Functional Programming
WS 2007/08

Christian Sternagel[1] (VO + PS)
Friedrich Neurauter[2] (PS)
Harald Zankl[3] (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

23 November 2007

[1] christian.sternagel@uibk.ac.at
[2] friedrich.neurauter@uibk.ac.at
[3] harald.zankl@uibk.ac.at

Overview

Overview

λ-Calculus

λ-Terms

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

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*"
$\lambda xyz.x\ z\ (y\ z)$	$(\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z)))))$	"*lambda x y z to ...*"
$\lambda x.x\ x$	$(\lambda x.(x\ x))$	"*λx to (x applied to x)*"
$(\lambda x.x)\ x$	$((\lambda x.x)\ x)$	"*(λx to x) applied to x*"

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.

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

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

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	$\varnothing$	empty : 'a set
membership test	$e \in S$	mem : 'a −> 'a set −> bool
union	$S \cup T$	union : 'a set −> 'a set −> 'a set
difference	$S \setminus T$	diff : 'a set −> 'a set −> 'a set

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
;;
```

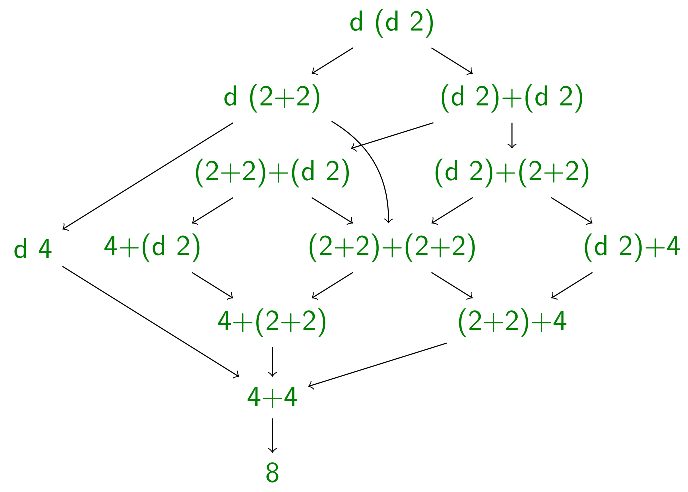

Difference: $S \setminus T$

```
let rec remove x = function
  | Empty -> Empty
  | Node (lt, v, rt) when x = v -> union lt rt
  | Node (lt, v, rt) when x < v -> Node (remove x lt, v, rt)
  | Node (lt, v, rt) when x > v -> Node (lt, v, remove x rt)
;;
let rec diff xt = function
  | Empty -> xt
  | Node (lt, v, rt) -> diff (remove v xt) (union lt rt)
;;
```

Overview

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:

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

- call-by-name:

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

(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.,

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

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.,

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

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

- $(\lambda x.(\lambda y.y)\ x)\ z$ 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 ($\mathit{WHNF}(t)$) 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

Type for λ-Terms

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

Example

x_0	var 'x'
$\lambda x_0.x_0$	abs ['x'] (var 'x')
$x_0\ x_0$	app [var 'x'; var 'x']

Abbreviations

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


Variable Renaming

Idea

to ensure $\mathcal{BV}\mathrm{ar}(s) \cap \mathcal{V}\mathrm{ar}(t) = \varnothing$ 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)
;;
```

β-Steps

Single β-step without context

```
let beta = function
  | App (Abs (x, u), v) -> substitute x v (rename_bound (max_index v + 1) u)
  | _ -> failwith "Lambda.beta: not a redex"
;;
```