Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Șerbănuță

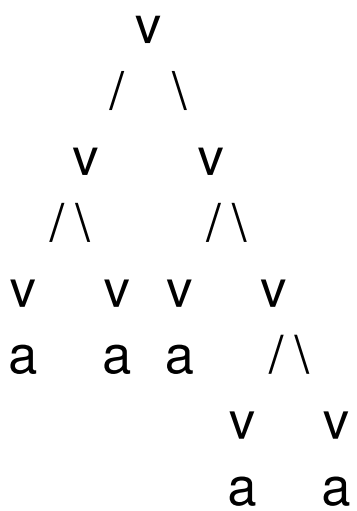
Departamentul de Informatică, FMI, UB

ADT: arbori, monoizi

Finger trees

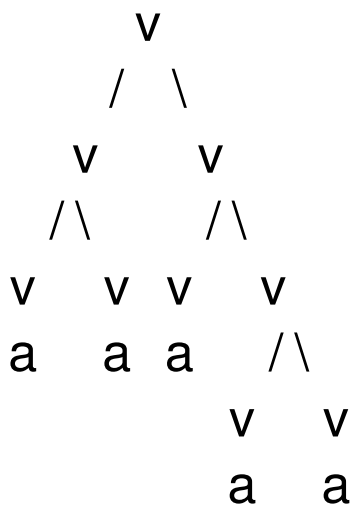
https://apfelmus.nfshost.com/articles/monoid-fingertree.html

- implementarea eficientă și unitară a structurilor de date funcționale
- informația se află în frunzelor
- nodurilor interne conțin valori a căror structură determină funcționalitatea arborelui



Finger Tree

```
data  FTree v a =   Leaf v a
                  | Node v  (FTree v a) (FTree v a)
                  deriving Show
```



Finger Tree

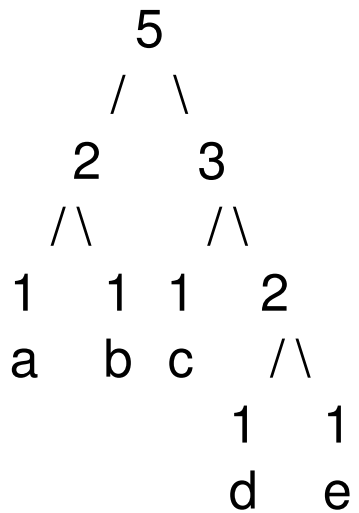
```
data  FTree v a =   Leaf v a
                  | Node v  (FTree v a) (FTree v a)
                  deriving Show

exFT = Node  5
             (Node 2
                  (Leaf 1 'a')
                  (Leaf 1 'b'))
             (Node 3
                  (Leaf 1 'c')
                  (Node 2
                       (Leaf 1 'd')
                       (Leaf 1 'e')))


*Main> :t exFT
exFT :: FTree Int String
```

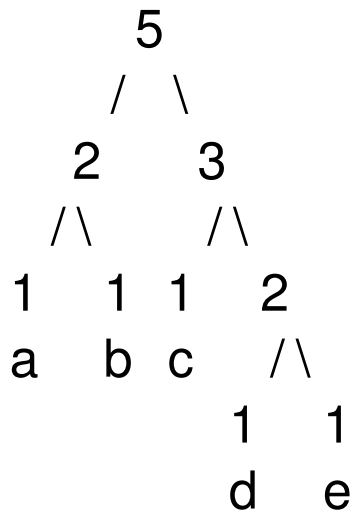
Finger Tree

Ce reprezinta informatia din arbore?



Finger Tree

Ce reprezinta informatia din arbore?



(P1) Valoarea fiecărui nod intern reprezintă numărul de elemente din arborele respectiv.

```
type Size = Int
exFT :: FTree Size Char
```

Finger Tree

- (P1) Valoarea fiecărui nod intern reprezintă numărul de elemente din arborele respectiv.

```
tag :: FTree v a -> v
tag (Leaf n _) = n
tag (Node n _ _) = n
```

- Lista datelor este alcătuită din frunze.

```
toList :: FTree v a -> [a]
toList (Leaf _ a)     = [a]
toList (Node _ x y) = toList x ++ toList y
```

Finger Tree

```
data  FTree v a =   Leaf v a
                  | Node v  (FTree v a) (FTree v a)
                  deriving Show
```

Construcția unui arbore cu (P1) se poate face cu funcții constructor specifice:

```
type Size = Integer

leaf :: a -> FTree Size a
leaf x = Leaf 1 x

node :: FTree Size a -> FTree Size a -> FTree Size a
node  t1 t2 = Node ((tag t1) + (tag t2)) t1 t2
```

```
exFT == node (node (leaf 'a')(leaf 'b'))
             (node (leaf 'c') (node (leaf 'd')(leaf 'e')))
```

Finger Tree

Accesarea elementului din poziția **n**

```
(!!!) :: FTree Size a -> Int -> a
(Leaf _ a)       !!! 0 = a
(Node _ x y)   !!! n
      | n < tag x      = x !!! n
      | otherwise      = y !!! (n - tag x)
```

```
*Main> exFT !!! 3
'd'
```

Dacă arborele se menține echilibrat, timpul de acces poate fi îmbunătățit.

Finger Tree

Priority Queue

(P2) Valoarea fiecărui nod intern reprezintă cea mai mica prioritate din arborele respectiv.

```
pqFT = Node  2
             (Node 4
                   (Leaf 16 'a')
                   (Leaf 4 'b'))
             (Node 2
                   (Leaf 2 'c')
                   (Node 8
                         (Leaf 32 'd')
                         (Leaf 8 'e')))
```

```
type Priority = Int
pqFT :: FTree Priority Char
```

Finger Tree

Construcția unui arbore cu (P2) se poate face cu funcții constructor specifice:

```
type Priority = Int

pleaf ::Priority -> a -> FTree Priority a
pleaf n x = Leaf n x

pnode :: FTree Priority a -> FTree Priority a -> FTree
    Priority a
pnode  t1 t2 = Node ((tag t1) `min` (tag t2)) t1 t2
```

```
exFT == pnode (pnode (pleaf 16 'a')(pleaf 4 'b'))
              (pnode (pleaf 2 'c') (pnode (pleaf 32 'd')(
                  pleaf 8 'e')))
```

Finger Tree

Priority Queue - determinarea elementului cîștigător

```
winner :: FTree Priority a -> a
winner t = go t
    where
    go (Leaf _ a)          = a
    go (Node _ x y)
        | tag x == tag t = go x
        | tag y == tag t = go y
```

```
*Main> winner pqFT
'c'
```

Dacă arborele se menține echilibrat, timpul de acces poate fi îmbunătățit.

Finger Trees

Unificarea celor două exemple

Observăm că

- Pentru arbori cu (P1) funcția **tag** verifică:
  $tag :: FTree\ Size\ a \to Size$
  $tag(Leaf\ \_) = 1$
  $tag(Node\ \_\ x\ y) = tag\ x\ +\ tag\ y$

- Pentru arbori cu (P2) funcția **tag** verifică:
  $tag :: FTree\ Priority\ a \to Priority$
  $tag(Leaf\ \_\ a) = priority\ a$
  $tag(Node\ \_\ x\ y) = tag\ x\ `min`\ tag\ y$

Finger Trees

Unificarea celor două exemple

Observăm că

- Pentru arbori cu (P1) funcția **tag** verifică:
  $tag :: FTree\ Size\ a \to Size$
  $tag(Leaf\ \_) = 1$
  $tag(Node\ \_\ x\ y) = tag\ x\ +\ tag\ y$

  $(Size, +, 0)$ monoid

- Pentru arbori cu (P2) funcția **tag** verifică:
  $tag :: FTree\ Priority\ a \to Priority$
  $tag(Leaf\ \_\ a) = priority\ a$
  $tag(Node\ \_\ x\ y) = tag\ x\ `min`\ tag\ y$

  $(Priority, min, maxBound)$ monoid

Finger Trees

Unificarea celor două exemple folosind monoizi

- Pentru arbori cu (P1) definim o instanța **Monoid** a lui **Size**:

```
instance Monoid Size where
    mempty  = 0
    mappend = (+)
```

- Pentru arbori cu (P2) definim o instanța **Monoid** a lui **Priority**:

```
instance Monoid Priority where
    mempty  = maxBound
    mappend = min
```

Atenție!

În acest exemplu **Size** și **Priority** sunt redenumiri ale lui **Int**. Pentru a putea fi făcute instanțe ale clasei **Monoid** simultan trebuie folosit **newtype**.

Finger Trees

Unificarea celor două exemple folosind monoizi

Constructorul pentru **Node**

```
node :: Monoid v => FTree v a -> FTree v a -> FTree v a
node x y = Node (tag x <> tag y) x y
```

Constructorul pentru **Leaf**

Cum transmitem tag-urile asociate frunzelor?

```
leaf :: Monoid v => (a->v) -> a -> FTree v a
leaf measure x = Leaf (measure x) x
```

Transmitem ca parametru o funcție care asociază fiecărei date tag-ul corespunzător.

Finger Trees

Unificarea celor două exemple folosind monoizi

Constructorii pentru **Node** și **Leaf**

```
node :: Monoid v => FTree v a -> FTree v a -> FTree v a
node x y = Node (tag x <> tag y) x y

leaf :: Monoid v => (a->v) -> a -> FTree v a
leaf measure x = Leaf (measure x) x
```

```
priority :: Char -> Int
*Main> leaf priority 'a'
Leaf 16 'a'
*Main> node (leaf priority 'a') (leaf priority 'b')
Node 3 (Leaf 16 'a') (Leaf 4 'b')
*Main> node (Leaf 16 'a') (Leaf 4 'b') :: FTree Int Char
Node 3 (Leaf 16 'a') (Leaf 4 'b')
```

Finger Trees

Unificarea căutării

```
search :: Monoid v => (v -> Bool) -> FTree v a -> Maybe a
search p t
    | p (tag t) = Just (go mempty p t)
    | otherwise = Nothing
    where
    go i p (Leaf _ a) = a
    go i p (Node _ l r)
        | p (i <> tag l) = go i p l
        | otherwise      = go (i <> tag l) p r
```

Finger Trees

Unificarea căutării

Int ca **Size**

```
instance Monoid Int where
    mempty  = 0
    mappend = (+)

win k t = search (>= k) t
```

Int ca **Priority**

```
instance Monoid Int where
    mempty  = maxBound
    mappend = min        -- Int ca Priority

win t = search (== tag t) t
```