BRICS RS-96-37 Brodal & Okasaki: Optimal Purely Functional Priority Queues

BRICS

Basic Research in Computer Science

Optimal Purely Functional Priority Queues

Gerth Stølting Brodal
Chris Okasaki

BRICS Report Series RS-96-37

ISSN 0909-0878 October 1996

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through World Wide Web and anonymous FTP:

`http://www.brics.dk/`
`ftp://ftp.brics.dk/pub/BRICS`

Optimal Purely Functional Priority Queues

Gerth Stølting Brodal*
BRICS†, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark

Chris Okasaki‡
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA 15213

Abstract

Brodal recently introduced the first implementation of imperative priority queues to support *findMin*, *insert*, and *meld* in $O(1)$ worst-case time, and *deleteMin* in $O(\log n)$ worst-case time. These bounds are asymptotically optimal among all comparison-based priority queues. In this paper, we adapt Brodal's data structure to a purely functional setting. In doing so, we both simplify the data structure and clarify its relationship to the binomial queues of Vuillemin, which support all four operations in $O(\log n)$ time. Specifically, we derive our implementation from binomial queues in three steps: first, we reduce the running time of *insert* to $O(1)$ by eliminating the possibility of cascading links; second, we reduce the running time of *findMin* to $O(1)$ by adding a global root to hold the minimum element; and finally, we reduce the running time of *meld* to $O(1)$ by allowing priority queues to contain other priority queues. Each of these steps is expressed using ML-style functors. The last transformation, known as data-structural bootstrapping, is an interesting application of higher-order functors and recursive structures.

*Research partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II) and by the Danish Natural Science Research Council (Grant No. 9400044). E-mail: `gerth@daimi.aau.dk`.

†Basic Research in Computer Science, Centre of the Danish National Research Foundation.

‡Research supported by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. E-mail: `cokasaki@cs.cmu.edu`.

1 Introduction

Purely functional data structures differ from imperative data structures in at least two respects. First, many imperative data structures rely crucially on destructive assignments for efficiency, whereas purely functional data structures are forbidden from using destructive assignments. Second, purely functional data structures are automatically ***persistent*** (Driscoll *et al.*, 1989), meaning that, after an update, both the new and old versions of a data structure are available for further accesses and updates. In contrast, imperative data structures are almost always ***ephemeral***, meaning that, after an update, only the new version of a data structure is available. In many cases, these differences prevent functional programmers from simply using off-the-shelf data structures, such as those described in most algorithms texts. The design of efficient purely functional data structures is thus of great theoretical and practical interest to functional programmers, as well as to imperative programmers for those occasions when a persistent data structure is required. In this paper, we consider the design of an efficient purely functional priority queue.

The priority queue is a fundamental abstraction in computer programming, arguably surpassed in importance only by the dictionary and the sequence. Many implementations of priority queues have been proposed over the years; a small sampling includes (Williams, 1964; Crane, 1972; Vuillemin, 1978; Fredman & Tarjan, 1987; Brodal, 1996). However, all of these consider only imperative priority queues. Very little has been written about purely functional priority queues. To our knowledge, only Paulson (1991), Kaldewaij and Schoenmakers (1991), Schoenmakers (1992), and King (1994) have explicitly treated priority queues in a purely functional setting.

We consider priority queues that support the following operations:

findMin (q)	Return the minimum element of queue q.
insert (x, q)	Insert the element x into queue q.
meld (q_1, q_2)	Merge queues q_1 and q_2 into a single queue.
deleteMin (q)	Discard the minimum element of queue q.

In addition, priority queues supply a value ***empty*** representing the empty

```
signature ORDERED =
sig
   type T                              (* type of ordered elements *)
   val leq : T × T → bool              (* total ordering relation *)
end

signature PRIORITY_QUEUE =
sig
   structure Elem : ORDERED

   type T                              (* type of priority queues *)

   val empty     : T
   val isEmpty   : T → bool

   val insert    : Elem.T × T → T
   val meld      : T × T → T

   exception EMPTY

   val findMin   : T → Elem.T          (* raises EMPTY if queue is empty *)
   val deleteMin : T → T               (* raises EMPTY if queue is empty *)
end
```

Figure 1: Signature for priority queues.

queue and a predicate *isEmpty*. For simplicity, we will ignore empty queues except when presenting actual code. Figure 1 displays a Standard ML signature for these priority queues.

Brodal (1995) recently introduced the first imperative data structure to support all these operations in $O(1)$ worst-case time except *deleteMin*, which requires $O(\log n)$ worst-case time. Several previous implementations, most notably Fibonacci heaps (Fredman & Tarjan, 1987), had achieved these bounds, but in an amortized, rather that worst-case, sense. It is easy to show by reduction to sorting that these bounds are asymptotically optimal among all comparison-based priority queues — the bound on *deleteMin* cannot be decreased without simultaneously increasing the bounds on *findMin*, *insert*, and/or *meld*.

It is reasonably straightforward to adapt Brodal's data structure to a purely functional setting by combining the recursive-slowdown technique of Kaplan and Tarjan (1995) with a purely functional implementation of double-ended queues (Hood, 1982; Okasaki, 1995c).

However, this approach suffers from at least two defects, one practical and one pedagogical. First, both recursive slowdown and double-ended queues carry non-trivial overheads, so the resulting data structure is quite slow in practice (even though asymptotically optimal). Second, the resulting design is difficult to explain and understand. The design choices are intermingled, and it is difficult to see the purpose and contribution of each. Furthermore, the relationship to other priority queue designs is obscured.

For these reasons, we take an indirect approach to adapting Brodal's data structure. First, we isolate the design choices in Brodal's data structure and rethink each in a functional, rather than imperative, environment. This allows us to replace recursive slowdown with a simpler technique borrowed from the random-access lists of Okasaki (1995b) and to eliminate the need for double-ended queues altogether. Then, starting from a well-known antecedent — the binomial queues of Vuillemin (1978) — we reintroduce each modification, one at a time. This both simplifies the data structure and clarifies its relationship to other priority queue designs.

We begin by reviewing binomial queues, which support all four major operations in $O(\log n)$ time. We then derive our data structure from binomial queues in three steps. First, we describe a variant of binomial queues, called *skew binomial queues*, that reduces the running time of *insert* to $O(1)$ by eliminating the possibility of cascading links. Second, we reduce the running time of *findMin* to $O(1)$ by adding a global root to hold the minimum element. Third, we apply a technique of Buchsbaum *et al.* (Buchsbaum *et al.*, 1995; Buchsbaum & Tarjan, 1995) called *data-structural bootstrapping*, which reduces the running time of *meld* to $O(1)$ by allowing priority queues to contain other priority queues. Each of these steps is expressed using ML-style functors. The last transformation, data-structural bootstrapping, is an interesting application of higher-order functors and recursive structures. After describing a few possible optimizations, we conclude with brief discussions of related work and future work.

All source code is presented in Standard ML (Milner *et al.*, 1990) and is available through the World Wide Web from

`http://foxnet.cs.cmu.edu/people/cokasaki/priority.html`

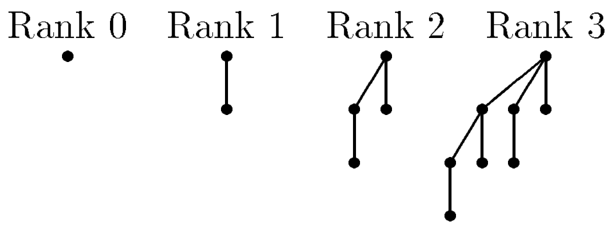

Figure 2: Binomial trees of ranks 0–3.

2 Binomial Queues

Binomial queues are an elegant form of priority queue introduced by Vuillemin (1978) and extensively studied by Brown (1978). Although they considered binomial queues only in an imperative setting, King (1994) has shown that binomial queues work equally well in a functional setting. In this section, we briefly review binomial queues — see King (1994) for more details.

Binomial queues are composed of more primitive objects known as binomial trees. Binomial trees are inductively defined as follows:

- A binomial tree of rank 0 is a singleton node.
- A binomial tree of rank $r + 1$ is formed by ***linking*** two binomial trees of rank r, making one tree the leftmost child of the other.

From this definition, it is easy to see that a binomial tree of rank r contains exactly 2^r nodes. There is a second, equivalent definition of binomial trees that is sometimes more convenient: a binomial tree of rank r is a node with r children $t_1 \dots t_r$, where each t_i is a binomial tree of rank $r - i$. Figure 2 illustrates several binomial trees of varying rank.

Assuming a total ordering on nodes, a binomial tree is said to be ***heap-ordered*** if every node is $\leq$ each of its descendants. To preserve heap order when linking two heap-ordered binomial trees, we make the tree with the larger root a child of the tree with the smaller root, with ties broken arbitrarily.

A binomial queue is a forest of heap-ordered binomial trees where no two trees have the same rank. Because binomial trees have sizes of the form 2^r, the ranks of the trees in a binomial queue of size n are distributed according to the ones in the binary representation of n. For example, consider a binomial queue of size 21. The binary representation of 21 is 10101, and the binomial queue contains trees of ranks 0, 2, and 4 (of sizes 1, 4, and 16, respectively). Note that a binomial queue of size n contains at most $\lfloor \log_2(n+1) \rfloor$ trees.

We are now ready to describe the operations on binomial queues. Since all the trees in a binomial queue are heap-ordered, we know that the minimum element in a binomial queue is the root of one of the trees. We can find this minimum element in $O(\log n)$ time by scanning through the roots. To insert a new element into a queue, we first create a new singleton tree (i.e., a binomial tree of rank 0). We then step through the existing trees in increasing order of rank until we find a missing rank, linking trees of equal rank as we go. Inserting an element into a binomial queue corresponds precisely to adding one to a binary number, with each link corresponding to a carry. The worst case is insertion into a queue of size $n = 2^k - 1$, requiring a total of k links and $O(\log n)$ time. The analogy to binary addition also applies to melding two queues. We step through the trees of both queues in increasing order of rank, linking trees of equal rank as we go. Once again, each link corresponds to a carry. This also requires $O(\log n)$ time.

The trickiest operation is *deleteMin*. We first find the tree with the minimum root and remove it from the queue. We discard the root, but then must return its children to the queue. However, the children themselves constitute a valid binomial queue (i.e., a forest of heap-ordered binomial trees with no two trees of the same rank), and so may be melded with the remaining trees of the queue. Both finding the tree to remove and returning the children to the queue require $O(\log n)$ time, for a total of $O(\log n)$ time.

Figure 3 gives an implementation of binomial queues as a Standard ML functor that takes a structure specifying a type of ordered elements and produces a structure of priority queues containing elements of the specified type. Two aspects of this implementation deserve further explanation. First, the conflicting requirements of ***insert*** and ***link*** lead to a confusing inconsistency, common to virtually all im-

```
functor BinomialQueue (E : ORDERED) : PRIORITY_QUEUE =
struct
   structure Elem = E

   type Rank = int
   datatype Tree = Node of Elem.T × Rank × Tree list
   type T = Tree list

   (* auxiliary functions *)
   fun root (Node (x,r,c)) = x
   fun rank (Node (x,r,c)) = r
   fun link (t1 as Node (x1,r1,c1), t2 as Node (x2,r2,c2)) =   (* r1 = r2 *)
         if Elem.leq (x1, x2) then Node (x1,r1+1,t2 :: c1) else Node (x2,r2+1,t1 :: c2)
   fun ins (t, [ ]) = [t]
     | ins (t, t' :: ts) =   (* rank t ≤ rank t' *)
         if rank t < rank t' then t :: t' :: ts else ins (link (t, t'), ts)

   val empty = [ ]
   fun isEmpty ts = null ts

   fun insert (x, ts) = ins (Node (x,0,[ ]), ts)
   fun meld ([ ], ts) = ts
     | meld (ts, [ ]) = ts
     | meld (t1 :: ts1, t2 :: ts2) =
         if rank t1 < rank t2 then t1 :: meld (ts1, t2 :: ts2)
         else if rank t2 < rank t1 then t2 :: meld (t1 :: ts1, ts2)
         else ins (link (t1, t2), meld (ts1, ts2))

   exception EMPTY

   fun findMin [ ] = raise EMPTY
     | findMin [t] = root t
     | findMin (t :: ts) =
         let val x = findMin ts
         in if Elem.leq (root t, x) then root t else x end
   fun deleteMin [ ] = raise EMPTY
     | deleteMin ts =
         let fun getMin [t] = (t, [ ])
               | getMin (t :: ts) =
                   let val (t', ts') = getMin ts
                   in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
             val (Node (x,r,c), ts) = getMin ts
         in meld (rev c, ts) end
end
```

Figure 3: A functor implementing binomial queues.

plementations of binomial queues. The trees in binomial queues are maintained in increasing order of rank to support the ***insert*** operation efficiently. On the other hand, the children of binomial trees are maintained in decreasing order of rank to support the ***link*** operation efficiently. This discrepancy compels us to reverse the children of the deleted node during a *deleteMin*. Second, for clarity, every node contains its rank. In a realistic implementation, however, only the roots would store their ranks. The ranks of all other nodes are uniquely determined by the ranks of their parents and their positions among their siblings. King (1994) describes an alternative representation that eliminates all ranks, at the cost of introducing placeholders for those ranks corresponding to the zeros in the binary representation of the size of the queue.

3 Skew Binomial Queues

In this section, we describe a variant of binomial queues, called ***skew binomial queues***, that supports insertion in $O(1)$ worst-case time. The problem with binomial queues is that inserting a single element into a queue might result in a long cascade of links, just as adding one to a binary number might result in a long cascade of carries. We can reduce the cost of an insert to at most a single link by borrowing a technique from random-access lists (Okasaki, 1995b). Random-access lists are based on a variant number system, called skew binary numbers (Myers, 1983), in which adding one causes at most a single carry.

In skew binary numbers, the kth digit represents $2^{k+1} - 1$, rather than 2^k as in ordinary binary numbers. Every digit is either zero or one, except that the lowest non-zero digit may be two. For instance, 92 is written $\mathsf{002101}$ (least-significant digit first). A carry occurs when adding one to a number whose lowest non-zero digit is two. For instance, $\mathsf{1} + \mathsf{002101} = \mathsf{000201}$. Because the next higher digit is guaranteed not to be two, only a single carry is ever necessary.

Just as binomial queues are composed of binomial trees, skew binomial queues are composed of skew binomial trees. Skew binomial trees are inductively defined as follows:

- A skew binomial tree of rank 0 is a singleton node.

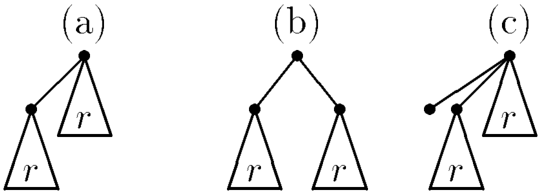

Figure 4: The three methods of constructing a skew binomial tree of rank $r+1$. (a) a simple link. (b) a type A skew link. (c) a type B skew link.

- A skew binomial tree of rank $r+1$ is formed in one of three ways:
  - a ***simple link***, making a skew binomial tree of rank r the leftmost child of another skew binomial tree of rank r;
  - a ***type A skew link***, making two skew binomial trees of rank r the children of a skew binomial tree of rank 0; or
  - a ***type B skew link***, making a skew binomial tree of rank 0 and a skew binomial tree of rank r the leftmost children of another skew binomial tree of rank r.

Figure 4 illustrates the three kinds of links. Note that type A and type B skew links are equivalent when $r=0$. Ordinary binomial trees and perfectly balanced binary trees are special cases of skew binomial trees obtained by allowing only simple links and type A skew links, respectively. A skew binomial tree of rank r constructed entirely with skew links (type A or type B) contains exactly $2^{r+1}-1$ nodes, but, in general, the size of a skew binomial tree t of rank r is bounded by $2^r \le |t| \le 2^{r+1}-1$. In addition, the height of a skew binomial tree is equal to its rank. Once again, there is a second, equivalent definition: a skew binomial tree of rank $r>0$ is a node with up to $2k$ children $s_1t_1 \ldots s_kt_k$ $(1 \le k \le r)$, where each t_i is a skew binomial tree of rank $r-i$ and each s_i is a skew binomial tree of rank 0, except that s_k has rank $r-k$ (which is 0 only when $k=r$). Every s_i is optional except that s_k is optional only when $k=r$. Although somewhat confusing, this definition arises naturally from the three methods of constructing a tree. Every s_kt_k pair is produced by a type A skew link, and every

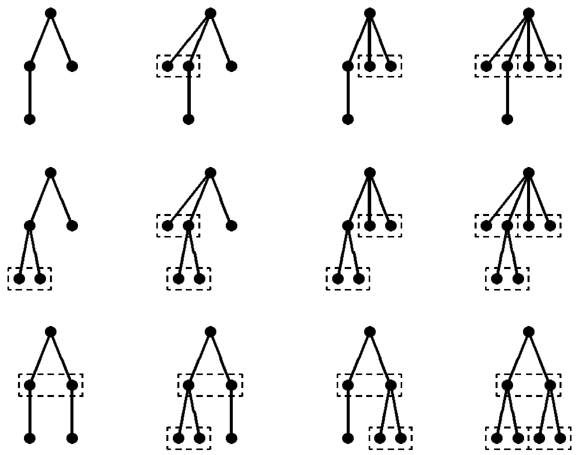

Figure 5: The twelve possible shapes of skew binomial trees of rank 2. Dashed boxes surround each $s_i t_i$ pair.

$s_i t_i$ pair ($i < k$) is produced by a type B skew link. Every t_i without a corresponding s_i is produced by a simple link. Unlike ordinary binomial trees, skew binomial trees may have many different shapes. For example, the twelve possible shapes of skew binomial trees of rank 2 are shown in Figure 5.

A skew binomial tree is heap-ordered if every node is $\leq$ each of its descendants. To preserve heap order during a simple link, we make the tree with the larger root a child of the tree with the smaller root. During a skew link, we make the two trees with larger roots children of the tree with the smallest root. We perform a type A skew link if the rank 0 tree has the smallest root, and a type B skew link if one of the rank r trees has the smallest root.

A skew binomial queue is a forest of heap-ordered skew binomial trees where no two trees have the same rank, except possibly the two smallest ranked trees. Since skew binomial trees of the same rank may have different sizes, there may be several ways to distribute the ranks for a queue of any particular size. For example, a skew binomial queue of size 4 may contain one rank 2 tree of size 4; two rank 1 trees, each of size 2; a rank 1 tree of size 3 and a rank 0 tree; or a rank 1 tree of size 2 and two rank 0 trees. However, the maximum number of trees in a queue is still $O(\log n)$.

We are now ready to describe the operations on skew binomial queues. The ***findMin*** and ***meld*** operations are almost unchanged. To find the minimum element in a skew binomial queue, we simply scan through the roots, taking $O(\log n)$ time. To meld two queues, we step through the trees of both queues in increasing order of rank, performing a simple link (not a skew link!) whenever we find two trees of equal rank. Once again, this requires $O(\log n)$ time.

The big advantage of skew binomial queues over ordinary binomial queues is that we can now insert a new element in $O(1)$ time. We first create a new singleton tree (i.e., a skew binomial tree of rank 0). We then check the ranks of the two smallest trees in the queue. If both trees have rank r, then we skew link these two trees with the new rank 0 tree to get a new rank $r+1$ tree. We know that there can be no more than one existing rank $r+1$ tree, and that this is the smallest rank in the new queue, so we simply add the new tree to the queue. If the two smallest trees in the queue have different ranks, then we simply add the new rank 0 tree to the queue. Since there was at most one existing tree of rank 0, the new queue contains at most two trees of the smallest rank. In either case, we are done.

Again, ***deleteMin*** is the most complicated operation. We first find and remove the tree with the minimum root. After discarding the root, we partition its children into two groups, those with rank 0 and those with rank > 0. Other than s_k and t_k, every s_i has rank 0 and every t_i has rank > 0. The ranks of s_k and t_k are both 0 when $k = r$ and both > 0 when $k < r$. Note that every rank 0 child contains a single element. The children with rank > 0 constitute a valid skew binomial queue, so we meld these children with the remaining trees in the queue. Finally, we reinsert each of the rank 0 children. Each of these steps requires $O(\log n)$ time, so the total time required is $O(\log n)$.

Figures 6 and 7 present an implementation of skew binomial queues as a Standard ML functor. Like the binomial queue functor, this functor takes a structure specifying a type of ordered elements and produces a structure of priority queues containing elements of the specified type. Once again, lists of trees are maintained in different orders for different purposes. The trees in a queue are maintained in increasing order of rank (except that the first two trees may have the same rank), but the children of skew binomial trees are maintained in a more complicated

```
functor SkewBinomialQueue (E : ORDERED) : PRIORITY_QUEUE =
struct
  structure Elem = E

  type Rank = int
  datatype Tree = Node of Elem.T × Rank × Tree list
  type T = Tree list

  (* auxiliary functions *)
  fun root (Node (x,r,c)) = x
  fun rank (Node (x,r,c)) = r
  fun link (t1 as Node (x1,r1,c1), t2 as Node (x2,r2,c2)) =   (* r1 = r2 *)
        if Elem.leq (x1,x2) then Node (x1,r1+1,t2 :: c1) else Node (x2,r2+1,t1 :: c2)
  fun skewLink (t0 as Node (x0,r0,_), t1 as Node (x1,r1,c1), t2 as Node (x2,r2,c2)) =
        if Elem.leq (x1,x0) andalso Elem.leq (x1,x2) then Node (x1,r1+1,t0 :: t2 :: c1)
        else if Elem.leq (x2,x0) andalso Elem.leq (x2,x1) then Node (x2,r2+1,t0 :: t1 :: c2)
        else Node (x0,r1+1,[t1, t2])
  fun ins (t, [ ]) = [t]
    | ins (t, t' :: ts) =   (* rank t ≤ rank t' *)
        if rank t < rank t' then t :: t' :: ts else ins (link (t, t'), ts)

  fun uniqify [ ] = [ ]
    | uniqify (t :: ts) = ins (t, ts)   (* eliminate initial duplicate *)
  fun meldUniq ([ ], ts) = ts
    | meldUniq (ts, [ ]) = ts
    | meldUniq (t1 :: ts1, t2 :: ts2) =
        if rank t1 < rank t2 then t1 :: meldUniq (ts1, t2 :: ts2)
        else if rank t2 < rank t1 then t2 :: meldUniq (t1 :: ts1, ts2)
        else ins (link (t1, t2), meldUniq (ts1, ts2))

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

Figure 6: A functor implementing skew binomial queues (part I).

```
  fun insert (x, ts as t1 :: t2 :: rest) =
        if rank t1 = rank t2 then skewLink (Node (x,0,[ ]),t1,t2) :: rest
        else Node (x,0,[ ]) :: ts
    | insert (x, ts) = Node (x,0,[ ]) :: ts
  fun meld (ts, ts') = meldUniq (uniqify ts, uniqify ts')

  exception EMPTY

  fun findMin [ ] = raise EMPTY
    | findMin [t] = root t
    | findMin (t :: ts) =
        let val x = findMin ts
        in if Elem.leq (root t, x) then root t else x end
  fun deleteMin [ ] = raise EMPTY
    | deleteMin ts =
        let fun getMin [t] = (t, [ ])
              | getMin (t :: ts) =
                  let val (t', ts') = getMin ts
                  in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
            fun split (ts,xs,[ ]) = (ts, xs)
              | split (ts,xs,t :: c) =
                  if rank t = 0 then split (ts,root t :: xs,c) else split (t :: ts,xs,c)
            val (Node (x,r,c), ts) = getMin ts
            val (ts',xs') = split ([ ],[ ],c)
        in fold insert xs' (meld (ts, ts')) end
end
```

Figure 7: A functor implementing skew binomial queues (part II).

order. The t_i children are maintained in decreasing order of rank, but they are interleaved with the s_i children, which have rank 0 (except s_k, which has rank $r - k$). Furthermore, recall that each s_i is optional (except that s_k is optional only if $k = r$).

4 Adding a Global Root

We next describe a simple module-level transformation on priority queues to reduce the running time of ***findMin*** to $O(1)$. Although this transformation can be applied to any priority queue module, it is only useful on priority queues for which ***findMin*** requires more than $O(1)$ time.

Most implementations of priority queues represent a queue as a single heap-ordered tree so that the minimum element can always be found at the root in $O(1)$ time. Unfortunately, binomial queues and skew binomial queues represent a queue as a forest of heap-ordered trees, so finding the minimum element requires scanning all the roots in the forest. However, we can convert this forest into a single heap-ordered tree, thereby supporting ***findMin*** in $O(1)$ time, by simply adding a global root to hold the minimum element. In general, this tree will not be a binomial or skew binomial tree, but this is irrelevant since the global root will be treated separately from the rest of the queue. The details of this transformation are quite routine, but we present them anyway as a warm-up for the more complicated transformation in the next section.

Given some type P_α of primitive priority queues containing elements of type α, we define the type of rooted priority queues RP_α to be

$$RP_\alpha = \{\textbf{\textit{empty}}\} + (\alpha \times P_\alpha)$$

In other words, a rooted priority queue is either empty or a pair of a single element (the root) and a primitive priority queue. We maintain the invariant that the minimum element of any non-empty priority queue is at the root. For each operation f on priority queues, let f

```
functor AddRoot (Q : PRIORITY_QUEUE) : PRIORITY_QUEUE =
struct
   structure Elem = Q.Elem

   datatype T = Empty | Root of Elem.T × Q.T

   val empty = Empty
   fun isEmpty Empty = true
     | isEmpty (Root _) = false

   fun insert (y, Empty) = Root (y, Q.empty)
     | insert (y, Root (x, q)) =
         if Elem.leq (y, x) then Root (y, Q.insert (x, q)) else Root (x, Q.insert (y, q))
   fun meld (Empty, rq) = rq
     | meld (rq, Empty) = rq
     | meld (Root (x1, q1), Root (x2, q2)) =
         if Elem.leq (x1, x2) then Root (x1, Q.insert (x2, Q.meld (q1, q2)))
                              else  Root (x2, Q.insert (x1, Q.meld (q1, q2)))

   exception EMPTY

   fun findMin Empty = raise EMPTY
     | findMin (Root (x, q)) = x
   fun deleteMin Empty = raise EMPTY
     | deleteMin (Root (x, q)) =
         if Q.isEmpty q then Empty else Root (Q.findMin q, Q.deleteMin q)
end
```

Figure 8: A functor for adding a global root to existing priority queues.

and f' indicate the operations on P_α and RP_α, respectively. Then,

$$
\begin{array}{lcll}
\boldsymbol{findMin}'\,(\langle x, q\rangle) & = & x & \\
\boldsymbol{insert}'\,(y, \langle x, q\rangle) & = & \langle x, \boldsymbol{insert}\,(y, q)\rangle & \text{if } x \leq y \\
\boldsymbol{insert}'\,(y, \langle x, q\rangle) & = & \langle y, \boldsymbol{insert}\,(x, q)\rangle & \text{if } y < x \\
\boldsymbol{meld}'\,(\langle x_1, q_1\rangle, \langle x_2, q_2\rangle) & = & \langle x_1, \boldsymbol{insert}\,(x_2, \boldsymbol{meld}\,(q_1, q_2))\rangle & \text{if } x_1 \leq x_2 \\
\boldsymbol{meld}'\,(\langle x_1, q_1\rangle, \langle x_2, q_2\rangle) & = & \langle x_2, \boldsymbol{insert}\,(x_1, \boldsymbol{meld}\,(q_1, q_2))\rangle & \text{if } x_2 < x_1 \\
\boldsymbol{deleteMin}'\,(\langle x, q\rangle) & = & \langle \boldsymbol{findMin}\,(q), \boldsymbol{deleteMin}\,(q)\rangle &
\end{array}
$$

In Figure 8, we present this transformation as a Standard ML functor that takes a priority queue structure and produces a new structure incorporating this optimization. When applied to the skew binomial queues of the previous section, this tranformation produces a priority queue that supports both ***insert*** and ***findMin*** in $O(1)$ time. However, ***meld*** and ***deleteMin*** still require $O(\log n)$ time.

If a program requires several priority queues with different element types, it may be more convenient to implement this transformation as a higher-order functor (MacQueen & Tofte, 1994). First-order functors can only take and return structures, but higher-order functors can take and return other functors as well. Although the definition of Standard ML (Milner ***et al.***, 1990) describes only first-order functors, some implementations of Standard ML, notably Standard ML of New Jersey, support higher-order functors.

A priority queue functor, such as ***BinomialQueue*** or ***SkewBinomialQueue***, is one that takes a structure specifying a type of ordered elements and returns a structure of priority queues containing elements of the specified type. The following higher-order functor takes a priority queue functor and returns a priority queue functor incorporating the ***AddRoot*** optimization.

```
functor AddRootToFun (functor MakeQ (E : ORDERED) :
                              sig
                                 include PRIORITY_QUEUE
                                 sharing Elem = E
                              end)
                      (E : ORDERED) : PRIORITY_QUEUE =
        AddRoot (MakeQ (E))
```

Note that this functor is curried, so although it appears to take two arguments, it actually takes one argument (***MakeQ***) and returns a functor that takes the second argument (***E***). The sharing constraint is necessary to ensure that the functor ***MakeQ*** returns a priority queue with the desired element type. Without the sharing constraint, ***MakeQ*** might ignore ***E*** and return a priority queue structure with some arbitrary element type.

Now, if we need both a string priority queue and an integer priority queue, we can write

```
functor RootedSkewBinomialQueue =
        AddRootToFun (functor MakeQ = SkewBinomialQueue)
structure StringQueue = RootedSkewBinomialQueue (StringElem)
structure IntQueue    = RootedSkewBinomialQueue (IntElem)
```

where ***StringElem*** and ***IntElem*** match the ***ORDERED*** signature and define the desired orderings over strings and integers, respectively.

5 Bootstrapping Priority Queues

Finally, we improve the running time of ***meld*** to $O(1)$ by applying a technique of Buchsbaum ***et al.*** (Buchsbaum ***et al.***, 1995; Buchsbaum & Tarjan, 1995) called ***data-structural bootstrapping.*** The basic idea is to reduce melding to simple insertion by using priority queues that contain other priority queues. Then, to meld two priority queues, we simply insert one priority queue into the other.

As in the previous section, we describe bootstrapping as a module-level transformation on priority queues. Let P_α be the type of primitive priority queues containing elements of type α. We wish to construct the type BP_α of bootstrapped priority queues containing elements of type α. A bootstrapped priority queue will be a primitive priority queue whose "elements" are other bootstrapped priority queues. As a first attempt, we consider

$$BP_\alpha = P_{P_\alpha}$$

Here we have applied a single level of bootstrapping. However, this simple solution does not work because the elements of the top-level primitive priority queue have the wrong type — they are simple primitive priority queues rather than bootstrapped priority queues. Clearly, we need to apply the idea of bootstrapping recursively, as in

$$BP_\alpha = P_{BP_\alpha}$$

Unfortunately, this solution offers no place to store simple elements. We therefore borrow from the previous section and add a root to every primitive priority queue.

$$BP_\alpha = \alpha \times P_{BP_\alpha}$$

Thus, a bootstrapped priority queue is a simple element (which should be the minimum element in the queue) paired with a primitive priority queue containing other bootstrapped priority queues ordered by their respective minimums. Since bootstrapping adds a root to every primitive priority queue, the bootstrapping transformation subsumes the

AddRoot transformation. Finally, we must allow for the possibility of an empty queue. The final definition is thus

$$BP_\alpha = \{\textbf{\textit{empty}}\} + R_\alpha \text{ where } R_\alpha = \alpha \times P_{R_\alpha}$$

Note that the primitive priority queues contain only non-empty bootstrapped priority queues as elements.

Now, each of the operations on bootstrapped priority queues can be defined in terms of the operations on the primitive priority queues. For each operation f on priority queues, let f and f' indicate the operations on P_{R_α} and BP_α, respectively. Then,

$$\begin{array}{lcll}
\textbf{\textit{findMin}}'\,(\langle x, q\rangle) & = & x & \\
\textbf{\textit{insert}}'\,(x, q) & = & \textbf{\textit{meld}}'\,(\langle x, \textbf{\textit{empty}}\rangle, q) & \\
\textbf{\textit{meld}}'\,(\langle x_1, q_1\rangle, \langle x_2, q_2\rangle) & = & \langle x_1, \textbf{\textit{insert}}\,(\langle x_2, q_2\rangle, q_1)\rangle & \text{if } x_1 \leq x_2 \\
\textbf{\textit{meld}}'\,(\langle x_1, q_1\rangle, \langle x_2, q_2\rangle) & = & \langle x_2, \textbf{\textit{insert}}\,(\langle x_1, q_1\rangle, q_2)\rangle & \text{if } x_2 < x_1 \\
\textbf{\textit{deleteMin}}'\,(\langle x, q\rangle) & = & \langle y, \textbf{\textit{meld}}(q_1, q_2)\rangle & \\
& & \quad \text{where } \langle y, q_1\rangle \;=\; \textbf{\textit{findMin}}\,(q) & \\
& & \quad \phantom{\text{where } \langle y,} q_2 \;=\; \textbf{\textit{deleteMin}}\,(q) &
\end{array}$$

Next, we consider the efficiency of bootstrapped priority queues. Since the minimum element is stored at the root, ***findMin*** requires $O(1)$ time regardless of the underlying implementation. The ***insert*** and ***meld*** operations depend only on the ***insert*** of the primitive implementation. By bootstrapping a priority queue with $O(1)$ insertion, such as the skew binomial queues of Section 3, we obtain both $O(1)$ insertion and $O(1)$ melding. Finally, ***deleteMin*** on bootstrapped priority queues depends on ***findMin***, ***meld***, and ***deleteMin*** from the underlying implementation. Since skew binomial queues support each of these in $O(\log n)$ time, ***deleteMin*** on bootstrapped skew binomial queues also requires $O(\log n)$ time.

In summary, bootstrapped skew binomial queues support every operation in $O(1)$ time except ***deleteMin***, which requires $O(\log n)$ time. It is easy to show by reduction to sorting that these bounds are optimal among all comparison-based priority queues. Other tradeoffs between the running times of the various operations are also possible, but no comparison-based priority queue can support ***insert*** in better than $O(\log n)$ worst-case time or ***meld*** in better than $O(n)$ worst-case time unless one of *findMin* or *deleteMin* takes at least $O(\log n)$ worst-case time (Brodal, 1995).

The bootstrapping process can be elegantly expressed in Standard ML extended with higher-order functors and recursive structures, as shown in Figure 9. The higher-order nature of *Bootstrap* is analogous to the higher-order nature of *AddRootToFun*, while the recursion between *RootedQ* and Q captures the recursion between R_α and P_{R_α}. Unfortunately, although some implementations of Standard ML support higher-order functors (MacQueen & Tofte, 1994), none support recursive structures, so the recursion between *RootedQ* and Q is forbidden. In fact, there are good reasons for not supporting recursion like this in general. For instance, this recursion may not even be sensible if *MakeQ* can have computational effects! However, many priority queue functors, such as *SkewBinomialQueue*, simply define a few datatypes and functions, and have no computational effects. For these well-behaved functors, the recursion between *RootedQ* and Q does appear to be sensible, and it would be pleasant to be able to bootstrap these functors in this manner.

Without recursive structures, we can still implement bootstrapped priority queues, but much less cleanly. We manually specialize *Bootstrap* to each desired primitive priority queue by inlining the appropriate priority queue functor for *MakeQ* and eliminating Q and *RootedQ* as separate structures. This reduces the recursion on structures to recursion on datatypes, which is easily supported by Standard ML. Of course, as with any manual program transformation, this process is tedious and error-prone.

```
functor Bootstrap (functor MakeQ (E : ORDERED) : sig
                                                  include PRIORITY_QUEUE
                                                  sharing Elem = E
                                                end)
                 (E : ORDERED) : PRIORITY_QUEUE =
struct
  structure Elem = E

  (* recursive structures not supported in SML! *)
  structure rec RootedQ =
    struct
      datatype T = Root of Elem.T × Q.T
      fun leq (Root (x1, q1), Root (x2, q2)) = Elem.leq (x1, x2)
    end
  and Q = MakeQ (RootedQ)

  open RootedQ   (* expose Root constructor *)

  datatype T = Empty | NonEmpty of RootedQ.T

  val empty = Empty
  fun isEmpty Empty = true
    | isEmpty (NonEmpty _) = false

  fun insert (x, xs) = meld (NonEmpty (Root (x, Q.empty)), xs)
  and meld (Empty, xs) = xs
    | meld (xs, Empty) = xs
    | meld (NonEmpty (r1 as Root (x1, q1)), NonEmpty (r2 as Root (x2, q2))) =
        if Elem.leq (x1, x2) then NonEmpty (Root (x1, Q.insert (r2, q1)))
                             else  NonEmpty (Root (x2, Q.insert (r1, q2)))

  exception EMPTY

  fun findMin Empty = raise EMPTY
    | findMin (NonEmpty (Root (x, q))) = x
  fun deleteMin Empty = raise EMPTY
    | deleteMin (NonEmpty (Root (x, q))) =
        if Q.isEmpty q then Empty
        else let val (Root (y, q1)) = Q.findMin q
                 val q2 = Q.deleteMin q
             in NonEmpty (Root (y, Q.meld (q1, q2))) end
end
```

Figure 9: A higher-order functor for bootstrapping priority queues.

6 Optimizations

Although bootstrapped skew binomial queues as described in the previous section are asymptotically optimal, there are still further optimizations we can make. Consider the type of priority queues resulting from inlining *SkewBinomialQueue* for *MakeQ*:

$$\begin{aligned}
&\text{datatype } \mathit{Tree} = \mathit{Node} \text{ of } \mathit{Root} \times \mathit{Rank} \times \mathit{Tree} \text{ list} \\
&\quad\;\; \text{and } \mathit{Root} = \mathit{Root} \text{ of } \mathit{Elem.T} \times \mathit{Tree} \text{ list} \\
&\text{datatype } T \quad\;\, = \mathit{Empty} \mid \mathit{NonEmpty} \text{ of } \mathit{Root}
\end{aligned}$$

In this representation, a node has the form $\mathit{Node}(\mathit{Root}(x, f), r, c)$, where x is an element, f is a list of trees representing a forest, r is a rank, and c is a list of trees representing the children of the node. Since every node contains both x and f we can flatten the representation of nodes to be

$$\text{datatype } \mathit{Tree} = \mathit{Node} \text{ of } \mathit{Elem.T} \times \mathit{Tree} \text{ list} \times \mathit{Rank} \times \mathit{Tree} \text{ list}$$

In many implementations, this will eliminate an indirection on every access to x.

Next, note that f is completely ignored until its root is deleted. Thus, we do not require direct access to f and can in fact store it at the tail of c, combining the two into a single list representing $c +\!\!+ f$. This leads to the following representation, which usually saves a word of storage at every node:

$$\text{datatype } \mathit{Tree} = \mathit{Node} \text{ of } \mathit{Elem.T} \times \mathit{Rank} \times \mathit{Tree} \text{ list}$$

In this representation, it is necessary to traverse c during *deleteMin* to access f, but we need to traverse c anyway to extract the rank 0 children and reverse the remaining children. Given a rank r node, determining where c ends and f begins is usually quite easy. If $r = 0$, then $c = [\,]$. If $r = 1$, then c consists of either one or two rank 0 nodes. If $r > 1$, then c ends with either a pair of nodes of the same non-zero rank or a rank 1 node followed by one or two rank 0 nodes. The only ambiguities involve rank 0 nodes: it is sometimes impossible to distinguish the case where c ends with two rank 0 nodes from the case where c ends with a single rank 0 node and f begins with a rank

0 node. However, in every such situation, it does no harm to treat the ambiguous node as if it were part of c rather than f.

As a final simplification, note that the distinction between trees and roots is unnecessary, since every root can be treated as a tree of rank 0. Our final representation is then

$$\begin{array}{lll} \text{datatype } \mathit{Tree} & = & \mathit{Node} \text{ of } \mathit{Elem.T} \times \mathit{Rank} \times \mathit{Tree} \text{ list} \\ \text{datatype } T & = & \mathit{Empty} \mid \mathit{NonEmpty} \text{ of } \mathit{Tree} \end{array}$$

This increases the size of every root slightly, but also eliminates some minor copying during melds.

7 Related Work

Although there is an enormous literature on imperative priority queues, there has been very little work on purely functional priority queues.

Paulson (1991) describes a (non-meldable) priority queue combining the techniques of implicit heaps (Williams, 1964), which traditionally are implemented using arrays, with a balanced-tree representation of arrays supporting extension at the rear. Hoogerwoord (1992) represents arrays using the same trees as Paulson, but also allows the arrays to be extended at the front. A variant of Paulson's queues, using the slightly simpler front-extension of Hoogerwoord, appears to be part of the functional programming folklore.

King (1994) presents a purely functional implementation of binomial queues. Although binomial queues are considered to be rather complicated in imperative settings (Jones, 1986), King demonstrates that the more convenient list-processing capabilities of functional languages support binomial queues quite elegantly.

Schoenmakers (1992), extending earlier work with Kaldewaij (1991), uses functional notation to aid in the derivation of amortized bounds for a number of data structures, including three priority queues: skew heaps[1] (Sleator & Tarjan, 1986), Fibonacci heaps (Fredman & Tarjan, 1987), and pairing heaps (Fredman *et al.*, 1986). Schoenmakers also discusses splay trees (Sleator & Tarjan, 1985), a form of self-adjusting

[1]Note that the "skew" in skew heaps is completely unrelated to the "skew" in skew binomial queues.

binary search tree that has been shown by Jones (1986) to be particularly effective as a non-meldable priority queue. Each of these four data structures is efficient only in the amortized sense. Although he uses functional notation, Schoenmakers restricts his attention to ephemeral uses of data structures, where only the most recent version of a data structure may be accessed or updated. Ephemerality is closely related to the notion of ***linearity*** (Wadler, 1990). When persistence is allowed, traditional amortized analyses break down because operations on "expensive" versions of a data structure can be repeated arbitrarily often. Okasaki (1995a; 1996) describes how to use the memoization implicit in lazy evaluation to support amortized data structures whose bounds hold even under persistence. However, of the above data structures, only pairing heaps appear to be amenable to this technique.

Finally, our data structure borrows techniques from several sources. Skew linking is borrowed from the random-access lists of Okasaki (1995b), which in turn are a modification of the random-access stacks of Myers (1983). We use skew linking to reduce the cost of insertion in binomial queues to $O(1)$, but recursive slowdown (Kaplan & Tarjan, 1995) and lazy evaluation (Okasaki, 1996) could be used for the same purpose. Data-structural bootstrapping is used by Buchsbaum *et al.* (Buchsbaum *et al.*, 1995; Buchsbaum & Tarjan, 1995) to support catenation for double-ended queues, much as we use it to support melding for priority queues.

8 Discussion

We have described the first purely functional implementation of priority queues to support *findMin*, *insert*, and *meld* in $O(1)$ worst-case time, and *deleteMin* in $O(\log n)$ worst-case time. These bounds are asymptotically optimal among all comparison-based priority queues. Our data structure is an adaptation of an imperative data structure introduced by Brodal (1995), but we have both simplified his original data structure and clarified its relationship to the binomial queues of Vuillemin (1978). Our data structure is reasonably efficient in practice; however, there are several competing data structures that, although not asymptotically optimal, are somewhat faster than ours in practice.

Hence, our work is primarily of theoretical interest. The major area in which our data structure should be useful in practice is applications dominated by melding, particularly applications that also require persistent priority queues.

Although we have implemented our data structure in Standard ML, a strict functional language, it could easily be translated into other functional languages, even lazy languages such as Haskell (Hudak *et al.*, 1992). However, in a lazy language, the worst-case bounds become amortized because the actions of each ***insert***, ***meld***, and ***deleteMin*** are delayed until their results are needed by a ***findMin***. For instance, a ***findMin*** following a sequence of m insertions and melds will take $\Omega(m)$ time, although that time can be amortized over the insertions and melds in the usual way. This problem is not unique to our data structure — it applies to virtually all nominally worst-case data structures in a lazy language. See Okasaki (1995a; 1996) for a fuller discussion of the interaction between lazy evaluation and amortization.

Next, we note that imperative priority queues often support two additional operations, ***decreaseKey*** and ***delete***, that decrease and delete a specified element of the queue, respectively. The element in question is usually specified by a pointer into the middle of the queue, but this is awkward in a functional setting. One approach is to represent the queue as a binary search tree, so that we can efficiently search for arbitrary elements. This is essentially the approach taken by King (1994). Empirical comparisons by Jones (1986) suggest that splay trees would be ideal for this purpose, at least for predominantly ephemeral usage.[2] Unfortunately, melding binary search trees (including splay trees) requires $O(n)$ time.

An alternative approach is to use two priority queues, one containing "positive" occurrences of elements and one containing "negative" occurrences of elements. To delete an element, simply insert it into the negative queue. To decrease an element, delete the old value and insert the new value. Positive and negative occurrences of the same element cancel each other out when they both become the minimum elements of their respective queues. This approach can be viewed as

[2]However, since *findMin* on splay trees takes $O(\log n)$ amortized time, it may be desirable to first apply the *AddRoot* transformation of Section 4.

the functional analogue of the ***lazy delete*** operation of Tarjan (1983). This solution works well provided the number of negative elements is relatively small. However, when there are many positive-negative pairs that have not yet cancelled each other out, this solution may be inefficient in both time and space. Further research is needed to support ***decreaseKey*** and ***delete*** efficiently in a functional setting.

A final area of future work concerns the Standard ML module system. As noted in Section 5, recursive modules are not always sensible, and hence are currently disallowed in implementations of the language. However, recursion at the module level does appear to be sensible — and useful — for certain well-behaved modules. It would be interesting to formalize the conditions under which recursive modules should be allowed, and extend some implementation of Standard ML accordingly.

Acknowledgments

Thanks to Peter Lee, David King, and Amy Moormann Zaremski for their comments and suggestions on an earlier draft of this paper.

References

Brodal, Gerth Stølting. (1995). Fast meldable priority queues. *Pages 282–290 of: Workshop on Algorithms and Data Structures*. LNCS, vol. 955. Springer-Verlag.

Brodal, Gerth Stølting. 1996 (Jan.). Worst-case priority queues. *Pages 52–58 of: ACM-SIAM Symposium on Discrete Algorithms*.

Brown, Mark R. (1978). Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, **7**(3), 298–319.

Buchsbaum, Adam L., & Tarjan, Robert E. (1995). Confluently persistent deques via data structural bootstrapping. *Journal of Algorithms*, **18**(3), 513–547.

Buchsbaum, Adam L., Sundar, Rajamani, & Tarjan, Robert E. (1995). Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues. *SIAM Journal on Computing*, **24**(6), 1190–1206.

Crane, Clark Allan. 1972 (Feb.). *Linear lists and priority queues as balanced binary trees*. Ph.D. thesis, Computer Science Department, Stanford University. Available as STAN-CS-72-259.

Driscoll, James R., Sarnak, Neil, Sleator, Daniel D. K., & Tarjan, Robert E. (1989). Making data structures persistent. *Journal of Computer and System Sciences*, **38**(1), 86–124.

Fredman, Michael L., & Tarjan, Robert E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, **34**(3), 596–615.

Fredman, Michael L., Sedgewick, Robert, Sleator, Daniel D. K., & Tarjan, Robert E. (1986). The pairing heap: A new form of self-adjusting heap. *Algorithmica*, **1**(1), 111–129.

Hood, Robert. 1982 (Aug.). *The Efficient Implementation of Very-High-Level Programming Language Constructs*. Ph.D. thesis, Department of Computer Science, Cornell University. (Cornell TR 82-503).

Hoogerwoord, Rob R. (1992). A logarithmic implementation of flexible arrays. *Pages 191–207 of: Conference on Mathematics of Program Construction*. LNCS, vol. 669. Springer-Verlag.

Hudak, Paul, Peyton Jones, Simon, Wadler, Philip, Boutel, Brian, Fairbairn, Jon, Fasel, Joseph, Guzmán, María M., Hammond, Kevin, Hughes, John, Johnsson, Thomas, Kieburtz, Dick, Nikhil, Rishiyur, Partain, Will, & Peterson, John. (1992). Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices*, **27**(5).

Jones, Douglas W. (1986). An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, **29**(4), 300–311.

Kaldewaij, Anne, & Schoenmakers, Berry. (1991). The derivation of a tighter bound for top-down skew heaps. *Information Processing Letters*, **37**(5), 265–271.

Kaplan, Haim, & Tarjan, Robert E. 1995 (May). Persistent lists with catenation via recursive slow-down. *Pages 93–102 of: ACM Symposium on Theory of Computing*.

King, David J. 1994 (Sept.). Functional binomial queues. *Pages 141–150 of: Glasgow Workshop on Functional Programming*.

MacQueen, David B., & Tofte, Mads. 1994 (Apr.). A semantics for higher-order functors. *Pages 409–423 of: European Symposium on Programming*.

Milner, Robin, Tofte, Mads, & Harper, Robert. (1990). *The Definition of Standard ML*. Cambridge, Massachusetts: The MIT Press.

Myers, Eugene W. (1983). An applicative random-access stack. *Information Processing Letters*, **17**(5), 241–248.

Okasaki, Chris. 1995a (Oct.). Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. *Pages 646–654 of: IEEE Symposium on Foundations of Computer Science*.

Okasaki, Chris. 1995b (June). Purely functional random-access lists. *Pages 86–95 of: Conference on Functional Programming Languages and Computer Architecture.*

Okasaki, Chris. (1995c). Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, **5**(4), 583–592.

Okasaki, Chris. 1996 (May). The role of lazy evaluation in amortized data structures. *Pages 62–72 of: ACM SIGPLAN International Conference on Functional Programming.*

Paulson, Laurence C. (1991). *ML for the Working Programmer.* Cambridge: Cambridge University Press.

Schoenmakers, Berry. 1992 (Sept.). *Data Structures and Amortized Complexity in a Functional Setting.* Ph.D. thesis, Eindhoven University of Technology.

Sleator, Daniel D. K., & Tarjan, Robert E. (1985). Self-adjusting binary search trees. *Journal of the ACM*, **32**(3), 652–686.

Sleator, Daniel D. K., & Tarjan, Robert E. (1986). Self-adjusting heaps. *SIAM Journal on Computing*, **15**(1), 52–69.

Tarjan, Robert E. (1983). *Data Structures and Network Algorithms.* CBMS Regional Conference Series in Applied Mathematics, vol. 44. Philadelphia: Society for Industrial and Applied Mathematics.

Vuillemin, Jean. (1978). A data structure for manipulating priority queues. *Communications of the ACM*, **21**(4), 309–315.

Wadler, Philip. 1990 (Apr.). Linear types can change the world! *Pages 561–581 of: Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods.*

Williams, J. W. J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, **7**(6), 347–348.

Recent Publications in the BRICS Report Series

RS-96-37 **Gerth Stølting Brodal and Chris Okasaki.** ***Optimal Purely Functional Priority Queues*****. October 1996. 27 pp. To appear in *****Journal of Functional Programming*****, 6(6), December 1996.**

RS-96-36 **Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***On a Question of A. Salomaa: The Equational Theory of Regular Expressions over a Singleton Alphabet is not Finitely Based*****. October 1996. 16 pp.**

RS-96-35 **Gian Luca Cattani and Glynn Winskel.** ***Presheaf Models for Concurrency*****. October 1996. 16 pp. Presented at the *****Annual Conference of the European Association for Computer Science Logic*****, CSL '96.**

RS-96-34 **John Hatcliff and Olivier Danvy.** ***A Computational Formalization for Partial Evaluation (Extended Version)*****. October 1996. To appear in *****Mathematical Structures in Computer Science*****.**

RS-96-33 **Jonathan F. Buss, Gudmund Skovbjerg Frandsen, and Jeffrey Outlaw Shallit.** ***The Computational Complexity of Some Problems of Linear Algebra*****. September 1996. 39 pp.**

RS-96-32 **P. S. Thiagarajan.** ***Regular Trace Event Structures*****. September 1996. 34 pp.**

RS-96-31 **Ian Stark.** ***Names, Equations, Relations: Practical Ways to Reason about 'new'*****. September 1996. ii+22 pp.**

RS-96-30 **Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup.** ***Fusion Trees can be Implemented with AC^0 Instructions only*****. September 1996. 8 pp.**

RS-96-29 **Lars Arge.** ***The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation*****. August 1996. 35 pp. An extended abstract version appears in Staples, Eades, Kato, and Moffat, editors, *****Algorithms and Computation: 6th International Symposium*****, ISAAC '95 Proceedings, LNCS 1004, 1995, pages 82–91.**