Rufous

Ben Simner
How to choose?

Queue
empty :: Q a
snoc :: Q a -> a -> Q a
head :: Q a -> a
tail :: Q a -> Q a

Possible Implementations...
• Linked List
• Banker’s Queue
• Physicist’s Queue
How to choose?

• Pick easiest to implement …
• Pick most complicated …
• Pick best complexity …
• Write benchmarks …
Rufous!

ADT → Generate Programs → Run Programs → Aggregate Results → Report
A Program

v0 = empty
v1 = snoc v0 1
v2 = snoc v0 2
v3 = snoc v1 3
o1 = head v1
o2 = head v2
o3 = head v3

main = print (o1 + o2 + o3)
A Program

\[ v_0 = \text{empty} \]
\[ v_1 = \text{snoc } v_0 1 \]
\[ v_2 = \text{snoc } v_0 2 \]
\[ v_3 = \text{snoc } v_1 3 \]
\[ o_1 = \text{head } v_1 \]
\[ o_2 = \text{head } v_2 \]
\[ o_3 = \text{head } v_3 \]

main = print \( (o_1 + o_2 + o_3) \)

The DUG:
Rufous API

Defining the ADT

class Queue q where

    empty :: q a
    snoc :: q a -> a -> q a
    head :: q a -> a
    tail :: q a -> q a
Rufous API

Defining the ADT

class Queue q where
    empty :: q a
    snoc :: q a -> a -> q a
    head :: q a -> a
    tail :: q a -> q a

Defining the Implementation

instance Queue [] where
    empty = []
    snoc xs x = xs ++ [x]
    head (x:_) = x
    tail (_:xs) = xs
Preconditions

Undefined Applications

- head empty
- tail empty
Preconditions

Undefined Applications
• head empty
• tail empty

Shadow Implementations

newtype ShadowQueue a = S Int

instance Queue ShadowQueue where
  empty = S 0
  snoc (S n) _ = S (n + 1)
  tail (S n) | n > 0 = S (n - 1)
  tail (S n) | n == 0 = guardFailed
  head (S n) | n > 0 = S n
  head (S n) | n == 0 = guardFailed
Running Rufous

<table>
<thead>
<tr>
<th></th>
<th>empty</th>
<th>head</th>
<th>snoc</th>
<th>tail</th>
<th>mortality</th>
<th>pmf</th>
<th>pof</th>
<th>ListQueue</th>
<th>BQueue</th>
<th>RQueue</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2823</td>
<td>751</td>
<td>1283</td>
<td>593</td>
<td>0.636</td>
<td>0.153</td>
<td>0.421</td>
<td>9.005ms</td>
<td>7.159ms</td>
<td>6.936ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>504</td>
<td>90</td>
<td>204</td>
<td>82</td>
<td>0.694</td>
<td>0.152</td>
<td>0.379</td>
<td>0.685ms</td>
<td>0.598ms</td>
<td>0.528ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10237</td>
<td>732</td>
<td>786</td>
<td>239</td>
<td>0.927</td>
<td>0.22</td>
<td>0.613</td>
<td>13.553ms</td>
<td>12.243ms</td>
<td>12.455ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4863</td>
<td>1691</td>
<td>4471</td>
<td>1662</td>
<td>0.48</td>
<td>0.069</td>
<td>0.265</td>
<td>22.809ms</td>
<td>41.630ms</td>
<td>59.152ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>77</td>
<td>23</td>
<td>57</td>
<td>21</td>
<td>0.571</td>
<td>0.072</td>
<td>0.172</td>
<td>0.128ms</td>
<td>0.135ms</td>
<td>0.108ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tabular output

- *pmf, pof* denote sharing (breadth)
- *mortality* denotes lifespan (depth)
- **RQueue** wins (as predicted by literature!)
Evaluation

Good
• Class-based API
• Fair generation of test programs
• Portable, single language implementation.

Bad
• Extracting DUGs from programs difficult
• Implementation inefficient
• Hard to use tabular output
Related Work

Auburn (2000) defined the DUG

• Very similar to Rufous.
• Older, no longer compiles.
• Not user-friendly.
The Future

Towards Full Automation!

Easier Extraction of DUGs from Programs
Profile and Select best structure as code is running.
References

• Queue implementations from “Chris Okasaki. Purely functional data structures. Cambridge University Press, 1999”.

• Auburn described in “Graeme E Moss. Benchmarking Purely Functional Data Structures, PhD thesis, University of York, 2000”