

# Functional Programming

## WS 2007/08

Christian Sternagel<sup>1</sup> (VO + PS)  
Friedrich Neurauter<sup>2</sup> (PS)  
Harald Zankl<sup>3</sup> (PS)

Computational Logic  
Institute of Computer Science  
University of Innsbruck

23 November 2007

---

<sup>1</sup>[christian.sternagel@uibk.ac.at](mailto:christian.sternagel@uibk.ac.at)

<sup>2</sup>[friedrich.neurauter@uibk.ac.at](mailto:friedrich.neurauter@uibk.ac.at)

<sup>3</sup>[harald.zankl@uibk.ac.at](mailto:harald.zankl@uibk.ac.at)

# Overview

## Week 6 - Implementation of $\lambda$

Summary of Week 5

Implementation of Sets

Evaluation Strategies

$\lambda$  in OCaml

# Overview

## Week 6 - Implementation of $\lambda$

Summary of Week 5

Implementation of Sets

Evaluation Strategies

$\lambda$  in OCaml

# $\lambda$ -Calculus

## $\lambda$ -Terms

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

## Example

$x y$	$(x y)$	<i>"x applied to y"</i>
$\lambda x.x$	$(\lambda x.x)$	<i>"lambda x to x"</i>
$\lambda xy.x$	$(\lambda x.(\lambda y.x))$	<i>"lambda x y to x"</i>
$\lambda xyz.x z (y z)$	$(\lambda x.(\lambda y.(\lambda z.((x z) (y z))))))$	<i>"lambda x y z to ..."</i>
$\lambda x.x x$	$(\lambda x.(x x))$	<i>"lambda x to (x applied to x)"</i>
$(\lambda x.x) x$	$((\lambda x.x) x)$	<i>"(lambda x to x) applied to x"</i>

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

## $\beta$ -Reduction

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

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

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

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

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

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

# Overview

## Week 6 - Implementation of $\lambda$

Summary of Week 5

**Implementation of Sets**

Evaluation Strategies

$\lambda$  in OCaml

# 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 test	$e \in S$	<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 set = Empty | Node **of** 'a set \* 'a \* 'a set

## Empty set

**let** empty = Empty

## Membership Test: $e \in S$

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

## Union: $S \cup T$

```
let singleton x = Node (Empty, x, Empty);;
let rec insert x = function
  | Empty -> singleton x
  | Node (_, v, _) as n when x = v -> n
  | Node (lt, v, rt) when x < v -> Node (insert x lt, v, rt)
  | Node (lt, v, rt) when x > v -> Node (lt, v, insert x rt)
;;
let rec union xt = function
  | Empty -> xt
  | Node (lt, v, rt) -> union (union (insert v xt) lt) rt
;;
```

Difference:  $S \setminus T$

**let rec** remove x = **function**

| Empty  $\rightarrow$  Empty

| Node (lt, v, rt) **when** x = v  $\rightarrow$  union lt rt

| Node (lt, v, rt) **when** x < v  $\rightarrow$  Node (remove x lt, v, rt)

| Node (lt, v, rt) **when** x > v  $\rightarrow$  Node (lt, v, remove x rt)

::

**let rec** diff xt = **function**

| Empty  $\rightarrow$  xt

| Node (lt, v, rt)  $\rightarrow$  diff (remove v xt) (union lt rt)

::

# Overview

## Week 6 - Implementation of $\lambda$

Summary of Week 5

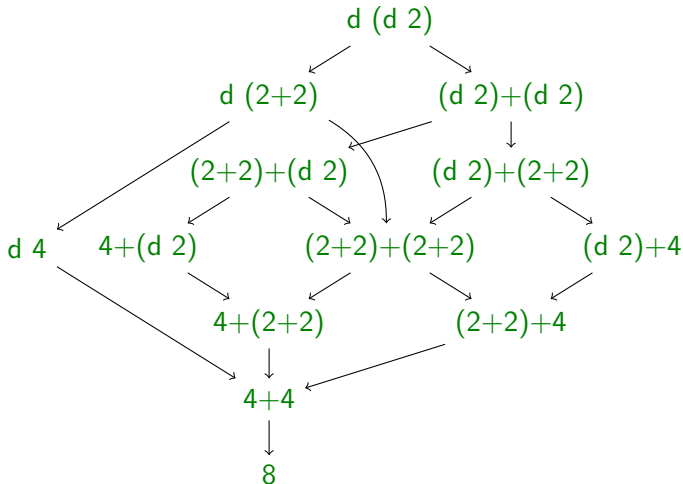
Implementation of Sets

**Evaluation Strategies**

$\lambda$  in OCaml

## 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

# Overview

## Week 6 - Implementation of $\lambda$

Summary of Week 5

Implementation of Sets

Evaluation Strategies

$\lambda$  in OCaml

## Type for $\lambda$ -Terms

```
type var = (Strng.t * int);;
type t = Var of var | Abs of (var * t) | App of (t * t);;
```

### Example

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

### Abbreviations

```
let var c = Var ([c], 0);;
let abs xs body = Lst.fold (fun x xs  $\rightarrow$  Abs (([x], 0), xs)) body xs;;
let app ts = Lst.fold_left1 (fun s t  $\rightarrow$  App (s, t)) ts;;
```

# 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)  $\rightarrow$  i

| Abs ((\_, i), u)  $\rightarrow$  max i (max\_index u)

| App (u, v)  $\rightarrow$  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  $\rightarrow$  if St.mem x bvs then Var (inc x i) else v  
    | Abs (x, u)  $\rightarrow$  Abs (inc x i, rename_bound i (St.insert x bvs) u)  
    | App (u, v)  $\rightarrow$  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  $\rightarrow$  **if** y = x **then** t **else** v

| Abs (y, u) **as** s  $\rightarrow$  **if** y = x **then** s **else** Abs (y, substitute x t u)

| App (u, v)  $\rightarrow$  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))  
| _ -> failwith "Lambda.beta: not a redex"  
;;
```