Amortization, Lazy Evaluation, and Persistence: Lists with Catenation via Lazy Linking*

Chris Okasaki School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213 (e-mail: cokasaki@cs.cmu.edu)

Abstract

Amortization has been underutilized in the design of persistent data structures, largely because traditional accounting schemes break down in a persistent setting. Such schemes depend on saving "credits" for future use, but a persistent data structure may have multiple "futures", each competing for the same credits. We describe how lazy evaluation can often remedy this problem, yielding persistent data structures with good amortized efficiency. In fact, such data structures can be implemented purely functionally in any functional language supporting lazy evaluation. As an example of this technique, we present a purely functional (and therefore persistent) implementation of lists that simultaneously support catenation and all other usual list primitives in constant amortized time. This data structure is much simpler than the only existing data structure with comparable bounds, the recently discovered catenable lists of Kaplan and Tarjan, which support all operations in constant worst-case time.

1 Introduction

Over the past fifteen years, amortization has become a powerful tool in the design and analysis of data structures. Data structures with good amortized bounds are often simpler and faster than data structures with equivalent worst-case bounds. These data structures typically achieve their bounds by balancing the cost of a few expensive operations with a large number of cheap operations. Unfortunately, such designs frequently break down in a persistent setting,

where the expensive operations may be repeated arbitrarily often.

To solve this problem, we must guarantee that if x is a version of a persistent data structure on which some operation f is expensive, then the first application of f to x may be expensive, but repeated applications will not be. We achieve this with lazy evaluation. Portions of x — those portions that make the operation "expensive" — will be evaluated lazily, that is, only on demand. Furthermore, once evaluation takes place, those portions are updated with the resulting values in a process known as memoization[14]. Subsequent operations on x will have immediate access to the memoized values, without duplicating the previous work.

We first review the basic concepts of lazy evaluation, amortization, and persistence. We next discuss why the traditional approach to amortization breaks down in a persistent setting. Then we outline our approach to amortization, which can often avoid such problems through judicious use of lazy evaluation. We illustrate our technique by presenting an implementation of persistent catenable lists that support catenation and all other usual list primitives (push, pop, first) in constant amortized time. Achieving constant time bounds for such a data structure has been an open problem for many years, solved only recently by Kaplan and Tarjan [12], who support all operations in constant worst-case time. However, our solution is much simpler than theirs.

2 Lazy Evaluation

Lazy evaluation is an evaluation strategy employed by many purely functional programming languages. This strategy has two essential properties. First, the execution of a given computation is delayed until its

^{*}This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

result is needed. Second, the first time a delayed computation is executed, its resulting value is cached so that the next time it is needed, it can be looked up rather than recomputed. This caching is known as memoization [14].

Both of these ideas have a long history in data structure design, although not always in combination. For instance, delaying the execution of potentially expensive computations (often deletions) is used to good effect in hash tables [23], priority queues [20, 7], and search trees [4]. Memoization, on the other hand, is the basic principle of such techniques as dynamic programming [1] and path compression [10, 22].

In addition to delaying expensive computations, lazy evaluation provides a convenient framework for partitioning expensive computations into more manageable fragments. For instance, a function that generates a list of values might compute only the first, and delay the rest. When needed, the second value can be computed, and the rest again delayed. Proceeding in this fashion, the entire list can be computed one element at a time. Such incremental computations will play an important role in the design of persistent amortized data structures.

3 Traditional Amortization

Amortization has become an established tool in the design and analysis of data structures. The basic insight behind amortization is that one often cares more about the total cost of a sequence of operations than the cost of any individual operation. Thus, rather than the worst-case cost of an individual operation, a more appropriate metric is often the amortized cost of each operation, which is obtained by averaging the total cost of a (worst-case) sequence over all the operations in the sequence. Such a metric allows certain individual operations to be more expensive than the stated cost, provided they are balanced by sufficiently many operations that are cheaper than the stated cost.

Tarjan [21] describes two methods for analyzing amortized costs, known as the banker's view and the physicist's view. We will concentrate on the former. In the banker's view of amortization, we allocate a certain number of credits to each operation. A credit is an imaginary device that can "pay" for one unit of cost, where cost may be an estimate of running time, space consumption, or any other measure of interest. If the credits allocated to a particular operation are greater than the actual cost of that operation, then the excess credits carry over to help pay for future operations. If the credits are less than the actual cost, then the

deficit is made up by credits remaining from previous operations. Proving amortized bounds consists of showing that, beginning with zero credits, the balance of credits never becomes negative. In some analyses, the balance is allowed to be negative at intermediate stages provided it can be proven that the balance will be non-negative by the end of the sequence.

4 Persistence

A persistent data structure is one that supports updates and queries to any previous version of the data structure, not just the most recent. Driscoll, Sarnak, Sleator, and Tarjan [4] describe several general techniques for making ordinary linked data structures persistent. However, these techniques do not apply when operations such as catenation can combine multiple versions of a data structure. In purely functional programming, where side-effects are disallowed, all data structures are inherently persistent. Hence, we will focus on purely functional data structures, and not concern ourselves with special machinery for persistence.

A persistent data structure differs from an ordinary data structure in that it may have multiple "futures". A particular version x of a data structure may be updated to yield a new version y, and the same version x may later be updated again to yield another new version z. Both y and z (and their respective futures) may be considered futures of x. In general, a particular version may have arbitrarily many futures.

It is this property of persistent data structures that causes traditional amortization schemes to break down. Such schemes depend on saving credits for future use. However, a given credit may only be spent once. This is incompatible with the multiple futures of persistent data structures, where competing futures may need the same credit.

Note that amortization is sometimes introduced when extending a fundamentally worst-case data structure with the persistence machinery of Driscoll, Sarnak, Sleator and Tarjan [4]. This is not the sort of amortization we are speaking of. Rather, we are concerned with whether the underlying data structure is itself amortized.

¹ For these purposes, memoization is not considered a side-effect since, in the absence of other side-effects, it cannot change the meaning of a program, only the efficiency.

5 Amortization with Lazy Evaluation

We next describe how to use lazy evaluation in the design of amortized data structures whose bounds hold even in a persistent setting. This approach is based on the following key observation: although a single credit cannot be spent by more than one future, it does no harm for multiple futures to each pay off the same debit. Thus, our emphasis will be on discharging past debits rather than accumulating credits.

But what exactly is a debit? In our framework, a debit corresponds to a delayed computation involving a single credit's worth of work. More expensive delayed computations may be assigned multiple debits. Every operation is allocated a certain number of credits that are used to discharge existing debits. If there are no existing debits, the excess credits are wasted — they are not accumulated. Proving amortized bounds in this framework consists of demonstrating that, given a certain credit allocation, every debit will be discharged by the time its delayed computation is needed. Note that discharging a debit does not mean executing the delayed computation; it is merely an accounting scheme to pay for the delayed computations. To extend the financial analogy, a delayed computation is like a layaway plan. To purchase an "item" — the result of the computation — you first reserve the item by delaying the computation. You then make regular payments by discharging the associated debits, and are eligible to receive the item only when it is completely paid for. (However, even after it is completely paid for, you do not necessarily execute the delayed computation unless its result is needed.) Thus, there are three important moments in the life cycle of a delayed computation: when it is created, when it is paid for, and when it is executed. The proof obligation is to show that the second moment precedes the third.

This scheme can be used whenever it is possible to predict in advance what the expensive operations will be. In that case, you delay each expensive operation prior to the point where its result will actually be needed. Subsequent operations discharge the debits associated with the delayed computation, and you eventually reach the point where the result is needed. Provided all of the debits have been discharged, you may now execute the delayed computation. Memoization allows the result of this delayed computation to be shared among all futures that need it, preventing duplicated work. Note that both aspects of lazy evaluation are important here; memoization effects the sharing, but the delay allows the sharing to propagate back to past versions and forward from those past ver-

sions to alternate futures. Without the delay, memoization would only propagate to futures of the current version, so it would be possible to duplicate expensive computations by returning to a version just a few operations earlier, and repeating the operations that led to the expensive computation.

Memoization is the only way in which separate futures interact. If one future executes a delayed computation that is later needed by a second future, then that second future may simply look up the memoized value and avoid recomputing it. As far as the accounting scheme is concerned, separate futures interact only by making the discharging of certain debits redundant. It is a strange but pleasant fact that, in this framework, the worst case occurs when there is only a single future, in which case all memoization is wasted. Hence, in the analysis, we can ignore the possibility (and complications!) of multiple futures and need only show that any individual future will itself discharge all debits relevant to that future.

At first glance, the need to predict expensive operations in advance seems to drastically limit the applicability of this technique. However, incremental functions can often ease this restriction by breaking large computations into manageable chunks, each of which can be paid for separately. Thus, instead of needing to predict a large computation far enough in advance to be able to pay for the entire computation between the time that it is delayed and the time that it is executed, one must only be able to predict the large computation far enough in advance to be able to pay for the first chunk and then pay for each successive chunk as needed. If the chunks are small enough, as will be the case with catenable lists, the computation need not be set up in advance at all!

The sole previous example of this technique is the purely functional queues and deques of Okasaki [17]. Those data structures go further and actually eliminate the amortization by regarding the discharging of debits as a literal activity and appropriately scheduling the premature evaluation of lazy computations. Obtaining worst-case data structures in this manner should be possible whenever such a schedule can be maintained in no more time than the desired operations themselves. Raman [18] discusses other methods for eliminating amortization.

6 Lists with Catenation

To illustrate our technique, we next present an extended example: persistent lists that support catenation and all other usual list primitives in constant amortized time. In the normal linked representation of lists, catenation requires time linear in the size of the first list. While it is not difficult to design alternative representations of lists that support constant time catenation (see, for example, [11]), such alternative representations seem almost inevitably to sacrifice constant time behavior in one or more of the other usual list primitives. Balanced trees can easily support all list operations, including catenation, in logarithmic time [16]. Driscoll, Sleator, and Tarjan [5] and Buchsbaum and Tarian [3] investigated rather complicated sublogarithmic implementations. Kaplan and Tarjan [12] finally achieved constant worst-case bounds for this problem, but their solution is still fairly complicated. Our data structure is much simpler, but is efficient in the amortized, rather than worst-case, sense.

A catenable list is one that supports the following operations:

- makelist(x): Create a new list containing the element x.
- first(L): Extract the first element of list L.
- pop(L): Return a new list containing all the elements of L except the first.
- $catenate(L_1, L_2)$: Construct a new list containing the elements of L_1 followed by the elements of L_2 .

The familiar push and inject operations, which add elements to the front and rear of a list, are subsumed by makelist and catenate since they can be simulated by catenate(makelist(x), L) and catenate(L, makelist(x)). Since a catenable list supports insertion at both ends, but removal only from the front, it is more properly regarded as a (catenable) output-restricted deque.

We represent catenable lists as n-ary trees containing at most one element in each node. Nodes containing an element are called occupied, nodes without an element are called vacant. Vacant nodes are required to have at least two children; such nodes represent delayed computations arising during pop's. The order of elements within the list is determined by the left-to-right preorder traversal of the tree, ignoring vacant nodes. Hence, the first element of the list is contained in the first occupied node along the leftmost path from the root. See Figure 1 for a sample tree.

The two fundamental operations on trees are link and force. The former is the primitive for catenating trees, while the latter restructures trees to guarantee that the root is occupied. These operations are mutually recursive. Given two non-empty trees T_1 and T_2 , $link(T_1, T_2)$ first forces T_1 and then adds T_2 as the

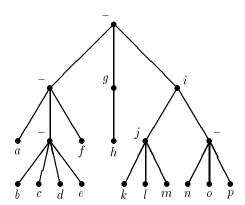


Figure 1: A tree representing the list $[a \dots p]$. Vacant nodes are indicated by dashes (-).

rightmost child of T_1 . The children of a given node are stored in a queue, from left to right, using the purely functional constant-time queues of Hood and Melville [9] or Okasaki [17]. Thus, $link(T_1, T_2)$ merely adds T_2 to the end of the child queue of T_1 .

If the root of T is occupied, then force(T) does nothing. Otherwise, the root of T is a vacant node with children $T_1 \dots T_m$. If m = 2, then force(T) links T_1 and T_2 . If m > 2, then force(T) creates a new vacant node with children $T_2 \dots T_m$ and links T_1 with the new node. In either case, we memoize the result by overwriting the original vacant node with the new tree. Since force may call link, and vice versa, a single call to force may result in a cascade of changes: if the leftmost path from the root begins with several vacant nodes, then forcing the root will force each of those nodes. Figure 2 illustrates this process. Forcing is closely related to a form of path compression known as path reversal [22, 8], augmented with memoization at each intermediate stage to support persistence efficiently.

The purpose of vacant nodes is to implement lazy linking. A vacant node with children $T_1 \ldots T_m$ represents the delayed computation of

$$link(T_1, link(T_2, \ldots, link(T_{m-1}, T_m) \ldots))$$

This computation is executed by forcing the vacant node. Note, however, that lazy linking is performed incrementally since forcing a vacant node executes only the outermost link, delaying the remaining computation of $link(T_2, \ldots, link(T_{m-1}, T_m) \ldots)$ within another vacant node.

We can now describe the list operations: makelist simply creates a new occupied node with no children; first forces the root and returns its contents; pop forces and discards the root, lazily linking its children by col-

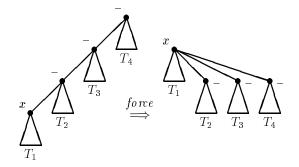


Figure 2: Illustration of *force*. The leftmost path of vacant nodes is compressed to raise the first element to the root.

lecting them under a new vacant node; and, finally, catenate verifies that both lists are non-empty and links the two trees. Psuedocode for each of these operations is given in Figure 3.

Although we have not described these operations in a purely functional manner (particularly force), they are all straightforward to implement in any functional language supporting lazy evaluation. Source code written in Standard ML [15] is available through the World Wide Web from

http://foxnet.cs.cmu.edu/people/cokasaki/catenation.html

6.1 Analysis

By inspection of Figure 3, every operation clearly runs in constant worst-case time, excluding the time spent forcing vacant nodes. We next show that allocating three credits per *pop* suffices to pay for each lazy link (*i.e.*, discharge each associated debit) before it is executed, establishing the constant amortized bound.

For reasoning about these algorithms, it is often more convenient to think in terms of reference trees than the actual trees generated by the algorithms. The reference tree corresponding to a given actual tree is that same tree as it would appear if every lazy link were executed (i.e., if every vacant node were forced). Thus, in a reference tree, every node is occupied. Figure 4 illustrates the effects of the pop operation in terms of reference trees. The value of reference trees is that they are unchanged by force; that is, if T' = force(T), then both T and T' have the same reference tree. In particular, when we speak of the depth of a given node, we shall be concerned with its depth in the reference tree corresponding to its actual tree (a fixed quantity), rather than its actual depth (which may increase or decrease with every force).

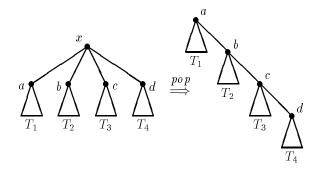


Figure 4: Illustration of pop in terms of reference trees.

Recall that the *pop* operation on a tree with root x and children $T_1 ldots T_m$ effectively deletes x and lazily links $T_1 ldots T_m$ in the following pattern:

$$link(T_1, link(T_2, \ldots, link(T_{m-1}, T_m) \ldots))$$

To account for these m-1 links, we assign a total of m-1 debits to the nodes that will become the roots of each partial result. Specifically, the *i*th debit is charged to the first occupied node of T_i . In addition, every pop operation discharges the first three debits in the tree (that is, those three debits whose nodes are earliest in preorder). No other list operation either creates or discharges debits.

Let $d_T(i)$ be the number of (undischarged) debits on the *i*th occupied node of tree T, and let $D_T(i) = \sum_{j=1}^i d_T(j)$. $D_T(i)$ is called the *cumulative debt* of node i and reflects all the debits that must be discharged before the *i*th element can be accessed. In addition, let D_T be the total debt of all nodes in T. To show that the *force*'s at the beginning of each pop, first, and catenate require only constant amortized time, we must prove that $D_T(1) = 0$ (i.e., that all the lazy links involving the first occupied node have been paid for). To allow for a series of pop's, each of which may discharge only a constant number of debits, we also require that $D_T(i)$ be linear in i. We satisfy both of these requirements by maintaining the following invariant, called the *left-linear debt invariant*:

$$D_T(i) < i + depth_T(i)$$

where $depth_T(i)$ is the depth of the *i*th node in the reference tree corresponding to T. The intuition for this invariant is that we only manipulate the tree at or near the root; nodes deep in the tree are somehow carried along for free with their shallower ancestors. Thus, we wish to reward greater depth by allowing those nodes a greater cumulative debt.

```
function link(T_1,T_2) = /* make T_2 last child of T_1 */
     force(T_1);
     let x = contents(T_1);
     let q = children(T_1);
     let q' = Q.inject(T_2,q);
     return new node with contents x and children q'.
function fore(T) = /* guarantee root is occupied */
     if T is occupied then return;
     let q = children(T);
     let T_1 = Q.first(q);
     let q' = Q.pop(q);
     if Q.size(q) = 2 then
          let T_2 = Q.first(q');
     else /* Q.size(q) > 2 */
         let T_2 = new vacant node with children q';
     T := link(T_1, T_2); /* execute delayed link and memoize result */
     return.
function makelist(x) =
     return new node with contents x and no children.
function first(T) =
     if T is empty then error;
     force(T);
     return contents(T).
function pop(T) =
     if T is empty then error;
     force(T);
     let q = children(T);
     if Q.size(q) = 0 then return empty list;
     elsif Q.size(q) = 1 then return Q.first(q);
     else return new vacant node with children q.
                                                      /* set up lazy links */
function catenate(T_1,T_2) =
     if T_1 is empty then return T_2;
     elsif T_2 is empty then return T_1;
     else return link(T_1,T_2).
```

Figure 3: Psuedocode for catenable lists. The functions Q.inject, Q.first, Q.pop, and Q.size operate on queues rather than catenable lists.

In proving that *catenate* preserves the left-linear debt invariant, we will also require that every nonempty tree and subtree produced by the above list operations satisfies the *total debt invariant*:

$$D_T < |T|$$

where |T| denotes the number of occupied nodes in T.

Theorem 1 catenate and pop maintain both the left-linear debt invariant and the total debt invariant.

Proof: (catenate) Assume T_1 and T_2 are non-empty, and let $T = catenate(T_1, T_2)$. Let $n = |T_1|$. Note that the index, depth, and cumulative debt of nodes from T_1 are unaffected by the catenation, so $D_T(i) = D_{T_1}(i)$ satisfies the left-linear debt invariant for $i \leq n$. However, nodes in T_2 accumulate the total debt of T_1 and increase in depth by one. Thus,

$$D_{T}(n+i) = D_{T_{1}} + D_{T_{2}}(i)$$

$$< n + D_{T_{2}}(i)$$

$$< n + i + depth_{T_{2}}(i)$$

$$< (n+i) + depth_{T}(n+i)$$

Finally, $D_T = D_{T_1} + D_{T_2} < |T_1| + |T_2| = |T|$.

(pop) Let T' = pop(T). After discarding the root of T, we combine the subtrees $T_1 \ldots T_m$, each of which initially satisfies the total debt invariant. The subtrees of T' (including T' itself) that are not subtrees of T are each of the form

$$T'_i = link(T_i, \ldots, link(T_{m-1}, T_m) \ldots)$$

(where, of course, the links are delayed). For instance, in Figure 4, these are the subtrees rooted at a, b, and c. Each such subtree is assigned m - j new debits. However, each T_k contains $\leq |T_k| - 1$ debits. Thus,

$$D_{T'_{j}} = m - j + D_{T_{j}} + \dots + D_{T_{m}}$$

$$\leq m - j + (|T_{j}| - 1) + \dots + (|T_{m}| - 1)$$

$$= |T'_{j}| - 1$$

$$< |T'_{j}|$$

Finally, we show that pop preserves the left-linear debt invariant to within three debits. Suppose the ith node of T appears in T_j . We know that $D_T(i) < i + depth_T(i)$, but consider how each of these three quantities changes with the pop. i decreases by 1 since the root of T is discarded. The depth of each node in T_j increases by j-2 (see Figure 4) while the cumulative debt of each node in T_j increases by j (except for nodes in T_m , whose cumulative debts only increase by

m-1). Thus,

$$\begin{array}{lll} D_{T'}(i-1) &=& D_{T}(i)+j\\ &<& i+depth_{T}(i)+j\\ &=& i+(depth_{T'}(i-1)-(j-2))+j\\ &=& (i-1)+depth_{T'}(i-1)+3 \end{array}$$

Using the three credits allocated to pop to discharge the first three debits in the tree restores the invariant.

Theorem 1 establishes the constant amortized bounds of pop, catenate, and first. Shewchuk [19] has developed an alternative proof based on the potential method of Sleator [21], using a potential function of twice the number of critical nodes, where a critical node is a vacant leftmost child of a vacant leftmost child.

If desired, catenate and first (but not pop) can be made to run in constant worst-case time by calling force at the end of every pop, rather than at the beginning of every operation. This guarantees that no root is ever vacant. Alternatively, catenate can be made to run in constant worst-case time by executing its single link lazily, that is, by creating a new vacant node with its two arguments as children.

6.2 Related Work

Driscoll, Sleator, and Tarjan [5] first studied the problem of catenation for persistent lists. They represent catenable lists as n-ary trees with the elements at the leaves. To keep the leftmost leaves near the root, they use a restructuring operation known as pull that removes the first grandchild of the root and reattaches it directly to the root. Unfortunately, catenation breaks all useful invariants based on this restructuring heuristic, so they are forced to develop quite a bit of machinery to support catenation. The resulting data structure supports catenation in $O(\log \log k)$ worst-case time, where k is the number of list operations (note that k may be much smaller than n), and all other operations in constant worst-case time.

Buchsbaum and Tarjan [3] use data-structural bootstrapping to recursively decompose catenable deques of size n into catenable deques of size $O(\log n)$. They use the pull operation of Driscoll, Sleator, and Tarjan to keep the tree balanced (of depth $O(\log n)$), and then use the smaller deques to represent the leftmost and rightmost paths of each subtree. This yields a data structure that supports deletion of the first or last element in $O(\log^* k)$ worst-case time, and all other operations in constant worst-case time.

Finally, Kaplan and Tarjan [12] recently discovered an implementation of catenable lists based on the principle of recursive slow-down. A catenable list is represented recursively as two (non-catenable) deques, together with a catenable list whose elements include other catenable lists and (non-catenable) deques. One deque represents the prefix of the list, the other the suffix. The inner catenable list represents the middle of the outer list. Since this representation is recursive, the representation of the inner list itself contains an inner list and so on. Operations on the outer list may affect the inner lists, but recursive slow-down prevents cascades of changes. This data structure supports all operations in constant worst-case time.

In contrast, our data structure supports all operations in constant amortized time. The worst-case bounds of Kaplan and Tarjan may be preferable in some circumstances, such as real-time or parallel programming, but if amortized bounds are acceptable, then the simplicity of our data structure should be appealing. The next challenge is to implement catenable deques supporting all operations in constant time. Kaplan and Tarjan have successfully extended their solution for lists to catenable deques [12], and our approach seems likely to yield a solution as well.

7 Conclusion

We have presented a framework for designing persistent (in fact, purely functional) data structures with good amortized efficiency. The key feature of our technique is the use of lazy evaluation to share the results of expensive computations across multiple "futures". Amortization has been largely ignored in the design of persistent data structures, perhaps because the role of lazy evaluation was previously unappreciated. Our technique should apply to any data structure for which it is possible to predict an expensive computation far enough in advance to set up the computation lazily and then pay for it by the time it is needed. Incremental functions play an important role in this process, since they allow large computations to be decomposed into smaller fragments, each of which can be "purchased" separately.

As an example of our technique, we have described an implementation of purely functional catenable lists that support catenation and all other list primitives in constant amortized time. There are many potential applications of this data structure, including the implementation of full functional jumps [6] and support for the Bird-Meertens formalism [2, 13], in which append (catenate) is often given preference over cons (push). Other data structures that can be made per-

sistent using our technique include queues and deques [17] and priority queues.

Acknowledgements

Thanks to Bob Tarjan for pointing out the connection to path reversal. Thanks also to Jonathan Shewchuk, Sasha Wood, and Mark Leone for their comments and suggestions on an earlier draft of this paper.

References

- [1] Richard Bellman. Dynamic Programming. Princeton University Press, 1957.
- [2] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, Logic of Programming and Calculi of Discrete Design, pages 5-42. Springer-Verlag, 1987.
- [3] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. In Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 155-164, 1993.
- [4] 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.
- [5] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943-959, September 1994.
- [6] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: a mathematical semantics for handling full functional jumps. In Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, pages 52-62, 1988.
- [7] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596-615, July 1987.
- [8] David Ginat, Daniel D. K. Sleator, and Robert E. Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 39(1):3-5, April 1989.

- [9] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50-53, November 1981.
- [10] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. SIAM Journal on Computing, 2(4):294-303, December 1973.
- [11] R. John Muir Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141-144, March 1986.
- [12] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In Proceedings of the 27th Annual ACM Symposium on the Theory of Computing, pages 93-102, 1995.
- [13] Lambert Meertens. Algorithmics towards programming as a mathematical activity. In Proceedings of the CWI Symposium on Mathematics and Computer Science, pages 289-334, 1986.
- [14] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.
- [15] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML. The MIT Press, Cambridge, Massachusetts, 1990.
- [16] Eugene W. Myers. Efficient applicative data

- types. In Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pages 66-75, 1984.
- [17] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, October 1995. To appear.
- [18] Rajeev Raman. Eliminating Amortization: On Data Structures with Guaranteed Response Times. PhD thesis, Department of Computer Sciences, University of Rochester, 1992.
- [19] Jonathan R. Shewchuk. Private communication, April 1995.
- [20] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting heaps. SIAM Journal on Computing, 15(1):52-69, February 1986.
- [21] Robert E. Tarjan. Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods, 6(2):306-318, April 1985.
- [22] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245-281, April 1984.
- [23] Christopher Van Wyk and Jeffrey Scott Vitter. The complexity of hashing with lazy deletion. *Algorithmica*, 1(1):17-29, 1986.