The Role of Lazy Evaluation in Amortized Data Structures

Chris Okasaki*
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA 15213
(e-mail: cokasaki@cs.cmu.edu)

Abstract

Traditional techniques for designing and analyzing amortized data structures in an imperative setting are of limited use in a functional setting because they apply only to single-threaded data structures, yet functional data structures can be non-single-threaded. In earlier work, we showed how lazy evaluation supports functional amortized data structures and described a technique (the banker's method) for analyzing such data structures. In this paper, we present a new analysis technique (the physicist's method) and show how one can sometimes derive a worst-case data structure from an amortized data structure by appropriately scheduling the premature execution of delayed components. We use these techniques to develop new implementations of FIFO queues and binomial queues.

1 Introduction

Functional programmers have long debated the relative merits of strict versus lazy evaluation. Although lazy evaluation has many benefits [11], strict evaluation is clearly superior in at least one area: ease of reasoning about asymptotic complexity. Because of the unpredictable nature of lazy evaluation, it is notoriously difficult to reason about the complexity of algorithms in such a language. However, there are some algorithms based on lazy evaluation that cannot be programmed in (pure) strict languages without an increase in asymptotic complexity. We explore one class of such algorithms — amortized data structures — and describe techniques for reasoning about their complexity.

Several researchers have developed theoretical frameworks for analyzing the time complexity of lazy programs [1, 19, 20, 25]. However, these frameworks are not yet mature enough to be useful in practice. One difficulty is that these frameworks are, in some ways, too general. In each of these systems, the cost of a program is calculated with respect to some context, which is a description of the demand on the result of the program. However, this approach is often inappropriate for a methodology of program development in which data structures are designed as abstract data types whose behavior, including time complexity, is specified in isolation. Instead, we develop ad hoc, but pragmatically useful, techniques for reasoning about the time complexity of lazy amortized data structures without regard to the contexts in which the data structures will be used.

A data structure is called *persistent* [5] if, after an update, the old version of the data structure is still accessible. A data structure that is not persistent is called *ephemeral*. In functional programming terminology, an ephemeral data structure is one that must be single-threaded [21] and a persistent data structure is one that may be non-single-threaded. Aside from the obvious distinction regarding assignments, persistence is the fundamental difference between functional and imperative data structures. Functional data structures are automatically persistent, whereas imperative data structures are almost always ephemeral. Traditional techniques for designing and analyzing amortized data structures were developed for imperative data structures and apply only in the ephemeral case. Functional (and therefore persistent) amortized data structures require different techniques.

In [16], we showed how lazy evaluation can be used to support persistent amortized data structures, and described the banker's method, a technique for analyzing the time complexity of such data structures. In this paper, we begin by reviewing these previous results in a functional setting. Then, we describe the physicist's method, an alternative technique for analyzing functional amortized data structures. The physicist's method is less powerful than the banker's method, but is usually much simpler. Next, we show how one can sometimes derive a worst-case data structure from an amortized data structure by appropriately scheduling the premature execution of delayed components. This technique requires both strict and lazy evaluation. Finally, after a brief discussion of related work, we conclude with advice on designing amortized data structures.

To illustrate our techniques, we introduce several new implementations of common data structures. In Section 3, we describe an extremely simple FIFO queue requiring $O(1)$ amortized time per operation. In Section 4, we show that binomial queues implemented with lazy evaluation support insertion in only $O(1)$ amortized time. Finally, in Section 5, we adapt this implementation of binomial queues to support insertion in $O(1)$ worst-case time.

We present some source code in Haskell [10], and some in

*This research was sponsored 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.

ICFP '96 5/96 PA, USA
© 1996 ACM 0-89791-771-5/96/0005...$3.50

Standard ML [15]. Since Standard ML is strict, we extend the language with the following primitives for lazy evaluation:

```
type 'a susp
val delay : (unit -> 'a) -> 'a susp
val force : 'a susp -> 'a
```

These primitives are actually supported by several implementations of Standard ML.[1]

For clarity, we present only the relevant fragments of the source code for some data structures. The complete implementations are included in Appendix A.

2 Amortization and Lazy Evaluation

Amortization is a method of accounting for the cost of sequences of operations [23]. The amortized cost of an individual operation is obtained by averaging the total cost of a worst-case sequence over all the operations in the sequence. Given a bound on the amortized cost of an individual operation, one can calculate a bound for the cost of a sequence of operations by simply multiplying by the length of the sequence. Cost can be measured in time, space, or any other resource of interest, but in this paper we will restrict our attention to running time as the sole measure of cost.

In an amortized data structure, certain operations are allowed to be more expensive than the desired bound, provided they are balanced by a sufficient number of inexpensive operations. Persistent data structures are problematic in this regard, since they allow expensive operations to be repeated arbitrarily often. To obtain meaningful amortized bounds for persistent data structures, we must ensure that if x is some instance of a data structure on which some operation f is more expensive than the desired bound, then the first application of f to x may be expensive, but subsequent applications will not be. This is impossible under both call-by-value and call-by-name since each application of f to x will take exactly the same amount of time. Of the three major evaluation orders, only call-by-need (i.e., lazy evaluation) supports the desired behavior. If x contains some delayed component that is demanded by f, then the first application of f to x will force the (potentially expensive) evaluation of that component and memoize the result. Subsequent applications may then access the memoized result directly.

Tarjan [23] describes two techniques for analyzing ephemeral amortized data structures: the *banker's method* and the *physicist's method*. Both of these techniques account for future expensive operations by prepaying. Whenever the amortized cost of an operation is greater than the actual cost, the excess is saved to pay for future operations. Whenever the amortized cost is less than the actual cost, the deficit is made up from earlier savings. The two techniques differ in how they keep track of these savings.

In the banker's method, the savings are represented as *credits* that are associated with individual locations in the data structure. These credits are used to pay for future accesses to these locations. In the physicist's method, the savings are represented as *potential* that is associated with the data structure as a whole. Inexpensive operations increase this potential and expensive operations decrease it.

[1] It is possible to implement these primitives in Standard ML using references and assignments, but not with the same degree of polymorphism.

Unfortunately, neither of these techniques is appropriate for analyzing persistent data structures. An ephemeral (single-threaded) data structure has only a single future, but a persistent (non-single-threaded) data structure may have many futures, one for each thread. Then, the whole idea of saving credits (or potential) for future use breaks down because each of those futures may need the same credits. To cope with persistent data structures, we base our analyses on debt, rather than savings, where debt accounts for the cost of delayed computations. The intuition is that, although savings cannot be spent more than once, it does no harm to pay off a debt more than once (in the same way that it does no harm to demand a memoized value more than once). By allowing debt to be paid off multiple times, we avoid the Gordian knot of analyzing interthread dependencies. If we can show that *every* individual thread pays off its own debt, then surely the *first* thread to force a delayed computation will pay off the relevant debt. Subsequent threads demanding the same result may unnecessarily pay off the same debt, but this does no harm.

3 The Banker's Method

We adapt the banker's method to account for lazy evaluation and persistence by replacing credits with debits. Each debit represents a constant amount of delayed work. When we initially delay a given computation, we create a number of debits proportional to its eventual actual cost and associate each debit with a location in the data structure. The choice of location for each debit depends on the nature of the computation. If the computation is *monolithic* (i.e., once begun, it runs to completion), then all debits are usually assigned to the root of the result. On the other hand, if the computation is *incremental* (i.e., decomposable into fragments that may be executed independently), then the debits may be distributed among the roots of the partial results.

Each operation is allowed to discharge a number of debits proportional to its amortized cost. The order in which debits should be discharged depends on how the data structure will be accessed; debits on nodes likely to be accessed soon should be discharged first. To prove an amortized bound, we must show that, whenever we access a location (possibly triggering the execution of a delayed computation), all debits associated with that location have already been discharged (and hence the delayed computation has been paid for).

Incremental functions play an important role in the banker's method because they allow debits to be dispersed to different locations in a data structure. Then, each location can be accessed as soon as its debits are discharged, without waiting for the debits at other locations to be discharged. In practice, this means that the initial partial results of an incremental computation can be paid for very quickly, and that subsequent partial results may be paid for as they are needed. Monolithic functions, on the other hand, are much less flexible. The programmer must anticipate when the result of an expensive monolithic computation will be needed, and set up the computation far enough in advance to be able to discharge all its debits by the time its result is needed.

The banker's method was first adapted to a persistent setting in [16].

Example: Amortized FIFO Queues

As an example of the banker's method, we next give an implementation of persistent FIFO queues that supports all standard operations in $O(1)$ amortized time. This implementation is similar to, but simpler than, that presented in [17]. We represent a queue as a pair of lists $\langle f, r\rangle$, where f is the front segment of the queue, and r is the rear segment of the queue in reverse order. Elements are enqueued at the head of r and dequeued from the head of f. We explicitly maintain the lengths of f and r, and guarantee that $|f| \geq |r|$ by *rotating* the queue whenever $|r| = |f| + 1$. A rotation transfers elements from r to f by replacing $\langle f, r\rangle$ with $\langle f +\!\!+ \mathit{reverse}\ r, [\,]\rangle$. Figure 1 gives Haskell source code for this implementation.

Rotations are the only non-trivial computations in this implementation. We account for the cost of rotations using the banker's method. Every rotation creates $|f| + |r| = 2|f| + 1$ debits — $|f|$ debits for the append and $|r|$ debits for the reverse. Since the append function is incremental, we disperse the first $|f|$ debits across the first $|f|$ elements of the resulting queue. However, since the reverse function is monolithic, we assign all the remaining debits to the first element of the reversed list (i.e., the $(|f| + 1)$-st element of the resulting queue). We discharge debits at the rate of one debit per *enqueue* and two debits per *dequeue*. To prove the amortized bounds, we must show that the first element of the queue never has any undischarged debits.

Let d_i be the number of debits on element i, and let $D_i = \sum_{j=0}^{i} d_i$. We maintain the following invariant:

$$D_i \leq \min(2i, |f| - |r|)$$

The $2i$ term guarantees that all debits on the first element have been discharged (i.e., $D_0 = d_0 = 0$), and the $|f| - |r|$ term guarantees that all debits in the entire queue have been discharged whenever the lists are of equal length (i.e., just before the next rotation).

Now, every *enqueue* that does not cause a rotation simply adds a new element to the rear list, increasing $|r|$ by one and decreasing $|f| - |r|$ by one. Discharging a single debit restores the invariant. Every *dequeue* that does not cause a rotation simply removes an element from the front list. This decreases $|f|$ by one (and hence $|f| - |r|$ by one), but, more importantly, it decreases i by one for every remaining element, which in turn decreases $2i$ by two. Discharging the first two debits in the queue restores the invariant.

Finally, we consider an *enqueue* or *dequeue* that causes a rotation. Immediately prior to the operation, we know that $|f| = |r|$, so $D_i = 0$. Hence, after the rotation, the only debits are those generated by the rotation itself. These debits are distributed such that

$$d_i = \begin{cases} 1 & \text{if } i < m \\ m+1 & \text{if } i = m \\ 0 & \text{if } i > m \end{cases} \quad \text{and} \quad D_i = \begin{cases} i+1 & \text{if } i < m \\ 2m+1 & \text{if } i \geq m \end{cases}$$

where m is the length of f at the beginning of the rotation. This debit distribution violates the invariant at both location 0 and location m, but discharging the debit on the first element restores the invariant.

4 The Physicist's Method

Like the banker's method, the physicist's method can also be adapted to support persistent amortized data structures based on lazy evaluation. In the traditional physicist's method, one describes a function Φ that maps each data structure to a potential representing a lower bound on the total savings generated by the sequence of operations that created the data structure. The amortized cost of an operation is defined to be the actual cost of the operation plus the change in potential. To work with debt instead of savings, we replace Φ with a function Ψ that maps each data structure to a potential representing an upper bound on the total unpaid debt of the delayed computations within that data structure. Note that Ψ is used only in the analysis of a data structure; it does not actually appear in the program text. In this framework, the amortized cost of an operation is defined to be the actual cost of the computations delayed by the operation minus the change in potential. More formally, if $\hat{c}_p$ and c_p are the amortized and actual cost, respectively, of some operation p, and x and x' are the versions of the data structure before and after the operation, respectively, then

$$\hat{c}_p = c_p - (\Psi(x') - \Psi(x))$$

To prove amortized bounds in this framework, we must show that the entire debt of a data structure has been paid off before we force any part of the data structure. Because we only know the total debt of the data structure as a whole, and not the debt of individual locations, we cannot access certain locations early, as we can with the banker's method. However, when applicable, the physicist's method tends to yield much simpler proofs than the banker's method.

Example: Amortized Binomial Queues

Binomial queues are an elegant form of priority queue invented by Vuillemin [24]. Inserting an element into a binomial queue requires $O(\log n)$ worst-case time, but it is well known that imperative (i.e., ephemeral) binomial queues support insertion in $O(1)$ amortized time [13]. We now show, using the physicist's method, that persistent binomial queues implemented with lazy evaluation also support insertion in $O(1)$ amortized time.

A binomial queue is a forest of heap-ordered trees. The structure of this forest is governed by the binary representation of the size of the queue; if the ith digit is one, then the forest contains a tree of size 2^i. Two trees of size 2^i can be combined to form a tree of size 2^{i+1} in constant time by the *link* operation. To insert an element into the queue, we create a new singleton tree and repeatedly link trees of equal size until all sizes are unique. This process is analogous to adding one to a binary number. See [14] for more details about binomial queues.

Figure 2 gives a fragment of a Haskell implementation of binomial queues. In this implementation, we use a sparse representation of binomial queues, meaning we do not explicitly represent the zeros in the binary representation of the size of the queue. This requires that we tag each tree with its size. Note that *insert* is monolithic because *addUnique* does not return until it has performed all the necessary links. In Section 5, we will also consider a non-sparse representation of binomial queues for which *insert* is incremental.

To analyze the current data structure using the physicist's method, we first define the potential function to be $\Psi(q) = Z(|q|)$, where $Z(n)$ is the number of zeros in the (minimum length) binary representation of n. Next, we show that the amortized cost of inserting an element into

```
data Queue a = Queue Int [a] Int [a]
    -- Invariants:  each queue has the form Queue lenf f lenr r
    --              where lenf = |f| ∧ lenr = |r| ∧ lenf ≥ lenr

empty :: Queue a
empty = Queue 0 [] 0 []

isEmpty :: Queue a -> Bool
isEmpty (Queue lenf f lenr r) = (lenf == 0)    -- since lenf ≥ lenr, lenf = 0 implies lenr = 0

enqueue :: a -> Queue a -> Queue a
enqueue x (Queue lenf f lenr r) = makeq lenf f (lenr+1) (x:r)

dequeue :: Queue a -> (a, Queue a)
dequeue (Queue (lenf+1) (x:f) lenr r) = (x, makeq lenf f lenr r)

-- auxiliary pseudo-constructor: guarantees lenf ≥ lenr
makeq :: Int -> [a] -> Int -> [a] -> Queue a
makeq lenf f lenr r | lenr <= lenf   = Queue lenf f lenr r
                    | lenr == lenf+1 = Queue (lenf+lenr) (f ++ reverse r) 0 []
```

Figure 1. A Haskell implementation of amortized FIFO queues.

```
data Tree a = Node a [Tree a]        -- children in decreasing order of size

type BinQueue a = [(Int,Tree a)]     -- trees in increasing order of size

insert :: Ord a => a -> BinQueue a -> BinQueue a
insert x q = addUnique (1, Node x []) q

-- auxiliary functions

-- add a new tree and link until all sizes are unique
addUnique :: Ord a => (Int,Tree a) -> BinQueue a -> BinQueue a
addUnique (n,t) [] = [(n,t)]
addUnique (n,t) ((n',t') : q) | n < n'  = (n,t) : (n',t') : q
                              | n == n' = addUnique (n+n', link t t') q

-- make the tree with the larger root a child of the tree with the smaller root
link :: Ord a => Tree a -> Tree a -> Tree a
link (Node x c) (Node y d) | x <= y = Node x (Node y d : c)
                           | y < x  = Node y (Node x c : d)
```

Figure 2. A fragment of a Haskell implementation of amortized binomial queues.

a binomial queue of size n is two. Suppose that the lowest m digits in the binary representation of n are ones. Then, inserting the element will eventually generate a total of $m+1$ calls to *addUnique*. Now, consider the change in potential. The lowest m digits have changed from ones to zeros and the next digit has changed from zero to one, so the change in potential is $m-1$. The amortized cost of insertion is thus $(m+1)-(m-1)=2$.

The remaining operations supported by binomial queues — finding the minimum element, deleting the minimum element, and merging two queues — all have an actual cost of $O(\log n)$. Before these operations can inspect the data structure, they must first pay off the outstanding debt. However, the outstanding debt is bounded by $Z(n) = O(\log n)$, so the total amortized cost of these operations is still $O(\log n)$. Source code for these operations appears in Appendix A.

5 Eliminating Amortization

Amortized and worst-case data structures differ mainly in when the computations charged to a given operation occur. In a worst-case data structure, all the computations charged to an operation occur during the operation. In an amortized data structure, some of the computations charged to an operation may actually occur during later operations. From this, we see that virtually all nominally worst-case data structures become amortized when implemented in an entirely lazy language because many computations are unnecessarily delayed. To describe true worst-case data structures, we therefore need a strict language. If we want to describe both amortized and worst-case data structures, we need a language that supports both lazy and strict evaluation. Given such a language, we can also consider an intriguing hybrid approach: worst-case data structures that use lazy evaluation internally. Such data structures can be obtained from amortized data structures by appropriately scheduling the premature execution of delayed components. The trick is to regard paying off debt as a literal activity, and to execute each delayed computation as it is paid for.

In a worst-case data structure, we no longer have the freedom for certain operations to be more expensive than the desired bound. Incremental functions assume a vital role by decomposing expensive computations into fragments, each of which can be executed within the allotted time. Often a given fragment will depend on a fragment of an earlier computation. The difficult part of proving a worst-case bound in this framework is guaranteeing that executions of fragments will never cascade. This is done by showing that, whenever a given fragment is executed, all fragments on which it depends have already been executed and memoized.

Implementing a worst-case data structure in this framework requires extending the amortized data structure with an extra component, called the *schedule*, that imposes an order on the delayed computations within the data structure. Every operation, in addition to whatever manipulations it performs on the data structure itself, executes the first few jobs in the schedule. The exact number of jobs executed is governed by the (previously) amortized cost of the operation. For this technique to apply, maintaining the schedule cannot require more time than the desired worst-case bounds.

A special case of this general technique was first used to implement worst-case FIFO queues in [17].

Example: Worst-Case Binomial Queues

We now return to the example of binomial queues, and modify the earlier implementation to support insertions in $O(1)$ worst-case time. Recall that, in our earlier implementation, *insert* was monolithic. We first make *insert* incremental by changing the data structure to represent explicitly the zeros in the binary representation of the size of the queue. Then, every call to *addUnique* can return a partial result. The final call to *addUnique* returns a stream whose first element is a *One*. All the intermediate calls to *addUnique* return streams beginning with a *Zero*. This implementation appears in Figure 3. In addition to modifying the representation, we have also changed the source language from Haskell to Standard ML (extended with primitives for lazy evaluation). Other than the choice of language, this implementation of binomial queues is very similar to that of King [14]. Note that this change in representation does not affect the amortized analysis from the previous section. In particular, the amortized analysis also holds for King's implementation.

Now, we extend binomial queues with a schedule of delayed computations. The only delayed computations in this implementation are calls to *addUnique*. Thus, the schedule will be a list of unevaluated calls to *addUnique*.

```
type Schedule = Digit Stream list
```

To execute a job, we force the first element in this list. If the result is a *One*, then this is the last fragment of a computation. Otherwise, the tail of the result is another unevaluated call to *addUnique*, so we put it back in the schedule. We execute two jobs by calling *execute* twice.

```
fun execute [] = []
  | execute (job :: schedule) =
      case force job of
          Cons (One t, _) => schedule
        | Cons (Zero, job') => job' :: schedule

val execute2 = execute o execute
```

Finally, we update *insert* to maintain the schedule. Since the amortized cost of *insert* is two, we execute two jobs per insertion.

```
type BinQueue = Digit Stream * Schedule

fun insert x (q, schedule) =
      let val q' = addUnique (Node (x,[])) q
      in (q', execute2 (q' :: schedule)) end
```

This completes the changes necessary to convert the amortized bounds to worst-case bounds. The remaining operations on binomial queues all ignore the schedule and return entirely evaluated queues. Source code for these operations appears in Appendix A. The previous amortized analysis guarantees that any binomial queue contains at most $O(\log n)$ unevaluated computations, but the overhead of evaluating these during the normal execution of the remaining operations is absorbed without increasing the already $O(\log n)$ worst-case cost of these operations.

We must next show that, whenever *execute* forces a job of the form *addUnique t q*, q has already been evaluated and memoized. Define the *range* of a call to *addUnique* to be the partial result of that call together with the partial results of all its recursive calls. Note that every range consists of a (possibly empty) sequence of *Zero*'s followed by a *One*. We

```
datatype 'a StreamNode = Nil | Cons of 'a * 'a Stream
withtype 'a Stream = 'a StreamNode susp

type Elem = ...  (* elements may be any ordered type *)

datatype Tree = Node of Elem * Tree list  (* children in decreasing order of size *)
datatype Digit = Zero | One of Tree
type BinQueue = Digit Stream              (* digits/trees in increasing order of size *)

fun addUnique t q =  (* add one to low-order digit, link/carry if already a one *)
  delay (fn () => case force q of
                    Nil => Cons (One t, emptyStream)
                  | Cons (Zero, q) => Cons (One t, q)
                  | Cons (One t', q) => Cons (Zero, addUnique (link t t') q))

fun insert x q = addUnique (Node (x,[])) q
```

Figure 3. An alternative Standard ML implementation of amortized binomial queues.

say that two ranges *overlap* if any of their partial results have the same index within the stream of digits. Note that all unevaluated computations in a binomial queue are in the range of some job on the schedule. Thus, we can show that, for any job *addUnique t q*, q has already been evaluated and memoized by proving that no two jobs in the same schedule ever have overlapping ranges.

In fact, we prove a slightly stronger result. Define a *completed zero* to be a *Zero* whose cell in the stream has already been evaluated and memoized. Then, every valid binomial queue contains at least two completed zeros prior to the first range in the schedule, and at least one completed zero between every two adjacent ranges in the schedule. Proof: Consider a binomial queue immediately prior to an *insert*. Let r_1 and r_2 be the first two ranges in the schedule. Let z_1 and z_2 be the two completed zeros before r_1, and let z_3 be the completed zero between r_1 and r_2. Now, before executing two jobs, *insert* first adds a new range r_0 to the front of the schedule. Note that r_0 terminates in a *One* that replaces z_1. Let m be the number of *Zero*'s in r_0. There are three cases.

Case 1. $m = 0$. The only digit in r_0 is a *One*, so r_0 is eliminated by executing a single job. The second job forces the first digit of r_1. If this digit is *Zero*, then it becomes the second completed zero (along with z_2) before the first range. If this digit is *One*, then r_1 is eliminated and r_2 becomes the new first range. The two completed zeros prior to r_2 are z_2 and z_3.

Case 2. $m = 1$. The first two digits of the old digit stream were *One* and *Zero* (z_1), but they are replaced with *Zero* and *One*. Executing two jobs evaluates and memoizes both of these digits, and eliminates r_0. The leading *Zero* replaces z_1 as one of the two completed zeros before the first range (r_1).

Case 3. $m \geq 2$. The first two digits of r_0 are both *Zero*'s. They are both completed by executing the first two jobs, and become the two completed zeros before the new first range (the rest of r_0). z_2 becomes the single completed zero between r_0 and r_1. □

Once we have an implementation of binomial queues supporting *insert* in $O(1)$ worst-case time, we can improve the bounds of *findMin* and *merge* to $O(1)$ worst-case time using the bootstrapping transformation of Brodal and Okasaki [2]. The $O(\log n)$ bound for *deleteMin* is unaffected by this transformation.

6 Related Work

There has been very little previous work on amortized functional data structures. Schoenmakers [22] used functional notation to aid in deriving bounds for many amortized data structures, but considered only single-threaded data structures. Gries [6], Burton [3], and Hoogerwoord [9] described purely functional queues and double-ended queues with amortized bounds, but, again, supported only single-threaded queues. We first described unrestricted amortized queues in [17] as an intermediate step in the development of worst-case queues based on lazy evaluation. However, because of the concern with worst-case bounds, that implementation is more complicated than the implementation in Section 3. We derived the worst-case queues from the amortized queues using techniques similar to those in Section 5. In [16], we recognized the importance of lazy evaluation to non-single-threaded amortized data structures in general, and adapted the banker's method to analyze such data structures. We then used the banker's method to describe an implementation of lists supporting catenation and all other usual list primitives in $O(1)$ amortized time. This paper extends our earlier work by adapting the physicist's method to cope with persistence and by generalizing the technique for eliminating amortization. In addition, we introduce several new data structures, which may be useful in their own right.

For every amortized functional data structure currently known, there is a competing worst-case data structure that does not depend on lazy evaluation. Examples include queues [8], double-ended queues [7, 4], catenable lists [12], and skew binomial queues [2]. In every case, the amortized data structure is significantly simpler than the worst-case version. However, the amortized data structure is usually slightly slower in practice, mostly because of overheads associated with lazy evaluation. Memoization, in particular, causes problems for many garbage collectors. Of course, if both data structures are implemented in a lazy language, then both data structures will pay these overheads. In that case, the amortized data structure should usually be faster as well as simpler.

Our research is also related to earlier studies in the im-

perative community. Driscoll, Sarnak, Sleator, and Tarjan [5] described several techniques for implementing persistent imperative data structures, and Raman [18] explored techniques for eliminating amortization from imperative data structures.

7 Discussion

We have shown how lazy evaluation is essential to the design of amortized functional data structures, and given several techniques for analyzing such data structures. In addition, we have described how to eliminate amortization from data structures based on lazy evaluation by prematurely executing delayed components in a pattern suggested by the amortized analysis. Finally, we have illustrated our techniques with new implementations of FIFO queues and binomial queues.

We have made several observations about the relationship between evaluation order and kind of bound (amortized or worst-case). Amortized data structures require lazy evaluation, but worst-case data structures require strict evaluation. Thus, from our point of view, the ideal functional programming language would seamlessly support both evaluation orders. Currently, both major functional languages — Haskell and Standard ML — fail to meet this criterion. Haskell has only limited support for strict evaluation, and Standard ML has only limited support for lazy evaluation.

We close with some hints on designing amortized functional data structures.

- Identify a (potentially expensive) procedure for reorganizing your data to make future operations cheap. Make sure that the procedure is executed lazily.
- Consider the access patterns of the data structure. If operations routinely need the entire result of the reorganizing procedure, attempt to use the physicist's method. If operations routinely inspect partial results, use the banker's method instead.
- Especially if using the banker's method, make the procedure as incremental as possible. If some or all of the procedure cannot be made incremental, arrange to set up the computation well in advance of when it will be needed to allow time to pay for the computation.
- If the entire procedure can be made incremental, and if there is some clear order in which the fragments of the computation should be executed, consider converting the data structure to support worst-case bounds by explicitly scheduling the premature evaluation of each fragment.

References

[1] Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Conference on Functional Programming Languages and Computer Architecture*, pages 157–165, 1989.

[2] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. Submitted for publication.

[3] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[4] Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, 1993.

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

[6] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[7] Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs.* PhD thesis, Department of Computer Science, Cornell University, 1982.

[8] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, November 1981.

[9] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.

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

[11] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[12] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *ACM Symposium on Theory of Computing*, pages 93–102, 1995.

[13] Chan Meng Khoong and Hon Wai Leong. Double-ended binomial queues. In *International Symposium on Algorithms and Computation*, volume 762 of *LNCS*, pages 128–137. Springer-Verlag, 1993.

[14] David J. King. Functional binomial queues. In *Glasgow Workshop on Functional Programming*, pages 141–150, September 1994.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[16] Chris Okasaki. Amortization, lazy evaluation, and persistence. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, 1995.

[17] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4), October 1995.

[18] Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Times.* PhD thesis, Department of Computer Sciences, University of Rochester, 1992.

[19] David Sands. Complexity analysis for a lazy higher-order language. In *European Symposium on Programming*, volume 432 of *LNCS*, pages 361–376. Springer-Verlag, 1990.

[20] David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, August 1995.

[21] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[22] Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting.* PhD thesis, Eindhoven University of Technology, 1992.

[23] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

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

[25] Philip Wadler. Strictness analysis aids time analysis. In *ACM Symposium on Principles of Programming Languages*, pages 119–132, 1988.

A Complete Source Code For Binomial Queues

In this appendix, we include the complete implementations of two variations of binomial queues. The first supports insertion in $O(1)$ amortized time and the second supports insertion in $O(1)$ worst-case time. Fragments of these implementations appeared earlier in Sections 4 and 5, respectively. Complete Haskell source code for amortized binomial queues is given in Figure 4. Complete Standard ML source code for worst-case binomial queues is given in Figures 5 and 6.

```
data Tree a = Node a [Tree a]          -- children in decreasing order of size

type BinQueue a = [(Int,Tree a)]       -- trees in increasing order of size

empty :: BinQueue a
empty = []

isEmpty :: BinQueue a -> Bool
isEmpty = null

insert :: Ord a => a -> BinQueue a -> BinQueue a
insert x q = addUnique (1, Node x []) q

merge :: Ord a => BinQueue a -> BinQueue a -> BinQueue a
merge [] q = q
merge q [] = q
merge ((n1,t1) : q1) ((n2,t2) : q2)
    | n1 < n2  = (n1,t1) : merge q1 ((n2,t2) : q2)
    | n1 > n2  = (n2,t2) : merge ((n1,t1) : q1) q2
    | n1 == n2 = addUnique (n1+n2, link t1 t2) (merge q1 q2)

findMin :: Ord a => BinQueue a -> a
findMin = minimum . map (root . snd)     -- return the minimum root

deleteMin :: Ord a => BinQueue a -> BinQueue a
deleteMin q = merge c' q'
    where (Node x c, q') = getMin q
          c' = zip sizes (reverse c)     -- convert children into a valid BinQueue
          sizes = 1 : map (2 *) sizes    -- [1,2,4,8,...]

-- auxiliary functions

root :: Tree a -> a
root (Node x c) = x

-- add a new tree and link until all sizes are unique
addUnique :: Ord a => (Int,Tree a) -> BinQueue a -> BinQueue a
addUnique (n,t) [] = [(n,t)]
addUnique (n,t) ((n',t') : q) | n < n'  = (n,t) : (n',t') : q
                              | n == n' = addUnique (n+n', link t t') q

-- make the tree with the larger root a child of the tree with the smaller root
link :: Ord a => Tree a -> Tree a -> Tree a
link (Node x c) (Node y d) | x <= y = Node x (Node y d : c)
                           | y < x  = Node y (Node x c : d)

-- find and remove the tree with the minimum root
getMin :: Ord a => BinQueue a -> (Tree a, BinQueue a)
getMin [(n,t)] = (t,[])
getMin ((n,t) : q) = let (t', q') = getMin q
                     in if root t <= root t' then (t, q) else (t', (n,t) : q')
```

Figure 4. A complete Haskell implementation of amortized binomial queues.

```
(* Streams *)

datatype 'a StreamNode = Nil | Cons of 'a * 'a Stream
withtype 'a Stream = 'a StreamNode susp

val emptyStream = delay (fn () => Nil)
fun isEmptyStream s = case force s of Nil => true | Cons (x, s) => false
fun cons (x, s) = delay (fn () => Cons (x, s))
fun normalize s = case force s of Nil => () | Cons (x, s) => normalize s

(* Binomial Queues *)

type Elem = ...  (* elements may be any ordered type *)

datatype Tree = Node of Elem * Tree list  (* children in decreasing order of size *)
datatype Digit = Zero | One of Tree
type Schedule = Digit Stream list         (* list of delayed calls to addUnique *)
type BinQueue = Digit Stream * Schedule   (* digits/trees in increasing order of size *)

exception Empty

local  (* auxiliary functions *)
  fun root (Node (x,c)) = x

  fun link (Node (x,c)) (Node (y,d)) =
        if x <= y then Node (x, Node (y,d) :: c)
                  else Node (y, Node (x,c) :: d)

  fun addUnique t q =  (* add one to low-order digit, link/carry if already a one *)
        delay (fn () => case force q of
                          Nil => Cons (One t, emptyStream)
                        | Cons (Zero, q) => Cons (One t, q)
                        | Cons (One t', q) => Cons (Zero, addUnique (link t t') q))

  fun smerge q1 q2 =  (* add digit streams, link/carry when two ones are in the same position *)
        case (force q1, force q2) of
            (Nil, _) => q2
          | (_, Nil) => q1
          | (Cons (Zero, q1), Cons (digit, q2)) => cons (digit, smerge q1 q2)
          | (Cons (digit, q1), Cons (Zero, q2)) => cons (digit, smerge q1 q2)
          | (Cons (One t1, q1), Cons (One t2, q2)) =>
                cons (Zero, addUnique (link t1 t2) (smerge q1 q2))

  fun getMin q =  (* find and remove the tree with the minimum root *)
        case force q of
            Nil => raise Empty
          | Cons (Zero, q) =>  (* zero is never the last digit *)
                let val (t, q) = getMin q
                in (t, cons (Zero, q)) end
          | Cons (One t, q) =>
                if isEmptyStream q then (t, emptyStream)
                else let val (t', q') = getMin q
                     in if root t <= root t' then (t, q) else (t', cons (One t, q')) end
```

Figure 5. A complete Standard ML implementation of worst-case binomial queues (part 1).

```
  fun execute [] = []
    | execute (job :: schedule) =  (* execute first job in schedule *)
        case force job of
            Cons (One t, _) => schedule            (* addUnique terminates *)
          | Cons (Zero, job') => job' :: schedule  (* addUnique continues *)

  val execute2 = execute o execute  (* execute two jobs *)

in
  val empty = (emptyStream, [])

  fun isEmpty (q, schedule) = isEmptyStream q

  fun insert x (q, schedule) =
        let val q' = addUnique (Node (x,[])) q
        in (q', execute2 (q' :: schedule)) end

  fun merge (q1,schedule1) (q2,schedule2) =
        let val q = smerge q1 q2
        in
            normalize q;  (* force and memoize entire stream *)
            (q, [])
        end

  fun findMin (q, schedule) =
        let val (t, _) = getMin q
        in root t end

  fun deleteMin (q, schedule) =
        let val (Node (x,c), q') = getMin q
            fun ones [] = emptyStream
              | ones (t :: ts) = cons (One t, ones ts)
            val c' = ones (rev c)  (* convert children into a queue *)
            val q'' = smerge c' q'
        in
            normalize q'';  (* force and memoize entire stream *)
            (q'', [])
        end
end
```

Figure 6. A complete Standard ML implementation of worst-case binomial queues (part 2).