DD2552 Seminar 10: Purely functional data structures, I

Karl Palmskog

KTH

Thursday September 28, 2023

Course material

- “Purely functional data structures”, book by Chris Okasaki (not at KTHB)
- see freely available PhD thesis, https://www.cs.cmu.edu/~rwh/students/okasaki.pdf

Advantages of purely functional data

- ease of reasoning (functional correctness)
- no mutation, “old” data continues to exists (if referenced)
- in general, fewer lines of code

Performance analysis

- focus on ML family of languages, asymptotic behavior
- assume eager evaluation
- we view datatypes as implemented via pointers
  - lists are similar to linked lists
  - `cons` takes $O(1)$ time
  - append (++) takes $O(n)$ time
- more formally, need a **cost semantics**
  - outside scope of course

Heap interface

```
signature HEAP = sig
 structure Elem : ORDERED
 type Heap
 val empty : Heap
 val isEmpty : Heap -> bool
 val insert : Elem.T * Heap -> Heap
 val merge : Heap * Heap -> Heap
 val findMin : Heap -> Elem.T
 val deleteMin : Heap -> Heap
end
```

Leftist Heaps

- invented by Donald Knuth in 1970s
- heap-ordered binary trees
- satisfy the leftist property: the rank of any left child is at least as large as the rank of its right sibling
- rank of node is the length of its right spine
    - rightmost path from the node in question to an empty node

SML implementation

```
functor LeftistHeap (Element : ORDERED) : HEAP =
 structure Elem = Element
 datatype Heap = E | T of int * Elem.T * Heap * Heap

 fun rank E = 0
   | rank T (r, _, _, _) = r
 fun makeT (x, a, b) =
  if rank a >= rank b then T (rank b + 1, x, a, b)
  else T (rank a + 1, x, b, a)

 val empty = E
 fun isEmpty E = true | isEmpty _ = false

 (* ... *)
end
```

SML implementation, continued

"two [leftist] heaps can be merged by merging their right spines as you would merge two sorted lists, and then swapping the children of nodes along this path as necessary to restore the leftist property" -Okasaki

```
fun merge (h, E) = h
 | merge (E, h) = h
 | merge (h1 as T(_, x, a1, b1), h2 as T(_, y, a2, b2)) =
 if Elem.leq (x, y) then makeT (x, a1, merge (b1, h2))
 else makeT (y, a2, merge (h1, b2))
fun insert (x, h) = merge (T (1, x, E, E), h)

fun findMin E = raise EMPTY
 | findMin T (_, x, _, _) = x
fun deleteMin E = raise EMPTY
 | deleteMin T (_, _, a, b) = merge (a, b)
```

Performance analysis

- length of each right spine is at most logarithmic, so `merge` runs in $O(\log n)$
- so, `insert` and `deleteMin` run in $O(\log n)$
- `findMin` and `isEmpty` run in $O(1)$

Binomial heaps

- constructed from binomial trees
  - binomial tree of rank 0 is a singleton node
  - binomial tree of rank $r + 1$ constructed by linking two binomial trees of rank r, making one tree the leftmost child of the other
- binomial heaps are lists of binomial trees where no trees have the same rank
- can be "faster" than leftist heaps

SML implementation

```
functor BinomialHeap (Element : ORDERED) : HEAP =
struct
 structure Elem = Element
 datatype Tree = Node of int * Elem.T * Tree list
 type Heap = Tree list
 fun rank (Node (r, x, c)) = r
 fun root (Node (r, x, c)) = x
 (* ... *)
end
```

SML implementation, continued

```
fun link (t1 as Node(r, x1, c1), t2 as Node(_, x2, c2)) =
 if Elem.leq (x1, x2) then Node (r + 1, x1, t2::c1)
 else Node (r + 1, x2, t1::c2)

fun insTree (t, []) = [t]
 | insTree (t, ts as t'::ts') =
 if rank t < rank t' then t::ts
 else insTree (link (t t'), ts')
```

SML implementation, continued

```
fun insert (x, ts) = insTree (Node (0, x, []), ts)

fun merge (ts1, []) = ts1
 | merge ([], ts2) = ts2
 | merge (ts1 as t1::ts1', ts2 as t2::ts2') =
 if rank t1 < rank t2 then t1::merge (ts1', ts2)
 else if rank t2 < rank t1 then t2::merge (ts2', ts1)
 else insTree (link (t1, t2), merge (ts1', ts2'))
```

SML implementation, continued

```
fun removeMinTree [] = raise EMPTY
 | 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

fun findMin ts =
 let val (t, _) = removeMinTree ts in root t end

fun deleteMin ts =
 let val (Node (_, x, ts1), ts2) = removeMinTree ts
 in merge (rev ts1, ts2) end
```

Performance analysis

- worst case is insertion into a heap of size $n = 2^k - 1$, $O(\log n)$ time
- `merge`, `findMin`, `deleteMin` also $O(\log n)$ time

Queue interface

```
signature QUEUE = sig
 type 'a queue
 val empty : 'a queue
 val isEmpty : 'a queue -> bool
 val snoc : 'a queue * 'a -> 'a queue
 val head : 'a queue -> 'a
 val tail : 'a queue -> 'a queue
end
```

Batched queues

- use a pair of lists

CakeML implementation fragment

```
datatype 'a queue = Q ('a list) ('a list)

val empty = Q [] []

fun isEmpty q =
 case q of Q [] xs => True | _ => False

fun checkf q = case q of Q [] xs =>
 (Q (reverse xs) []) | _ => q

fun snoc q x = case q of
 Q f r => (checkf (Q f (x::r)))

fun head q = case q of
 Q (x::_) _ => x

fun tail q = case q of
 Q (_::f) r => (checkf (Q f r))
```

Performance analysis

- `snoc` and `head` in $O(1)$ worst-case time
- `tail` takes $O(n)$ worst-case time
- but `tail` runs in $O(1)$ **amortized** time
  - every element in second list has 1 credit
  - every `snoc` takes one step and allocates credit to new element (cost 2)
  - every `tail` that doesn't reverse list takes one step (cost 1)
  - every `tail` that reverses list uses up all credits and takes a step (cost 1)
- is this implementation suitable for concurrency? Not really.
  - but "cannot be beat" otherwise

Anecdote on performance analysis

- mutation analysis is about:
  - modifying a software system, creating a mutant
  - seeing if tests/verification "kills" mutant
- surviving mutants may indicate inadequate tests/verification
- a mutant of merge sort survived formal functional correctness proofs
- "missing" invariant: merging needs to be done on powers-of-two sized lists, otherwise get $O(n^2)$ instead of $O(n \log n)$ sorting time

Pairing heaps

- heap-ordered multiway trees
- perform well in practice

SML implementation

```
functor PairingHeap (Element : ORDERED) : HEAP =
struct
 structure Elem = Element
 datatype Heap = E | T of Elem.T * Heap list
 val empty = E
 fun isEmpty E = true | isEmpty _ = false
 (* ... *)
end
```

SML implementation, continued

```
fun merge (h, E) = h
  | merge (E, h) = h
  | merge (h1 as T (x, hs1), h2 as T (y, hs2)) =
  if Elem.leq (x, y) then T (x, h2::hs1)
  else T (y, h1::hs2)

fun insert (x, h) = merge (T (x, []), h)

fun mergePairs [] = E
 | mergePairs [h] = h
 | mergePairs (h1::h2::hs) =
   merge (merge (h1, h2), mergePairs hs)
```

SML implementation, continued

```
fun findMin E = raise EMPTY
  | findMin (T (x, hs)) = x

fun deleteMin E = raise EMPTY
  | deleteMin (T (x, hs)) = mergePairs hs
```