

# Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science  
University of Innsbruck

WS 2017/2018

Lecture 6



## Overview

- Evaluation Strategies
- Abstract Data Types
- Sets and Binary Search Trees

## Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

CS,HZ,EZ (DCS @ UIBK)

lecture 6

2/26

Evaluation Strategies

## Recall $\lambda$ -Terms

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

## Examples

conventions	verbose	in words
$x y$	$(x y)$	" $x$ applied to $y$ "
$\lambda x. x$	$(\lambda x. x)$	"lambda $x$ to $x$ " (identity function)
$\lambda x y. x$	$(\lambda x. (\lambda y. x))$	"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 $x$ "

### Recall $\beta$ -Reduction

- term  $s$  ( $\beta$ -)reduces to term  $t$  in one step
- written:  $s \rightarrow_{\beta} t$
- iff there is subterm  $(\lambda x. u) v$  of  $s$ , s.t.,
- replacing  $(\lambda x. u) v$  in  $s$  by  $u[x := v]$  results in  $t$

### Examples

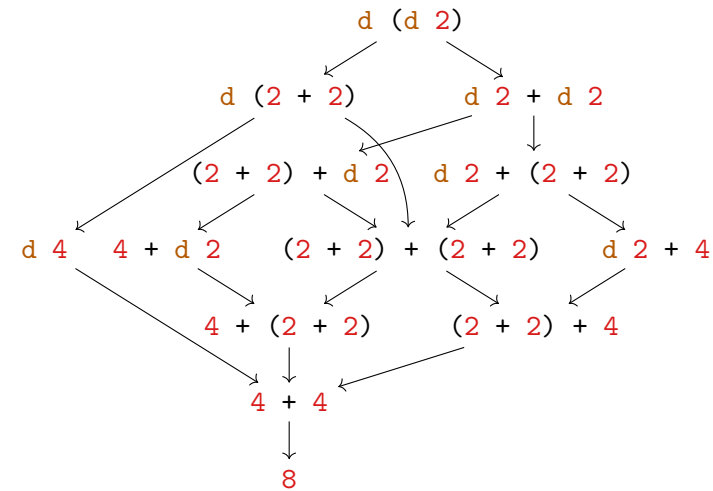
$$K = \lambda xy. x$$

$$I = \lambda x. x$$

$$\Omega = (\lambda x. x x) (\lambda x. x x)$$

### Order of Evaluation

- consider  $d\ x = x + x$
- the term  $d\ (d\ 2)$  may be evaluated as follows



### (Reduction) Strategies

what is called **evaluation strategy** in programming, is typically called **reduction strategy** in  $\lambda$ -calculus

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

### Example

- call by value

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

- call by name

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

### Applicative Order Reduction

- reduce rightmost innermost redex
- redex is **innermost** if it does not contain redexes itself

### Example

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

## Normal Order Reduction

- reduce leftmost outermost redex
- redex is **outermost** if it is not contained in another redex

### Example

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

## Exercises

- consider the  $\lambda$ -terms
- $S = \lambda xyz. x z (y z)$
- $K = \lambda xy. x$
- $I = \lambda x. x$
- reduce  $S K I$  to NF using applicative order reduction
- reduce  $S K I$  to NF using normal order reduction

## Further Classification of $\lambda$ -Terms

- a term that is not an application is called **value**
- a term is called **weak head normal form** (WHNF) if it satisfies:

$$\text{whnf}(x) = \text{true}$$

$$\text{whnf}(\lambda x. t) = \text{true}$$

$$\text{whnf}((\lambda x. t) u) = \text{false}$$

$$\text{whnf}(t u) = \text{whnf}(t)$$

### Examples

term $t$	value	WHNF
$(\lambda x. x) x$	✗	✗
$x y$	✗	✓
$x$	✓	✓
$\lambda x. (\lambda y. y) x$	✓	✓

## Call by Value

- stop at values
- otherwise choose outermost redex whose right-hand side is value
- corresponds to strict (or eager) evaluation
- adopted by most programming languages

## Call by Name

- stop at WHNFs
- otherwise same as normal order (that is, leftmost outermost redex)
- corresponds to lazy evaluation (without memoization)
- adopted for example by Haskell

## Idea

- hide implementation details
- just provide interface
- allows us to change implementation (e.g., make more efficient) without breaking client code

## Haskell

- consider module
 

```
module M (T, ...) where
  type T = C1 | ... | CN
```
- only name `T` is exported, but none of `C1` to `CN`
- thus we are not able to directly construct values of type `T`
- if we want to export `C1` to `CN`, we can use `T(..)` in export list

## Example – Sets as Lists

```
module Set (Set, empty, insert, mem, union, diff) where
import qualified Data.List as List
data Set a = Set [a]
```

```
empty :: Set a
empty = Set []
```

```
insert :: Eq a => a -> Set a -> Set a
insert x (Set xs) = Set $ List.nub $ x : xs
```

```
mem :: Eq a => a -> Set a -> Bool
x `mem` Set xs = x `elem` xs
```

```
union, diff :: Eq a => Set a -> Set a -> Set a
union (Set xs) (Set ys) = Set $ List.nub $ xs ++ ys
diff (Set xs) (Set ys) = Set $ xs List.\ \ ys
```

## Characteristics of Sets

- order of elements not important
- no duplicates

## Examples

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

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

## Operations on Sets

description	notation	Haskell
empty set	$\emptyset$	<code>empty :: Set a</code>
insertion	$\{x\} \cup S$	<code>insert :: a -&gt; Set a -&gt; Set a</code>
membership	$e \in S$	<code>mem :: a -&gt; Set a -&gt; Bool</code>
union	$S \cup T$	<code>union :: Set a -&gt; Set a -&gt; Set a</code>
difference	$S \setminus T$	<code>diff :: Set a -&gt; Set a -&gt; Set a</code>

## Note – Imports

- `import M` imports **all** functions and types exported by module `M`
- we may restrict to `f1, ..., fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything **except** the functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix `M.`
- `import qualified M as N`, same as `import qualified M` but additionally rename `M` to `N`

## New Types

- `data` with single constructor `Set` was used to hide implementation
- in this common special case use `newtype Set a = Set a` instead
- only difference: `newtype` has better performance than `data`

## Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides **selector functions** `n1 :: T -> t1, ..., nN :: T -> tN`

## Example

- `data Equation a = E { lhs :: a, rhs :: a }`

```
ghci> let e1 = E "10" "5+5"
ghci> let e2 = E { rhs = "5+5", lhs = "10" }
ghci> lhs e1
"10"
ghci> rhs e2
"5+5"
```

## Empty Set

```
empty :: Set a
empty = Set Empty
```

## Membership

```
mem :: Ord a => a -> Set a -> Bool
x `mem` s = x `memTree` rep s
  where
    memTree x Empty = False
    memTree x (Node y l r) =
      case compare x y of
        EQ -> True
        LT -> x `memTree` l
        GT -> x `memTree` r
```

## The Type

- use type `BTree` without prefix: `import BTree (BTree(..))`
- import remaining functions from `BTree` with prefix `import qualified BTree`
- internal representation of set is binary tree (with selector `rep`)  
`newtype Set a = Set { rep :: BTree a }`

## Note

- `newtype Set a = Set { rep :: BTree a }` is almost the same as writing `type Set a = BTree a`
- additionally type system prevents us from “accidentally” (that is, without constructor `Set`) using `BTrees` as `Sets`
- no runtime penalty (in contrast to `data Set a = Set { rep :: BTree }`)
- reason: `newtype` restricted to **single** constructor (usually of same name as newly introduced type)
- `data` may have arbitrarily many constructors (e.g., `Empty` and `Node`)

## Insertion

```
insert :: Ord a => a -> Set a -> Set a
insert x s = Set $ insertTree x $ rep s
```

```
insertTree :: Ord a => a -> BTree a -> BTree a
insertTree x Empty = Node x Empty Empty
insertTree x (Node y l r) =
  case compare x y of
    EQ -> Node y l r
    LT -> Node y (insertTree x l) r
    GT -> Node y l (insertTree x r)
```

## Union

```
union :: Ord a => Set a -> Set a -> Set a
union s t = Set $ rep s `unionTree` rep t
```

```
unionTree :: Ord a => BTree a -> BTree a -> BTree a
unionTree Empty s          = s
unionTree (Node x l r) s =
  insertTree x $ l `unionTree` r `unionTree` s
```

## Remove Given Element

```
removeTree :: Ord a => a -> BTree a -> BTree a
removeTree x Empty          = Empty
removeTree x (Node y l r) = case compare x y of
  LT -> Node y (removeTree x l) r
  GT -> Node y l (removeTree x r)
  EQ -> case splitMaxTree l of
    Nothing -> r
    Just (m, l') -> Node m l' r
```

### For Binary Search Tree (BST)

- **x** smaller **y**: **x** can only occur in **l**
- **x** greater **y**: **x** can only occur in **r**
- **x** equals **y**: remove current node and
- combine **l** and **r** into new BST
- therefore, take maximum of **l** as new root
- guarantees that all other elements in **l** are smaller and
- that all elements in **r** are greater

## Removing the Maximal Element

```
splitMaxTree :: BTree a -> Maybe (a, BTree a)
splitMaxTree Empty          = Nothing
splitMaxTree (Node x l Empty) = Just (x, l)
splitMaxTree (Node x l r)   =
  let Just (m, r') = splitMaxTree r
  in Just (m, Node x l r')
```

## The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-based error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

## Example – Safe Head

```
safeHead (x:_) = Just x
safeHead _     = Nothing
```

## Difference

```
diff :: Ord a => Set a -> Set a -> Set a
diff s t = Set $ rep s `diffTree` rep t
```

```
diffTree :: Ord a => BTree a -> BTree a -> BTree a
diffTree t Empty          = t
diffTree t (Node x l r) =
  removeTree x t `diffTree` l `diffTree` r
```

## Exercises (for November 24th)

1. Read Chapter 4 of *Real World Haskell* and Section 5 of the [lecture notes on the lambda calculus](#).
2. Reduce 'add 2 3' to NF using applicative and normal order reduction.
3. Let `type Strat = Term -> [Term]` be the type of reduction strategies. Implement the strategy `root :: Strat` which applies a single  $\beta$ -step at the root (if possible).
4. Implement a strategy combinator `nested :: Strat -> Strat` that, given a strategy `s`, results in a new strategy which tries to apply `s` at all non-root positions.
5. Building on the previous functions, implement single-step call by name reduction `cbn :: Strat`.
6. Implement the function `equals :: Ord a => Set a -> Set a -> Bool`, checking whether two sets are equal.

## Examples

- `root x = []` – no beta-step possible
- `root ((λx. x) u) = [u]` – root reduction
- `root (x (λx. t) u) = []` – no redex at root position
- single beta-steps strictly below root position

$$\text{nested root } (((\lambda x. x) y) ((\lambda z. z) w)) = [y ((\lambda z. z) w), (\lambda x. x) y w]$$

- single-step call by name reduction

$$\text{cbn } (((\lambda x. x) ((\lambda y. y) z)) ((\lambda w. w) v)) = [((\lambda y. y) z) ((\lambda w. w) v)]$$