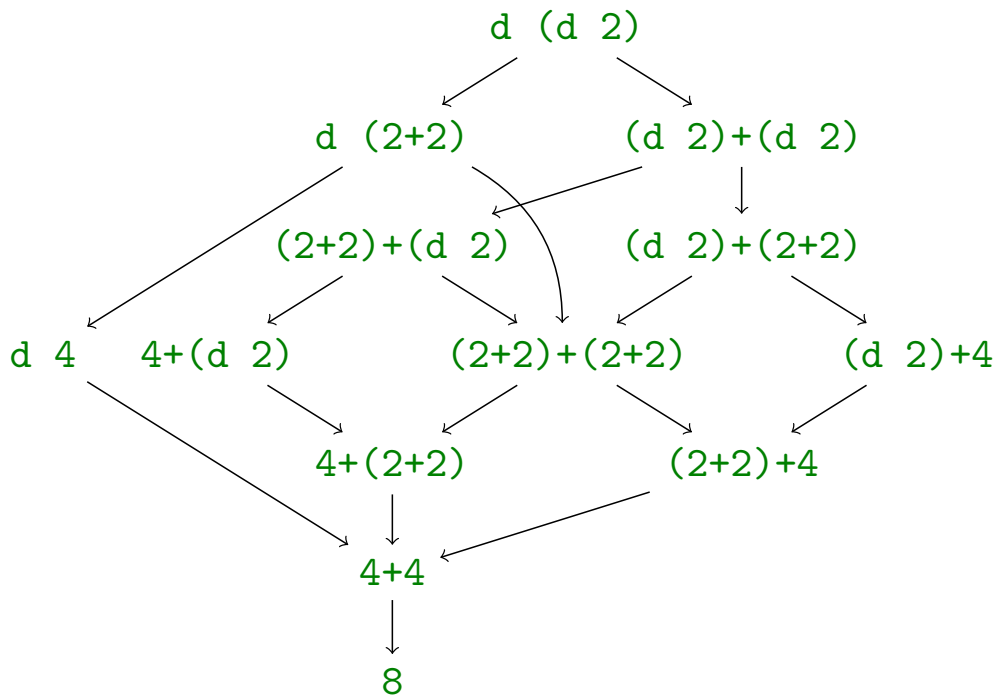
let rec remove x = function
| Empty      -> Empty
| Node(l,y,r) -> if x < y then Node(remove x l,y,r) else (
  if x > y then Node(l,y,remove x r) else (
    if l = Empty then r
    else let (m,l) = remove_max l in Node(l,m,r)
  )
)

let rec diff set = function
| Empty      -> set
| Node(l,v,r) -> diff (remove v set) (union l r)

```

## Example

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



## Strategies

### Strategy

- ▶ fixes evaluation order
- ▶ call-by-value
- ▶ call-by-name

### Example

- ▶ call-by-value:

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

- ▶ call-by-name:

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

## (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.,

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

### Example

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

- ▶  $(\lambda y. y) x$  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.,

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

### Example

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

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

### Definition (Weak head normal form)

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

$$t \neq u v$$

## 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



# $\lambda$ -Terms

## Type

```
type var = (Strng.t * int)
```

```
type t = Var of var | Abs of (var * t) | App of (t * t)
```

## Example

```
x0      Var(['x'],0)
 $\lambda$ x0.x0 Abs((['x'],0),Var(['x'],0))
x0 x0   App(Var(['x'],0),Var(['x'],0))
```

# $\lambda$ -Terms (cont'd)

## Abbreviations

```
let var s = Var(Strng.of_string s,0)
```

```
let abs xs body = Lst.foldr (fun x t ->
  Abs((Strng.of_string x,0),t)) xs body
```

```
let app ts = Lst.foldl1 (fun s t -> App(s,t)) ts
```

## Example

```
x0      var "x"
 $\lambda$ x0.x0 abs ["x"] (var "x")
x0 x0   app [var "x";var "x"]
```

# Variable Renaming

## Idea

to ensure  $\mathcal{B}\mathcal{V}\text{ar}(s) \cap \mathcal{V}\text{ar}(t) = \emptyset$  add maximal index in  $t$  plus 1 to indexes of bound variables in  $s$

## Maximal index

```
let rec max_index = function
| Var(_,i)      -> i
| Abs((_,i),u)  -> max i (max_index u)
| App(u,v)     -> max (max_index u) (max_index v)
```

# Variable Renaming (cont'd)

## Rename bound variables

```
let rename_bound i t =
  let inc (id,i) j = (id,i+j) in
  let rec rename_bound i bvs = function
  | Var x as v ->
    if St.mem x bvs then Var(inc x i) else v
  | Abs(x,u) ->
    Abs(inc x i, rename_bound i (St.insert x bvs) u)
  | App(u,v) ->
    App(rename_bound i bvs u, rename_bound i bvs v)
  in
  rename_bound i St.empty t
```

# Substitutions

## Replace single variable

```

let rec substitute x t = function
| Var y as v    -> if y = x then t else v
| Abs(y,u) as s -> if y = x then s
                  else Abs(y,substitute x t u)
| App(u,v)      -> App(substitute x t u,substitute x t v)

```

# $\beta$ -Steps

## Single $\beta$ -step without context

```

let beta = function
| App(Abs(x,u),v) ->
  substitute x v (rename_bound (max_index v + 1) u)
| _                -> failwith "not a  $\beta$ -redex"

```

# Reducts

## All possible $\beta$ -steps

```
let rec reducts = function
| Var _          -> []
| Abs(x,u)       -> Lst.map (fun t -> Abs(x,t)) (reducts u)
| App(u,v) as t ->
  let us = Lst.map (fun l -> App(l,v)) (reducts u) in
  let vs = Lst.map (fun r -> App(u,r)) (reducts v) in
  if is_redex t then beta t :: us @ vs
  else us @ vs
```