Chris Okasaki
Purely Functional Data Structures: Numerical Representations

Jakub Kowalski
Instytut Informatyki, Uniwersytet Wrocławski

9 marca 2010
speed


treeNode


treeNode :: Node [TreeNode] [TreeNode]

fun tail (CONS (x, xs)) = xs
fun append (NIL, ys) = ys
  | append (CONS (x, xs), ys) = CONS (x, append (xs, ys))

datatype α Nat =
  ZERO
  | SUCC of Nat
fun pred (SUCC n) = n
fun plus (ZERO, n) = n
  | plus (SUCC m, n) = SUCC (plus (m, n))

datatype α List =
  NIL
  | CONS of α × α List

fun
Dense representation

```ml
structure Dense = struct
    datatype Digit = ZERO | ONE
    type Nat = Digit list (* increasing order of significance *)
    fun inc [ ] = [ONE]
        | inc (ZERO :: ds) = ONE :: ds
        | inc (ONE :: ds) = ZERO :: inc ds (* carry *)
    fun dec [ONE] = [ ]
        | dec (ONE :: ds) = ZERO :: ds
        | inc (ZERO :: ds) = ONE :: dec ds (* borrow *)
    fun add (ds, [ ]) = ds
        | add ([ ], ds) = ds
        | add (d :: ds1, ZERO :: ds2) = d :: add(ds1, ds2)
        | add (ZERO :: ds1, d :: ds2) = d :: add(ds1, ds2)
        | add (ONE :: ds1, ONE :: ds1) =
            ZERO :: inc (add(ds1, ds2)) (* carry *)
end
```

Sparse by weight representation

```ml
structure SparseByWeight = struct
  type Nat = int list (* increasing order, each a power of two *)
  fun carry (w, [ ]) = [w]
    | carry (w, ws as w'::ws') =
      if w < w' then w ::ws else carry (2*w, ws')
  fun borrow (w, ws as w'::ws') =
    if w < w' then ws' else w::borrow (2*w, ws)
  fun inc ws = carry (1, ws)
  fun dec ws = borrow (1, ws)
  fun add (ws, [ ]) = ws
    | add ([ ], ws) = ws
    | add (m as w1 :: ws1, n as w2 :: ws2)
      if w1 < w2 then w1 :: add(ws1, n)
      else if w2 < w1 then w2 :: add(m, ws2)
      else carry (2*w1 :: add(ws1, ws2))
end
```

Jakub Kowalski
Chris Okasaki Purely Functional Data Structures: Numerical Representations
Trees in numerical representations

**Common kinds of trees**
- Complete binary leaf trees
- Binomial trees
- Pennants

**Easy operations**
- link
- unlink
signature RANDOM_ACCESS_LIST
sig
  type α RList
  val empty : α RList
  val isEmpty : α RList → bool
  val cons : α × α RList → α RList
  val head : α RList → α
  val tail : α RList → α RList
  val lookup : int × α RList → α
  val update : int × α × α RList → α RList
end
Types

**datatype** \( \alpha \) TREE = LEAF of \( \alpha \) | NODE of int \( \times \alpha \) Tree \( \times \alpha \) Tree

**datatype** \( \alpha \) Digit = ZERO | ONE of \( \alpha \) Tree

**type** \( \alpha \) RList = \( \alpha \) Digit list
inc and cons

\[
\textbf{fun} \quad \text{inc} \ [\ ] = [\text{ONE}]
\]
\[
\text{inc} \ (\text{ZERO} :: \text{ds}) = \text{ONE} :: \text{ds}
\]
\[
\text{inc} \ (\text{ONE} :: \text{ds}) = \text{ZERO} :: \text{inc} \ \text{ds}
\]

\[
\textbf{fun} \quad \text{cons} \ (x, \text{ts}) = \text{consTree} \ (\text{LEAF} \ x, \text{ts}) \quad (* \ O(\log n) \ wct *
\]

\[
\textbf{fun} \quad \text{consTree} \ (t, [\ ]) = [\text{ONE} \ t]
\]
\[
\text{consTree} \ (t, \text{ZERO} :: \text{ts}) = \text{ONE} \ t :: \text{ts}
\]
\[
\text{consTree} \ (t_1, \text{ONE} \ t_2 :: \text{ts}) = \text{ZERO} :: \text{consTree} \ (\text{link} \ (t_1, t_2), \text{ts})
\]

\[
\textbf{fun} \quad \text{link} \ (t_1, t_2) = \text{NODE} \ (\text{size} \ t_1 + \text{size} \ t_2, \ t_1, \ t_2)
\]
dec and head + tail

fun dec [ONE] = [ ]
  | dec (ONE :: ds) = ZERO :: ds
  | inc (ZERO :: ds) = ONE :: dec ds

fun head ts = let val (LEAF x, _) = unconsTree ts in x end
fun tail ts = let val (_, ts') = unconsTree ts in ts' end
fun unconsTree [ONE t] = (t, [ ]) (* O(log n) wct *)
  | unconsTree (ONE t :: ts) = (t, ZERO :: ts)
  | unconsTree (ZERO :: ts) = (t, ZERO :: ts)
  let val (NODE (_, t_1, t_2), ts') = unconsTree ts
  in (t_1, ONE t_2 :: ts) end
fun lookup (i, ZERO :: ts) = lookup (i, ts) (* O(log n) wct *)
| lookup (i, ONE t :: ts) =
  if i < size t then lookupTree (i, t) else lookup (i - size t, ts)

fun lookupTree (0, LEAF x) = x (* O(log n) wct *)
| lookupTree (i, NODE (w, t₁, t₂)) =
  if i < w div 2 then lookupTree (i, t₁)
  else lookupTree (i - w div 2, t₂)
fun update (i, y, ZERO :: ts) = ZERO :: update (i, y, ts)
| update (i, y, ONE t :: ts) = 
  if i < size t then ONE (updateTree (i, y, t)) :: ts 
  else ONE t :: update (i - size t, y, ts) 

fun updateTree (0, y, LEAF x) = LEAF y 
| updateTree (i, y, NODE (w, t₁, t₂)) = 
  if i < w div 2 then NODE (w, updateTree (i, y, t₁), t₂) 
  else NODE (w, t₁, updateTree (i - w div 2, y, t₂))
Zeroless representation

datatype Digit = ONE | TWO

type nat = Digit list

fun inc [ ] = [ONE]
| inc (ONE :: ds) = TWO :: ds
| inc (TWO :: ds) = ONE :: inc ds

16_{10} = 00001_{0,1} = 2111_{1,2}
fun head (ONE (LEAF x) :: _) = x
fun head (ONE (LEAF x) :: _) = x

datatype Digit = ONE of α Tree | TWO of α Tree × α Tree
fun head (ONE (LEAF x) :: _) = x (* O(1) wct *)
  | head (TWO (LEAF x, _) :: _) = x
lazy inc and dec

```
fun lazy inc ($NIL) = $CONS (ONE, $NIL)  (* O(1) at *)
| inc ($CONS (ZERO, ds)) = $CONS (ONE, ds)
| inc ($CONS (ONE, ds)) = $CONS (ZERO, inc ds)
```
lazy inc and dec

fun lazy inc (\$NIL) = \$CONS (ONE, \$NIL) (* O(1) at *)
  | inc (\$CONS (ZERO, ds)) = \$CONS (ONE, ds)
  | inc (\$CONS (ONE, ds)) = \$CONS (ZERO, inc ds)

fun lazy dec (\$CONS (ONE, \$NIL)) = \$NIL (* O(1) at *)
  | dec (\$CONS (ONE, ds)) = \$CONS (ZERO, ds)
  | dec (\$CONS (ZERO, ds)) = \$CONS (ONE, dec ds)
lazy inc and dec

fun lazy inc ($NIL) = $CONS (ONE, $NIL)  (* O(1) at *)
  | inc ($CONS (ZERO, ds)) = $CONS (ONE, ds)
  | inc ($CONS (ONE, ds)) = $CONS (ZERO, inc ds)

fun lazy dec ($CONS (ONE, $NIL)) = $NIL  (* O(1) at *)
  | dec ($CONS (ONE, ds)) = $CONS (ZERO, ds)
  | dec ($CONS (ZERO, ds)) = $CONS (ONE, dec ds)

Let us classify digits as safe and dangerous
Lazy inc and dec on redundant binary numbers

datatype Digit = ZERO | ONE | TWO

type Nat = Digit Stream

```haskell
fun lazy
  inc ($NIL) = $CONS (ONE, $NIL) (* O(1) at *)
  | inc ($CONS (ZERO, ds)) = $CONS (ONE, ds)
  | inc ($CONS (ONE, ds)) = $CONS (TWO, ds)
  | inc ($CONS (TWO, ds)) = $CONS (ONE, inc ds)

fun lazy
dec ($CONS (ONE, $NIL)) = $NIL (* O(1) at *)
  | dec ($CONS (ONE, ds)) = $CONS (ZERO, ds)
  | dec ($CONS (TWO, ds)) = $CONS (ONE, ds)
  | dec ($CONS (ZERO, ds)) = $CONS (ONE, dec ds)

inc(222) = 1111 dec(1111) = 0111
```
lazy inc and dec on redundant binary numbers

```haskell
datatype Digit = ZERO | ONE | TWO

type Nat = Digit Stream

fun lazy inc ($NIL) = $CONS (ONE, $NIL) (* O(1) at *)
  | inc ($CONS (ZERO, ds)) = $CONS (ONE, ds)
  | inc ($CONS (ONE, ds)) = $CONS (TWO, ds)
  | inc ($CONS (TWO, ds)) = $CONS (ONE, inc ds)

fun lazy dec ($CONS (ONE, $NIL)) = $NIL (* O(1) at *)
  | dec ($CONS (ONE, ds)) = $CONS (ZERO, ds)
  | dec ($CONS (TWO, ds)) = $CONS (ONE, ds)
  | dec ($CONS (ZERO, ds)) = $CONS (ONE, dec ds)

inc(222) = 1111   dec(1111) = 0111
```
Segmented binary numbers

datatype DigitBlock = ZEROS of int | ONES of int
type Nat = DigitBlock list
Segmented binary numbers

datatype DigitBlock = ZEROS of int | ONES of int
type Nat = DigitBlock list

fun zeros (i, [ ]) = [ ]
| zeros (0, blks) = blks
| zeros (i, ZEROS j :: blks) = ZEROS (i+j) :: blks
| zeros (i, blks) = ZEROS i :: blks

fun ones (0, blks) = blks
| ones (i, ONES j :: blks) = ONES (i+j) :: blks
| ones (i, blks) = ONES i :: blks
inc and dec on segmented binary numbers

fun inc [ ] = [ONES 1] (* O(1) wct *)
  | inc (ZEROS i :: blks) = ones (1, zeros (i-1, blks))
  | inc (ONES i :: blks) = ZEROS i :: inc blks

fun dec (ONES i :: blks) = zeros (1, ones (i-1, blks)) (* O(1) wct *)
  | dec (ZEROS i :: blks) = ONES i :: dec blks
Segmentation with redundant binary numbers

```haskell
datatype Digits = ZERO | ONES of int | TWO

type Nat = Digits list

fun simpleInc [] = [ONES 1]
  | simpleInc (ZERO :: ds) = ones (1, ds)
  | simpleInc (ONES i :: ds) = TWO :: one (i-1, ds)

fun fixup (TWO :: ds) = simpleInc ds
  | fixup (ONES i :: TWO :: ds) = ONES i :: simpleInc ds
  | fixup ds = ds

fun inc ds = fixup (simpleInc ds)
```

Jakub Kowalski

Chris Okasaki  Purely Functional Data Structures: Numerical Representations
Segmentation with redundant binary numbers

```haskell
datatype Digits = ZERO | ONES of int | TWO

type Nat = Digits list

fun ones (0, ds) = ds
  | ones (i, ONES j :: ds) = ONES (i+j) :: ds
  | ones (i, ds) = ONES i :: ds

fun simpleInc [ ] = [ONES 1]
  | simpleInc (ZERO :: ds) = ones (1, ds)
  | simpleInc (ONES i :: blks) = TWO :: one (i-1, ds)

fun fixup (TWO :: ds) = ZERO :: simpleInc ds
  | fixup (ONES i :: TWO :: ds) = ONES i :: ZERO :: simpleInc ds
  | fixup ds = ds

fun inc ds = fixup (simpleInc ds) (* O(1) wct *)
```

Jakub Kowalski

Chris Okasaki Purely Functional Data Structures: Numerical Representation
Sparse representation of skew binary numbers

```haskell
type Nat = int list (* 0*(2 | ε)(0 | 1)* *)

fun inc (ws as w₁ :: w₂ :: rest) = (* O(1) wct *)
  if w₁ = w₂ then (1+w₁+w₂) :: rest else 1 :: ws
  | inc ws = 1 :: ws

fun dec (1 :: ws) = ws (* O(1) wct *)
  | dec (w :: ws) = (w div 2) :: (w div 2) :: ws
```
Implementation part 1

```plaintext
datatype α Tree = LEAF of α | NODE of α × α Tree × α Tree
type α RList = (int × α Tree) list
```
Implementation part 1

datatype α Tree = LEAF of α | NODE of α × α Tree × α Tree

type α RList = (int × α Tree) list

fun cons (x, ts as (w₁, t₁) :: (w₂, t₂) :: rest) =  (* O(1) wct *)
  if w₁ = w₂ then (1+w₁+w₂, NODE (x, t₁, t₂) :: rest)
  else (1, LEAF x) :: ts
  | cons (x, ts) = (1, LEAF x) :: ts

fun head ((1, LEAF x) :: ts) = x  (* O(1) wct *)
  | head ((w, NODE (x, t₁, t₂)) :: ts) = x

fun tail ((1, LEAF x) :: ts) = ts  (* O(1) wct *)
  | tail ((w, NODE (x, t₁, t₂)) :: ts) = (w div 2, t₁) :: (w div 2, t₂) :: ts
fun lookup (i, (w, t) :: ts) = (* O(min(i, log n)) wct *)
  if i < w then lookupTree (w, i, t)
  else lookup (i - w, ts)

fun lookupTree (1, 0, LEAF x) = x
  | lookupTree (_, 0, NODE (x, _, _)) = x
  | lookupTree (w, i, NODE (x, t₁, t₂)) =
    if i < w div 2 then lookupTree (w div 2, i - 1, t₁)
    else lookupTree (w div 2, i - 1 - w div 2, t₂)
**Type**

```plaintext
datatype Tree = NODE of int × Elem.T × Elem.T list × Tree list
```

**Lemma**

If $t$ is a skew binomial tree of rank $r$, then $2^r \leq |t| \leq 2^{r+1} - 1$. 
fun link \((t_1 \text{ as } \text{NODE}(r, x_1, xs_1, c_1), t_2 \text{ as } \text{NODE}(-, x_2, xs_2, c_2))\) =
if \text{Elem.leq}(x_1, x_2) \text{then } \text{NODE}(r + 1, x_1, xs_1, t_2 :: c_1)
else \text{NODE}(r + 1, x_2, xs_2, t_1 :: c_2)

fun skewLink \((x, t_1, t_2)\) = (* \text{O(1) wct} *)
let val NODE(r, y, ys, c) = link(t_1, t_2)
in
if \text{Elem.leq}(x, y) \text{then } NODE(r, x, y :: ys, c)
else NODE(r, y, x :: ys, c)
end
fun insert (x, ts as t1 :: t2 :: rest) = (* O(1) wct *)
    if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest
    else NODE (0, x, [ ], [ ]) :: ts
    | insert (x, ts) = NODE (0, x, [ ], [ ]) :: ts
fun insert (x, ts as t₁ :: t₂ :: rest) = (* O(1) wct *)
    if rank t₁ = rank t₂ then skewLink (x, t₁, t₂) :: rest
    else NODE (0, x, [], []) :: ts
| insert (x, ts) = NODE (0, x, [], []) :: ts

fun normalize [ ] = [ ]
| (t :: ts) = insTree (t, ts)

fun merge (ts₁, ts₂) = mergeTrees (normalize ts₁, normalize ts₂)
fun insert (x, ts as t1 :: t2 :: rest) = (* O(1) wct *)
   if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest
   else NODE (0, x, [ ], [ ]) :: ts
| insert (x, ts) = NODE (0, x, [ ], [ ]) :: ts

fun normalize [ ] = [ ]
| (t :: ts) = insTree (t, ts)

fun merge (ts1, ts2) = mergeTrees (normalize ts1, normalize ts2)

fun deleteMin ts = (* O(log n) wct *)
  let val (NODE (_, x, xs, ts1), ts2) = removeMinTree ts
  fun insertAll ([ ], ts) = ts
  | insertAll (x :: xs, ts) = insertAll (xs, insert (x, ts))
in insertAll (xs, merge (rev ts1, ts2))
Positional Number Systems
Binary numbers
Skew Binary Numbers
Trinary and Quaternary Numbers
The end

Skew Binary Random-Access Lists
Skew Binomial Heaps

removeMinTree and findMin

fun removeMinTree [ ] = raise EMPTY (* O(log n) wct *)
  | removeMinTree [t] = (t, [ ])
  | removeMinTree (t :: ts) =
      let val (t’, ts’) = removeMinTree ts
      in if Elem.leq (root t, root t’) then (t, ts) else (t’, t :: ts’) end
  end

fun findMin ts = let val (t, _) = removeMinTree ts in root t end
Kinds of trees of base $k$

- Complete $k$-ary leaf trees
- $k$-nomial trees
- $k$-ary pennants
$k$-ary numbers

Kinds of trees of base $k$

- Complete $k$-ary leaf trees
- $k$-nomial trees
- $k$-ary pennants

Processing all digits in number: $(k + 1) \log_k n = \frac{k+1}{\log_2 k} \log n$
**k-ary numbers**

Kinds of trees of base $k$

- Complete $k$-ary leaf trees
- $k$-nomial trees
- $k$-ary pennants

Processing all digits in number: 

$$(k + 1) \log_k n = \frac{k+1}{\log_2 k} \log n$$

<table>
<thead>
<tr>
<th>$k$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{k+1}{\log_2 k}$</td>
<td>3.00</td>
<td>2.52</td>
<td>2.50</td>
<td>2.58</td>
<td>2.71</td>
<td>2.85</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Seminary based on
