Functional Programming

Functional Data Structures

H. Turgut Uyar

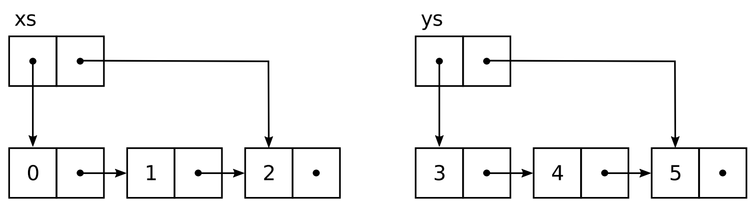
2013-2016

License

Topics

Appending Lists

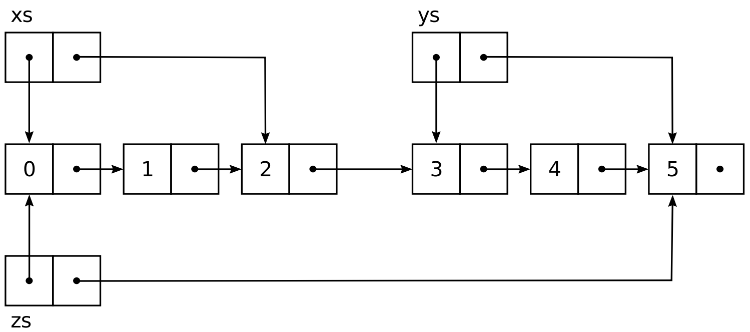
- append a list at the end another list, and get a third list



example: C

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

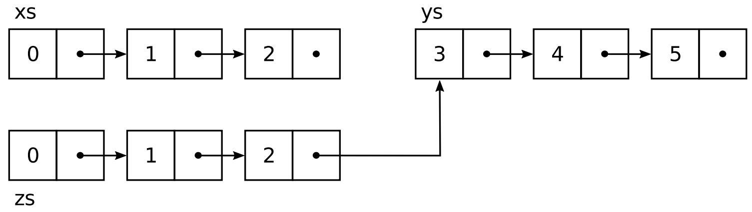
Appending Lists



- very fast
- destroys both `xs` and `ys`

Appending Lists

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



- copy some parts, share some parts

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

Abstract Data Types

- abstract data type:
- hidden representation
- public operations

Example: Natural Numbers

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


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
```


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
```


List Representation

- using an ordered list of elements without repetition

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

Empty Set

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


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
```


Set from List

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


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
```

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
```


Set Cardinality

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

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


Function Mapping

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


Tree Representation

- using an ordered binary tree of elements without repetition

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

Empty Set

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


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
```


Set from List

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


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
```

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
```


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
```


Function Mapping

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

- would the resulting tree be balanced?

References

Required Reading: Thompson

- Chapter 16: Abstract data types

Recommended Reading: Okasaki

- Purely Functional Data Structures