Purely Functional Data Structures

Purely Functional Data Structures
Chris Okasaki
Outline

• Persistence
• Functional vs. imperative data structures
• Amortized complexity analysis
• Amortization for persistent data structures
Immutability/persistence of data in FP

Persistence: updates do not affect existing references

Haskell:
xs = [1,2,3]
y$s = [7,8]$
zs = xs ++ ys

> zs
[1,2,3,7,8]
> xs
[1,2,3]

data is persistent

Ruby:
xs = [1,2,3]
y$s = [7,8]$
zs = xs.concat(ys)

> zs
[1,2,3,7,8]
> xs
[1,2,3,7,8]

data is ephemeral
Persistent data structures

Ephemeral (i.e. traditional) data structures:
  • updates destroy old versions

Persistent data structures:
  • old versions are unchanged by updates

Applications independent of pure FP:
  • editors (undo), version control, etc.
  • backtracking search
  • thread-safe data sharing
  • computational geometry algorithms
Degrees of persistence

- no persistence
  one version

- partial persistence
  update only last version

- full persistence
  update all versions

Purely functional = all data structures are fully persistent
Outline

• Persistence

• **Functional vs. imperative data structures**

• Amortized complexity analysis

• Amortization for persistent data structures
Example: imperative list concatenation

xs = [1,2,3]
yx = [7,8]
zx = xs.concat(ys)

• efficient: O(1) time and space
• side-effects, potential inconsistencies
• not persistent
Example: functional list concatenation

\[ \text{xs} = [1, 2, 3] \]
\[ \text{ys} = [7, 8] \]
\[ \text{zs} = \text{xs} \mathbin{\cdot} \text{ys} \]

- \( \mathcal{O}(|\text{xs}|) \) time and space requirement
- no side-effects
- fully persistent
**Model of functional data structures**

```
data T = C . . . | D . . .
x :: T
x = D (C . . .) . . .
```

\[ x \text{ is represented as a pointer data structure (tree/graph) in the heap} \]

To update the subterm \( y \):
- *update a copy* of the corresponding cell \( y \) in the heap
- *copy* all nodes on the *path* from the root to \( y \)
- (rest of the data structure is *shared* between \( x \) and \( y \))
Example: insert in binary search tree

```haskell
insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r) | x < y = Node y (insert x l) r
| otherwise = Node y l (insert x r)
```

\[ t = \text{Node 4 (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf))} \]
\[ \qquad \quad \quad \quad \quad \text{(Node 7 (Node 6 Leaf Leaf) (Node 8 Leaf (Node 9 Leaf Leaf))))} \]

\[ u = \text{insert 5 } t \]
Challenges of functional data structures

How to implement functional data structures efficiently?

• Optimize data type representation for common operations
• Goals: minimize traversal and copying
  • e.g. Haskell lists are optimized for stack operations but inefficient as queues
  • these goals are the rationale for the zipper pattern

How to analyze their time and space complexity?

• Rely heavily on amortization
• Lazy evaluation is crucial for amortizing w/ persistence
Outline

• Persistence
• Functional vs. imperative data structures
• Amortized complexity analysis
• Amortization for persistent data structures
Amortized vs. worst case analysis

“worst” worst case:
always assume maximal cost

\[ n \text{ ops} \times O(n) \text{ cost} \]
\[ \in O(n^2) \text{ total cost} \]

Ammortized worst case:
costs can be distributed over ops

\[ n \text{ ops} \times O(1) \text{ ammortized cost} \]
\[ \in O(n) \text{ total cost} \]
Tradeoffs of amortized analysis

• more accurate over lifetime of data structure

• opens up new design space e.g. self-adjusting data structures
  • can lead to overall faster data structures
    (in practice, or asymptotically over lifetime)
    e.g. splay trees e.g. union-find

• weaker guarantees about individual operations
  • not suitable for real-time applications
Banker’s method

For each operation $i$, define:
- $a_i$: amortized cost
- $t_i$: actual cost

Each operation gets $a_i$ credits

<table>
<thead>
<tr>
<th>Op is …</th>
<th>if …</th>
<th>then …</th>
</tr>
</thead>
<tbody>
<tr>
<td>cheap</td>
<td>$t_i &lt; a_i$</td>
<td>save $a_i - t_i$ credits</td>
</tr>
<tr>
<td>neutral</td>
<td>$t_i = a_i$</td>
<td></td>
</tr>
<tr>
<td>expensive</td>
<td>$t_i &gt; a_i$</td>
<td>spend $a_i - t_i$ previously saved credits</td>
</tr>
</tbody>
</table>

To show that $a_i$ is the amortized cost:
Show that we never run out of credits
Banker’s analysis of “two-stack” queue

\((L \text{ and } R)\)

Credits given \((a_i)\):

- enqueue: 2 credits
- dequeue: 1 credit

Actual cost \((t_i)\):

- enqueue: 1 credit – *save 1 credit to \(R)*!
- dequeue:
  - \(|L| > 0\): 1 credit
  - \(|L| = 0\): 1 + \(|R|\) credits – *spend the credits saved on \(R)*

So, both operations have ammortized \(O(1)\) cost
Outline

• Persistence
• Functional vs. imperative data structures
• Amortized complexity analysis
• Amortization for persistent data structures
Amortization and persistence

Bad news: if data structure is persistent, we can go into debt!

```haskell
q = foldr enqueue empty [1..5]  -- save 5 credits on R
r1 = dequeue q                 -- spend all credits on R
r2 = dequeue q                 -- spend all credits on R again!
```

Problem: lazy evaluation is working against us

Solution: make lazy evaluation work for us :-) 

Keys: structure data type and functions so that:

- expensive operations are memoized  
  buy them “on layaway”
- expensive operations can be “locally” paid for
Outline

• Persistence
• Functional vs. imperative data structures
• Amortized complexity analysis
• Amortization for persistent data structures
• Example: red-black trees
Red-black trees

A self-balancing binary search tree:

- every node is red or black
- leaves are valueless and black

Invariants:

- usual binary search tree invariant
- same # of black nodes on every root-to-leaf path
- every red node has two black children

Guarantee: longest path \( \leq 2 \times \) shortest path
Examples

Valid red-black trees:

Invalid red-black trees:
Insertion

Balance invariants

(1) same # of black nodes on every root-to-leaf path
(2) every red node has two black children

Strategy: will never violate (2)!

• always insert a red node
• if added after a black node, we’re done!
• else, “rebalance” to eliminate the red-red violation
  (may cause a new red-red violation, so recurse up the tree)
• set root to black
Rebalancing

After insert, four possible invalid cases:

If y’s parent is red, must rebalance again!