

Functional Programming

Functional Data Structures

H. Turgut Uyar

2013-2016

1 / 28

License



© 2013-2016 H. Turgut Uyar

You are free to:

- Share – copy and redistribute the material in any medium or format
- Adapt – remix, transform, and build upon the material

Under the following terms:

- Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made.
- NonCommercial – You may not use the material for commercial purposes.
- ShareAlike – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

For more information:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Read the full license:

<https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>

2 / 28

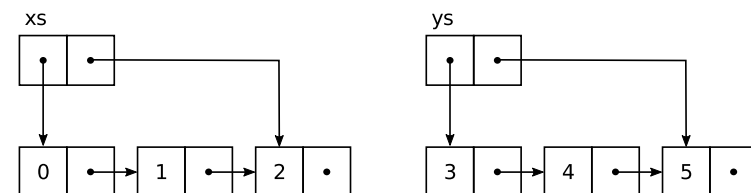
Topics

- 1 Functional Data
 - Immutability
 - Abstract Data Types
- 2 Example: Sets
 - Interface
 - List Representation
 - Tree Representation

3 / 28

Appending Lists

- append a list at the end another list, and get a third list

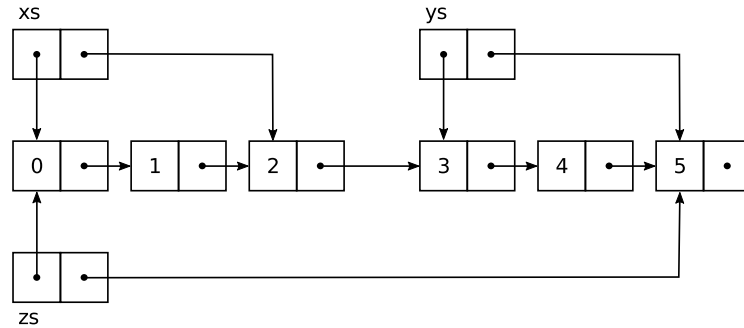


example: C

```
xs->last->next = ys->head;  
zs->head = xs->head;  
zs->last = ys->last;
```

4 / 28

Appending Lists

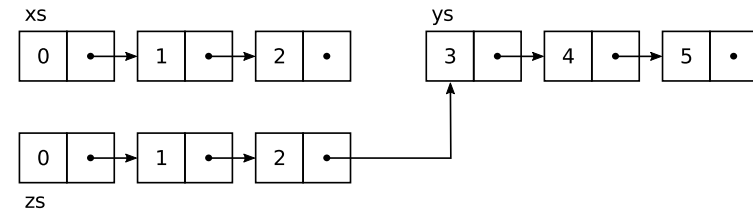


- very fast
- destroys both xs and ys

5 / 28

Appending Lists

```
(++) :: [a] -> [a] -> [a]
[]    ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```



- copy some parts, share some parts

6 / 28

Updating Lists

- update an element in a list:
update [0,1,2,3,4] 2 7 ~> [0,1,7,3,4]

```
update :: [a] -> Int -> a -> [a]
update [] _ _ = error "index out of bounds"
update (_:xs) 0 y = y : xs
update (x:xs) n y = x : update xs (n - 1) y
```

- exercise: draw data structures for above example values

7 / 28

Abstract Data Types

- **abstract data type:**
- hidden representation
- public operations

8 / 28

Example: Natural Numbers

```
module Nat (  
  Nat,  
  add,      -- Nat -> Nat -> Nat  
  sub       -- Nat -> Nat -> Nat  
) where
```

9 / 28

Example: Natural Numbers

```
data Nat = Zero | Succ Nat  
  deriving Show  
  
add :: Nat -> Nat -> Nat  
add n      Zero = n  
add Zero   n    = n  
add (Succ m) n  = Succ (add m n)  
  
sub :: Nat -> Nat -> Nat  
sub n      Zero = n  
sub Zero   _    = error "subtract from zero"  
sub (Succ n1) (Succ n2) = sub n1 n2
```

10 / 28

Set Interface

```
module Set (  
  Set,  
  empty,      -- Set a  
  add,        -- Ord a => Set a -> a -> Set a  
  makeSet,    -- Ord a => [a] -> Set a  
  contains,   -- Ord a => Set a -> a -> Bool  
  union,      -- Ord a => Set a -> Set a -> Set a  
  card,       -- Set a -> Int  
  mapSet      -- Ord b => (a -> b) -> Set a -> Set b  
) where
```

11 / 28

List Representation

- using an ordered list of elements without repetition

```
data Set a = OrderedList [a]  
  deriving Show
```

12 / 28

Empty Set

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

13 / 28

Adding Elements

```
add :: Ord a => Set a -> a -> Set a
add (OrderedList xs) x = OrderedList (insert xs x)

insert :: Ord a => [a] -> a -> [a]
insert [] y = [y]
insert xs@(x':xs') y
  | y < x'      = y : xs
  | y > x'      = x' : insert xs' y
  | otherwise   = xs
```

14 / 28

Set from List

```
makeSet :: Ord a => [a] -> Set a
makeSet = foldl add empty
```

15 / 28

Membership Check

```
contains :: Ord a => Set a -> a -> Bool
contains (OrderedList xs) = search xs

search :: Ord a => [a] -> a -> Bool
search [] _ = False
search (x:xs) y
  | y == x      = True
  | y < x       = False
  | otherwise   = search xs y
```

16 / 28

Set Union

```
union :: Ord a => Set a -> Set a -> Set a
union s1      (OrderedList []) = s1
union (OrderedList [])      s2 = s2
union (OrderedList (x:xs)) s2 =
    ((OrderedList xs) 'union' s2) 'add' x
```

17 / 28

Set Cardinality

```
card :: Set a -> Int
card = length . makeList

makeList :: Set a -> [a]
makeList (OrderedList xs) = xs
```

18 / 28

Function Mapping

```
mapSet :: Ord b => (a -> b) -> Set a -> Set b
mapSet f = makeSet . map f . makeList
```

19 / 28

Tree Representation

- using an ordered binary tree of elements without repetition

```
data Set a = Nil | Node a (Set a) (Set a)
    deriving Show
```

20 / 28

Empty Set

```
empty :: Set a
empty = Nil
```

21 / 28

Adding Elements

```
add :: Ord a => Set a -> a -> Set a
add Nil y = Node y Nil Nil
add s@(Node x left right) y
  | y < x    = Node x (add left y) right
  | y > x    = Node x left (add right y)
  | otherwise = s
```

22 / 28

Set from List

```
makeSet :: Ord a => [a] -> Set a
makeSet = foldl add empty
```

23 / 28

Membership Check

```
contains :: Ord a => Set a -> a -> Bool
contains Nil _ = False
contains (Node x left right) y
  | y < x    = contains left x
  | y > x    = contains right x
  | otherwise = True
```

24 / 28

Set Union

```
union :: Ord a => Set a -> Set a -> Set a
union s1 Nil = s1
union Nil s2 = s2
union (Node x left right) s2 =
  ((left 'union' right) 'union' s2) 'add' x
```

25 / 28

Set Cardinality

```
card :: Set a -> Int
card = length . makeList

makeList :: Set a -> [a]
makeList Nil = []
makeList (Node x left right) =
  makeList left ++ [x] ++ makeList right
```

26 / 28

Function Mapping

```
mapSet :: Ord b => (a -> b) -> Set a -> Set b
mapSet f = makeSet . map f . makeList
```

- would the resulting tree be balanced?

27 / 28

References

Required Reading: Thompson

- Chapter 16: Abstract data types

Recommended Reading: Okasaki

- Purely Functional Data Structures

28 / 28