DD2552 Seminar 11: Purely functional data structures, II

Karl Palmskog

KTH

Wednesday October 4, 2023

Course material

- Okasaki's "Purely functional data structures", ch. 9 and ch. 10
- see also freely available PhD thesis, https://www.cs.cmu.edu/~rwh/students/okasaki.pdf

Binary numbers

- in seminar 10, we saw binomial trees that have sizes that are a power of two
- at least two other tree structures have the same property
  - complete binary leaf trees
  - pennants
- these trees can implement **binary numbers**, including arithmetic

CakeML complete binary leaf tree

```
datatype btree = Leaf | Branch btree btree

fun height t =
 case t of Leaf => 0
 | Branch t1 t2 => 1 + (max (height t1) (height t2))

fun complete t =
 case t of Leaf => True
 | Branch t1 t2 => complete t1 andalso complete t2
    andalso height t1 = height t2

fun leaves t =
 case t of Leaf => 1
 | Branch t1 t2 => leaves t1 + leaves t2
```

Binary random-access lists

```
signature RANDOMACCESSLIST =
sig
  type 'a RList

  val empty : 'a RList
  val isEmpty : 'a RList -> bool

  val cons : 'a * 'a RList -> 'a RList
  val head : 'a RList -> 'a
  val tail : 'a RList -> 'a RList

  val lookup : int * 'a RList -> 'a
  val update : int * 'a * 'a RList -> 'a RList
end
```

Dense complete binary leaf tree

```
structure BinaryRandomAccessList : RANDOMACCESSLIST =
struct
 datatype 'a Tree = LEAF of 'a
  | NODE of int * 'a Tree * 'a Tree
 datatype 'a Digit = ZERO | ONE of 'a Tree
 type 'a RList = 'a Digit list
 (* ... *)
end
```

Dense complete binary leaf tree

```
val empty = []
fun isEmpty ts = null ts

fun size (LEAF x) = 1
  | size (NODE (w, t1, t2)) = w
fun link (t1, t2) = NODE (size t1 + size t2, t1, t2)
```

Dense complete binary leaf tree

```
fun consTree (t, []) = [ONE t]
  | consTree (t, ZERO::ts) = ONE t::ts
  | consTree (t1, ONE t2::ts) =
     ZERO::consTree (link (t1, t2), ts)

fun unconsTree [] = raise EMPTY
  | unconsTree [ONE t] = (t, [])
  | unconsTree (ONE t::ts) = (t, ZERO::ts)
  | unconsTree (ZERO::ts) =
  let val (NODE (_, t1, t2), ts') = unconsTree ts'
  in (t1, ONE t2::ts') end

fun cons (x, ts) = consTree (LEAF x, ts)
fun head ts = let val (LEAF x, _) = unconsTree ts in x end
fun tail ts = let val (_, ts') = unconsTree ts in ts' end
```

Dense complete binary leaf tree

```
fun lookupTree (0, LEAF x) = x
  | lookupTree (i, LEAF x) = raise SUBSCRIPT
  | lookupTree (i, NODE (w, t1, t2)) =
  if i < w div 2 then lookupTree (i, t1)
  else lookupTree (i - w div 2, t2)

fun updateTree (0, y, LEAF x) = LEAF y
  | updateTree (i, y, LEAF x) = raise SUBSCRIPT
  | 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))
```

Dense complete binary leaf tree

```
fun lookup (i, []) = raise SUBSCRIPT
  | lookup (i, ZERO::ts) = lookup (i, ts)
  | lookup (i, ONE t::ts) =
  if i < size t then lookupTree (i, t)
  else lookup (i - size t, ts)

fun update (i, y, []) = raise SUBSCRIPT
  | 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)
```

Zeroless binary numbers

```
structure ZLBinaryRandomAccessList : RANDOMACCESSLIST =
struct
 datatype 'a Tree = LEAF of 'a
  | NODE of int * 'a Tree * 'a Tree
 datatype 'a Digit = ONE of 'a Tree
  | TWO of 'a Tree * 'a Tree
 type 'a RList = 'a Digit list
 (* ... *)
 fun head [] = raise EMPTY
  | head (ONE (LEAF x)::_) = x
  | head (TWO (LEAF x, LEAF y)::_) = x
end
```

SML nonuniform recursive data

Data declaration works fine:

```
datatype 'a Seq = NIL | CONS of 'a * ('a * 'a) Seq
```

Instance works fine:

```
CONS (1, CONS ((2, 3), CONS (((4, 5), (6, 7)), NIL)))
```

But function declaration doesn't work:

```
fun sizeS NIL = 0
 | sizeS (CONS (X, ps)) = 1 + 2 * sizeS ps
```

We get this error:

```
Can't unify 'a to 'a * 'a
(Type variable to be unified occurs in type)
```

Unwinding SML nonuniform recursion

```
datatype 'a Ep = ELEM of 'a | PAIR of 'a Ep * 'a Ep
datatype 'a Seq = NIL | CONS of 'a Ep * 'a Seq

CONS (ELEM 1, CONS (PAIR (ELEM 2, ELEM 3), NIL))

fun sizeS NIL = 0
 | sizeS (CONS (X, ps)) = 1 + 2 * sizeS ps
```

OCaml nonuniform recursive data

```
type 'a seq = NIL | CONS of 'a * ('a * 'a) seq

CONS (1, CONS ((2, 3), CONS (((4, 5), (6, 7)), NIL)))

let rec size: 'a. 'a seq -> int = function
 | NIL -> 0
 | CONS (_, xs) -> 1 + 2 * (size xs)
```

Binary random-access lists revisited

```
structure NonUniformList : RANDOMACCESSLIST =
struct
 datatype 'a RList = NIL | ZERO of ('a * 'a) RList
  | ONE of 'a * ('a * 'a) RList
 (* ... *)
end
```

Sequence 0..10 can be represented as:

```
ONE(0, ONE ((1, 2),
 ZERO (ONE ((((3, 4),(5, 6)),((7, 8),(9, 10))), NIL))))
```

Binary random-access lists revisited

```
val empty = NIL
fun isEmpty NIL = true | isEmpty _ = false

fun cons (x, NIL) = ONE (x, NIL)
  | cons (x, ZERO ps) = ONE (x, ps)
  | cons (x, ONE (y, ps)) = ZERO (cons ((x, y), ps))

fun uncons NIL = raise EMPTY
  | uncons (ONE (x, ps)) = (x, ZERO ps)
  | uncons (ZERO ps) =
     let val ((x, y), ps') = uncons ps
     in (x, ONE (y, ps')) end

fun head xs = let val (x, _) = uncons xs in x end
fun tail xs = let val (_, xs') = uncons xs in xs' end
```

Binary random-access lists revisited

```
fun lookup (i, NIL) = raise SUBSCRIPT
  | lookup (0, ONE (x, ps)) = x
  | lookup (i, ONE (x, ps)) = lookup (i - 1, ZERO ps)
  | lookup (i, ZERO ps) =
  let val (x, y) = lookup (i div 2, ps)
  in if i mod 2 = 0 then x else y end

fun fupdate (f, i, NIL) = raise SUBSCRIPT
  | fupdate (f, 0, ONE (x, ps)) = ONE (f x, ps)
  | fupdate (f, i, ONE (x, ps)) =
     cons (x, fupdate (f, i - 1, ZERO ps))
  | fupdate (f, i, ZERO ps) =
  let fun f' (x, y) = if i mod 2 = 0
  then (f x, y) else (x, f y) in
  ZERO (fupdate (f', i div 2, ps)) end

fun update (i, y, xs) = fupdate (fn x => y, i, xs)
```

Analysis

- functions still run in $O(\log n)$
- main improvement: shorter code
- secondary improvement: easier to understand(?)

Finite map signature

```
signature FINITEMAP = sig
 type Key
 type 'a Map
 val empty : 'a Map
 val bind : Key * 'a * 'a Map -> 'a Map
 val lookup : Key * 'a Map -> 'a
end
```

Tries

- binary trees are usually oblivious to element type
- if we assume more about elements (search keys) we can get more
- prototypical example: strings as lists of characters
- we now assume keys are some form of lists

Trie functor

```
functor Trie (M : FINITEMAP) : FINITEMAP =
struct
 type Key = M.Key list
 datatype 'a Map = TRIE of 'a option * 'a Map M.Map
 val empty = TRIE (NONE, M.empty)

 fun lookup ([], TRIE (NONE, m)) = raise NOTFOUND
  | lookup ([], TRIE (SOME x, m)) = x
  | lookup (k::ks, TRIE (v, m)) =
     lookup (ks, M.lookup (k, m))

 fun bind ([], x, TRIE (_, m)) = TRIE (SOME x, m)
   | bind (k::ks, x, TRIE (v, m)) = let
  val t = M.lookup (k, m) handle NOTFOUND => empty
  val t' = bind (ks, x, t)
  in TRIE (v, M.bind (k, t', m)) end
end
```

Additional voluntary reading

- Functional Algorithms Verified book, https://functional-algorithms-verified.org/
- Persistent data structures course material by Xavier Leroy, https://xavierleroy.org/CdF/2022-2023/

Note on dependent types

- in SML and OCaml, types not allowed to depend directly on terms
    - no `R^3` type for 3-dimensional tuples/vectors of reals
- in a language of **dependent types** this restriction is lifted
- we assume
    - a universe of types U
    - a type $A : U$
    - a family of types $B : A \to U$
- then we can form a **dependent sum type** $\sum_{x:A} B(x)$
- B could be a predicate, and the sum type be interpreted as the subset of A imposed by B
- more formal notes (Section 5.3): http://www.cs.nott.ac.uk/~psztxa/oplss-22/dependent.pdf