Persistent Lists with Catenation via Recursive Slow-Down

Haim Kaplan* Robert E. Tarjan†

Abstract

We describe an efficient purely functional implementation of stacks with catenation. In addition to being an intriguing problem in its own right, functional implementation of catenable stacks is the tool required to add certain sophisticated programming constructs to functional programming languages. Our solution has a worst-case running time of $O(1)$ for each push, pop, and catenation. The best previously known solution has an $O(\log^* k)$ time bound for the k^{th} stack operation. Our solution is not only faster but simpler, and indeed we hope it may be practical.

The major new ingredient in our result is a general technique that we call recursive slow-down. Recursive slow-down is an algorithmic design principle that can give constant worst-case time bounds for operations on data structures. We expect this technique to have additional applications. Indeed, we have recently been able to extend the result described here to obtain a purely functional implementation of double-ended queues with catenation that takes constant time per operation.

STOC '95, Las Vegas, Nevada, USA
© 1995 ACM 0-89791-718-9/95/0005..$3 50

*Department of Computer Science, Princeton University, Princeton, NJ 08544 USA. Research supported by the Office of Naval Research, Contract No. N00014-91-J-1463 and a United States-Israel Educational Foundation (USIEF) Fulbright Grant. hkl@cs.princeton.edu.

†Department of Computer Science, Princeton University, Princeton, NJ 08544 USA and NEC Institute, Princeton, NJ. Research at Princeton University partially supported by the NSF, Grant No. CCR-8920505 and the Office of Naval Research, Contract No. N00014-91-J-1463. ret@cs.princeton.edu.

1 History of the Problem

A *persistent* data structure is one in which a change to the structure can be made without destroying the old version, so that all versions of the structure persist and can be accessed or (possibly) modified. In the functional programming literature, persistent structures are often called *immutable*. Purely functional[1] programming, without side effects, has the property that every structure created is automatically persistent. Persistent data structures arise not only in functional programming but also in text, program, and file editing and maintenance; computational geometry; and other algorithmic application areas. (See [5, 8, 9, 10, 11, 12, 13, 14, 21, 28, 29, 30, 31, 32, 33, 35].)

Several papers have dealt with the problem of adding persistence to general data structures in a way that is more efficient than the obvious solution of copying the entire structure whenever a change is made. In particular, Driscoll, Sarnak, Sleator, and Tarjan [11] described how to make pointer-based structures persistent using a technique called *node-splitting*, which is related to fractional cascading [6] in a way that is not yet fully understood. Dietz [10] described a method for making array-based structures persistent. Additional references on persistence can be found in those papers.

The general techniques in [10] and [11] fail to work on data structures that can be combined with each other rather than just be changed locally. Perhaps the simplest and probably the

[1] For the purposes of this paper, a "purely functional" data structure is one built using only the LISP functions car, cons, cdr. Though we do not state our constructions explicitly in terms of these functions, it is routine to verify that our structures are purely functional.

most important example of combining data structures is catenation of lists. This paper deals with the problem of making persistent list catenation efficient.

In order to describe our work and related results in detail, we need a little terminology. We consider the following operations for manipulating lists:
makelist(x): return a new list consisting of the singleton element x.
push(x, L): return the list that is formed by adding element x to the front of list L.
pop(L): return the pair consisting of the first element of list L and the list consisting of the second through last elements of L.
inject(x, L): return the list that is formed by adding element x to the back of list L.
eject(L): return the pair consisting of the last element on list L and the list consisting of the first through next-to-last elements of L.
catenate(K, L): return the list formed by catenating K and L, with K first.

In accordance with convention, we call a list subject only to *push* and *pop* a *stack* and a list subject only to *inject* and *pop* a *queue*. Adopting the terminology of Knuth [22], we call a list subject to all four operations *push*, *pop*, *inject*, and *eject* a *double-ended queue*, or *deque* (pronounced "deck"). In a departure from existing terminology, we call a list subject only to *push, pop*, and *inject* a *stack-ended queue*, or *steque* (pronounced "steck"). Knuth called steques *output-restricted deques*, but "stack-ended queue" is both easy to shorten and evokes the idea that a steque combines the functionalities of a stack and a queue.

Putting aside catenation for the moment, let us consider the problem of making non-catenable lists persistent. It is easy to make stacks persistent: we represent a stack by a pointer to a singly-linked list of its elements. To push an element onto a stack, we create a new node containing the new element and a pointer to the node containing the previously first element on the stack. To pop a stack, we retrieve the first element and a pointer to the node containing the previously second element.

A collection of stacks represented in this way consists of a collection of trees, with a pointer from each child to its parent. Two stacks with common suffixes can share one list representing the common suffix. (Having common suffixes does not guarantee this sharing, however, since two stacks identical in content can be built by two separate sequences of *push* operations. Maximum suffix sharing can be achieved by using a "hashed consing" technique in which a new node is created only if it corresponds to a distinct new stack. See [1, 34].)

Making a queue, steque, or deque persistent is not so simple. One approach, which has the advantage of giving a purely functional solution, is to attempt to represent such a data structure by a fixed number of stacks and to use the method described above to implement the stacks. The problem of representing a deque by a fixed number of stacks so that each deque operation becomes a constant number of stack operations is closely related to an old problem in Turing machine complexity, that of giving a real-time simulation of a (one-dimensional) multihead tape unit by a fixed number of (one-dimensional) single-head tape units: a k-head tape unit can be simulated by two stacks and $k - 1$ deques; a deque can be simulated by a two-head tape unit; a stack can be simulated by a one-head tape unit. Fisher, Meyer, and Rosenberg [15] used a tape-folding technique to simulate a multihead tape by a set of one-head tapes in real time. Leong and Seiferas [25] proposed an alternative method based on a linear-time simulation of Stoss [35]. Their method generalizes to multi-dimensional tape units.

Although the Fisher-Meyer-Rosenberg and Leong-Seiferas methods can both be adapted to solve the problem of simulating a deque by stacks, it is conceptionally simpler to deal with the deque

simulation problem directly. There are several works in the literature dealing with this problem [4, 7, 16, 17, 18, 19, 20, 26, 31], all of which use two key ideas. The first is that a deque can be represented by a pair of stacks, one representing the front part of the deque and the other representing the reversal of the rear part. When one stack becomes empty because of too many *pop* or *eject* operations, the deque, now all on one stack, is copied into two stacks each containing half of the deque elements. This fifty-fifty split guarantees that such copying, even though expensive, happens infrequently. A simple amortization argument shows that this gives a linear-time simulation of a deque by a constant number of stacks: k deque operations starting from an empty deque are simulated by $O(k)$ stack operations. This simple idea is the essence of Stoss's tape simulation [35]. The idea of representing a queue by two stacks in this way appears in [4, 17, 19]; this representation of a deque appears in [16, 18, 20, 31].

The second idea is to use incremental copying to convert this linear-time simulation into a real-time simulation: as soon as the two stacks become sufficiently unbalanced, recopying to create two balanced stacks begins. Because the recopying must proceed concurrently with deque operations, which among other things causes the size of the deque to be a moving target, the details of this simulation are a little complicated. Hood and Melville [19] first spelled out the details of this method for the case of a queue; Hood's thesis [18] describes the simulation for a deque. See also [16, 31]. Chuang and Goldberg [7] give a particularly nice description of the deque simulation. Okasaki [26] gives an variation of this simulation that uses "memoization" to avoid some of the explicit stack-to-stack copying; his solution gives persistence but is not purely functional since memoization is a side effect.

A completely different way to make a deque persistent is to apply the general mechanism of Driscoll, et. al [11], but this solution, too, is not purely functional, and the constant time-bound per deque operation is amortized, not worst-case.

Once catenation is added as an operation, the problem of making stacks or deques persistent becomes much harder; all the methods mentioned above fail. A straightforward use of balanced trees gives a representation of persistent catenable deques in which an operation on a deque or deques of total size n takes $O(\log n)$ time. Driscoll, Sleator, and Tarjan [12] combined a tree representation with several additional ideas to obtain an implementation of persistent catenable stacks in which the k^{th} operation takes $O(\log\log k)$ time. Buchsbaum and Tarjan [2] used a recursive decomposition of trees to obtain two implementations of persistent catenable deques. The first has a time bound of $2^{O(\log^* k)}$ and the second a bound of $O(\log^* k)$ for the k^{th} operation, where $\log^* k$ is the iterated logarithm, defined by $\log^{(1)} k = \log_2 k, \log^{(1)} k = \log\log^{(i-1)} k$ for $i > 1$, and $\log^* k = \min\{i \mid \log^{(i)} k \leq 1\}$.

2 Overview of Our Work

The spark for our work was an observation concerning the recurrence that gives the time bounds for the Buchsbaum-Tarjan data structures. That recurrence has the following form:

$$T(n) = O(1) + cT(\log n),$$

where c is a constant: an operation on a structure of size n takes a constant amount of time plus a fixed number of operations on recursive substructures of size $\log n$. In the first version of the Buchsbaum-Tarjan structure, c is a fixed constant greater than one, and the recurrence gives the time bound $T(n) = 2^{O(\log^* n)}$. In the second version of the structure, c equals one, and the recurrence gives the time bound $T(n) = O(\log^* n)$.

But suppose that we could design a structure in which the constant c were less than one. Then the

recurrence would give the bound $T(n) = O(1)$. Indeed, the recurrence $T(n) = O(1) + cT(n-1)$ gives the bound $T(n) = O(1)$ for any constant $c < 1$, such as $c = \frac{1}{2}$.

This means that we can obtain an $O(1)$ time bound for operations on a data structure if each operation requires $O(1)$ time plus half an operation on a smaller recursive substructure. But how do we perform half an operation? Computer time, after all, at least on digital computers, is discrete, not continuous, and cannot be divided into arbitrarily small parts. We can achieve the desired effect, however, if our data structure requires only **one** operation on a recursive substructure for every **two** operations on the top-level structure. We call this idea *recursive slow-down*. By adding recursive slowdown to a modified (and simplified) version of the Buchsbaum-Tarjan structure, we are able to obtain an $O(1)$-time, purely functional (and hence persistent) implementation of stacks with catenation.

The main novelty in our data structure is the mechanism for implementing recursive slowdown. Stated abstractly, the problem that we must solve is to allocate work cycles to the levels of a linear recursion so that the top level gets half the cycles, the second level gets one quarter of the cycles, the third level gets one eighth of the cycles and so on. As it happens *binary counting* has exactly the right properties to serve as a work allocation mechanism. Specifically, if we begin with zero and repeatedly add one in binary, each addition of one causes a unique bit position to change from zero to one. In every second addition this position is the one's bit, in every fourth addition it is the two's bit, in every eighth addition it is the four's bit, and so on. Binary counting can be implemented using a stack of stacks to represent blocks of consecutive zeros and ones and hence can be performed purely functionally; this gives the tool we need to implement recursive slowdown.

In our application (to persistent stacks with catenation), there is another and perhaps simpler way to view the binary counting mechanism. Each level of the recursive data structure has a *color*, green, yellow, or red, based on the state of the structure at that level. A red structure is bad but can be converted to a green structure at the cost of degrading the structure one level deeper in the recurrence from green to yellow or from yellow to red. We maintain the invariant on the recursion levels that red and green alternate, separated by arbitrary numbers of yellows.

Section 3 describes the use of recursive slowdown to make deques without catenation persistent, thereby illustrating the use of recursive slowdown in a simple setting. Section 4 describes its use to make stacks (or steques) with catenation persistent.

3 Persistent Deques without Catenation

3.1 Representation

Define a data structure to be *over a set A* if it stores elements from A. Let a 5-deque be a deque that can contain no more then 5 elements. For the purposes of this section *prefix* and *suffix* will denote 5-deques. Consider the following recursive representation of a deque d of elements from a universe A. Deque d is either empty or a triple consisting of a prefix over A, denoted by prefix(d), a deque over $A \times A$, denoted by $c(d)$, and a suffix over A, denoted by suffix(d). Deque d is ordered as follows: First are the elements of prefix(d) (ordered as they are in the prefix). Next are the elements of the pairs stored in $c(d)$ ordered consistently with the order of $c(d)$. The first component of each pair precedes the second. Last are the elements of suffix(d) (ordered as they are in the suffix).

A prefix or suffix is *green* if it has two or three elements. It is *yellow* if it has one or four el-

ements. It is *red* if it has zero or five elements. Order the colors such that $red < yellow < green$. The color of a deque $d \neq \emptyset$ for which $c(d) \neq \emptyset$ is defined as the minimum of its prefix and suffix colors. The color of a deque $d \neq \emptyset$ for which $c(d) = \emptyset$ is defined as the minimum of its prefix and suffix colors if both are not empty. Otherwise it is the color of the one which is not empty.

The representation suggested above for a deque d is actually a stack $S(d)$ in which the i-th element stores pointers to prefix($c^i(d)$) and suffix($c^i(d)$) for each non-empty $c^i(d)$.

The representation of d that our implementation will actually use in this section is a stack $S'(d)$ constructed from $S(d)$ as follows: Every maximal sequence $s_i, s_{i+1}, \ldots, s_{i+k}$ of elements corresponding to yellow deques in $S(d)$ is replaced by a single element $s_{i,k}$ in $S'(d)$. Element $s_{i,k}$ will contain a pointer to a length k secondary stack containing the yellow deques $s_i, \ldots, s_{i+k}$, with s_i on top. This representation allows us to access the topmost non-yellow deque in $S'(d)$ easily in the persistent setting.

3.2 Operations

We will maintain the following invariant:

Invariant 3.1 If $c^i(d)$ is a red deque then $i > 0$ and there exists a green deque $c^j(d)$, $j < i$ such that any deque $c^k(d)$, $j < k < i$ is yellow.

In particular this invariant implies that d is not red. It will always be possible to push or pop an element to/from prefix(d) and inject or eject an element to/from suffix(d) possibly degrading its color from green to yellow or from yellow to red.

After doing such an operation the invariant might be violated. We restore the invariant by doing a *fix* step defined as follows: Let $x = c^i(d)$ be the topmost non-yellow deque in $S(d)$. If x is red make x green by doing one or two operations on $c(x)$ as follows:

a) $c(x)$ is empty.

   a.1) $5 \leq |x| \leq 6$. Balance the prefix and the suffix such that there are two or three elements in each by moving elements from one to the other.

   a.2) $6 < |x| \leq 10$. If the prefix is of length 4-5 then push the last two elements of the prefix into $c(x)$. If the suffix has length 4-5 then inject the first two elements in the suffix into $c(x)$.

b) $c(x)$ is not empty.
   If the prefix length is 0-1 then pop a pair from $c(x)$ into the prefix. If the prefix length is 4-5 then push a pair from the prefix into $c(x)$. If the suffix length is 4-5 inject a pair from the suffix into $c(x)$. If the suffix length is 0-1 and $c(x)$ is not empty then eject a pair from $c(x)$ into the suffix.

3.3 Persistent Setting

We use $S'(d)$ in order to be able to perform the fix operation defined above in constant time. The result of the fix is a new deque d' represented by a stack $S'(d')$. The first non-yellow element in $S(d)$, x, is either the top element in both $S(d)$ and $S'(d)$ or the second element in $S'(d)$. Changes to $S'(d)$ are performed only if x is red. Otherwise $d' = d$. Let $s_0, \ldots, s_n$ be the elements in $S'(d)$ and $t_0, \ldots, t_n$ the elements in $S'(d')$. Assume that s_0 points to a stack of yellow deques and s_1 points to x which is red (the case in which s_0 points to x and there is no secondary yellow stack on top is handled similarly).

Assume first that $c(x)$ is green and represented by s_3. If its color is not changed by the fix then we only need to create t_0, t_1 and t_2. Set t_0 to point to the same secondary yellow stack to which s_0 points. Elements t_1 and t_2 will represent the new versions of x and $c(x)$. The bottom of $S'(d')$ is

identical to the bottom of $S'(d)$, i.e. $s_i = t_i$, for every $i \geq 3$. The fix may cause $c(x)$ to have a color degraded to yellow. In such a case, if s_4 points to a secondary yellow stack the modified $c(x)$ should be pushed on top of it. Element t_3 is set to point to the secondary stack obtained after the push. Other elements on the bottom do not change, i.e. $t_i = s_{i+1}$ for $i > 3$. The number of elements in $S'(d')$ is one less than the number of elements in $S'(d)$. If s_4 does not point to a yellow stack then a new stack must be created with $c(x)$ in it; t_3 points to it.

Another possibility is that $c(x)$ is yellow. Then it must be the top element in a secondary stack pointed to by s_3. The operations needed to create $S'(d')$ are similar and we omit the details here.

In any case in order to obtain $S'(d')$ only a constant number of elements on top of $S'(d)$ need to be replaced. There is also a possible change in some secondary yellow stack (push,pop or both (replace)).

A change to a prefix or a suffix requires copying the 5-deque before the change in order to make the change nondestructive.

4 Constant-Time Catenation

In this section we show how to implement persistent catenable stack-ended-queues (c-steques). The operations possible on a c-steque are inject(x,d), pop(d), push(x,d), catenate(d_1, d_2). Our implementation requires constant time and space for each of these operations.

4.1 Representation

In this section a *prefix* and a *suffix* will denote non-catenable deques. A prefix will always have at least two elements in it.

A c-steque d over A will be represented by either a suffix over A or a triple: a prefix over A (prefix(d)), a c-steque of *elements* (c(d)), and a suffix over A (suffix(d)). An *element* is either a prefix or a pair (prefix over A, c-steque of elements).

Define a prefix to be *green* if its cardinality is at least 4, *yellow* if it is 3 and red if it is 2. The *color of a c-steque* is the color of its prefix if it has one; otherwise, it is green.

The representation suggested above is actually a stack $S(d)$ as described in the previous section. A modified stack $S'(d)$ will be used as our representation for a c-steque d. The details concerning using $S'(d)$ instead of $S(d)$ appear in Section 4.3.

4.2 Operations

Invariant 4.1 If $c^i(d)$ is a red c-steque then $i > 0$ and there exists a green c-steque $c^j(d)$, $j < i$ such that any c-steque $c^k(d)$, $j < k < i$ is yellow. Every c-steque stored in an element of $c^i(d)$ for some $i \geq 0$ satisfies the same requirement except that there might not be a green c-steque above the topmost red c-steque.

Let d be a c-steque represented as defined above. Perform $push(x, d)$ by pushing x into prefix(d), or into suffix(d) if d has no prefix. $inject(x, d)$ injects x into suffix(d).

Let d_1, d_2 be steques. The operation $d :=$ $catenate(d_1, d_2)$ is performed as follows:

algorithm CATENATE(d_1, d_2)
if |suffix(d_1)| < 2 **then**
  if suffix(d_1) is not empty **then**
    Let x be the only element in suffix(d_1)
    if prefix(d_2) exists **then** push(x,prefix(d_2))
    else push(x, suffix(d_2))
  endif
  if prefix(d_1) exists **then**
    prefix(d) := prefix(d_1)
    if prefix(d_2) exists **then**

```
        inject((prefix(d2),c(d2)),c(d1))
    c(d) := c(d1)
  else if prefix(d2) exists then
        prefix(d) := prefix(d2)
        c(d):=c(d2)
       endif
  endif
else /* |suffix(d1)| ≥ 2 */
  if d1 has no prefix then
    if |suffix(d1)| ≥ 4
      prefix(d) := suffix(d1)
      makesteque(c(d)) /* create an empty c-steque */
      if prefix(d2) exists then
          inject((prefix(d2),c(d2)),c(d))
    else
      Push the elements (at most 3)
      in suffix(d1) onto prefix(d2)
      prefix(d) := prefix(d2)
      c(d):=c(d2)
    endif
  else
    prefix(d) := prefix(d1)
    inject(suffix(d1),c(d1))
    /* Note: a suffix with at least two elements
    could be considered a prefix */
    if prefix(d2) exists then
        inject((prefix(d2),c(d2)),c(d1))
    c(d) := c(d1)
  endif
suffix(d) := suffix(d2).
```

Note that if d_1 has a non-empty c-steque $c(d_1)$, then $S(d)$ will be $S(d_1)$ with modified top two elements (but without changes in colors). $S(d_2)$ without its top element gets stored as $c(d_2)$. In case d_1 has no $c(d_1)$, then c-steque $S(d)$ is either $S(d_2)$ with a modified top element which possibly has its color changed from yellow to green or a new stack with two green entries. The following lemma is easy to verify.

Lemma 4.2 Let d=catenate(d_1, d_2). d satisfies Invariant 4.1 if d_1 and d_2 do.

The operation $(x, d) := pop(d')$ is performed by popping the prefix of d', or its suffix if it has no prefix. The invariant guarantees that it is possible to do this. Doing this can cause d to violate the invariant. We restore the invariant by performing $fix(d)$ after the pop. The second component of the result of the pop is $fix(d)$, defined as follows:

```
algorithm FIX(d)
Let d_i be the first non-yellow c-steque in S(d).
Let d_{i+1} = c(d_i).
if d_i is red then
  if d_{i+1} is empty then push two elements from the
  prefix into the suffix.
  (make d_i a c-steque with suffix only.)
else
  (y, d_{i+1}) := pop(d_{i+1})
  let y= prefix_1 or y= (prefix_1, csteque_1)
  push the elements from d_i's prefix into prefix_1
  Let prefix_1 be the new prefix of d_i
  if y is a pair then d_{i+1} = catenate(csteque_1, d_{i+1}))
endif
```

Theorem 4.3 Let d be a c-steque after a pop and let $d' = fix(d)$. Then d' satisfies invariant 4.1.

Proof. Let d_i denote the i-th element in $S(d)$ and d'_i denote the i-th element in $S(d')$. Note that the only violation of the invariant which a single *pop* can cause is a missing green c-steque before the first red in $S(d)$. If that indeed happened then the color of d_i in the Algorithm $fix(d)$ will be red. After the fix, d'_i, the new version of d_i, is green and $d'_k = d_k$, $k < i$. Thus there is a green c-steque before the first red c-steque in $S(d')$. The bottom of $S(d')$, i.e $\{d'_j \mid j > i+1\}$ is identical either to the bottom of $S(d)$ before the fix or to the bottom of the retrieved csteque$_1$. In any case two red c-steques d'_k and d'_l with $l, k > i+1$ are separated by a green c-steque. Assume d'_{i+1} is red and let d'_j be the next red c-steque in $S(d')$ if there is one. If the bottom of $S(d')$ is identical to the bottom of $S(d)$ then there is a green c-steque d'_k, $i+1 < k < j$ since there is one between d_i and d_j in $S(d)$ and it is not d_{i+1} since its color must be yellow. Otherwise, $d'_{i+1} = \text{csteque}_1$ was stored red and according to Invariant 4.1 there should have been green steque d_l, $0 < l < j - i - 1$ in $S(csteque_1)$. Thus we have a green csteque

d'_{l+i+1}, $i+1 < l+i+1 < j$. ■

4.3 Persistent Setting

For a c-steque d, prefix(d) and suffix(d) are implemented as persistent non-catenable deques using the method in Section 3 (or using the stack-reversing method described in Section 1). As already mentioned we use $S'(d)$ as our actual representation. The key observation is that catenate(d_1, d_2) makes changes to no more then the top two elements in $S'(d_1)$ and $S'(d_2)$; one of the stacks which results is stored in an element and the other is returned as the catenate output.

Since we are using $S'(d)$ the first non-yellow element in $S(d)$ is accessible in constant time. It must be either the first or the second element in $S'(d)$. Let d' be the c-steque that is produced by the fix. If the popped element is not a pair then $S'(d')$ is obtained from $S'(d)$ as described in Section 3.3. $S'(d')$ and $S'(d)$ are the same except for at most four elements on top. Otherwise, if the popped element is a pair, a catenate is performed. $S'(d')$ is the stack formed by the catenate with an element which represents x (with its new prefix) pushed on top of it. If the first element in $S'(d)$ was a yellow secondary stack we push an element pointing to the same yellow stack on top of x in $S'(d')$.

5 Further Results

We have obtained some additional results that space limitations prevent us from discussing fully here. The method for catenable steques can be modified to avoid the use of non-catenable deques as prefixes and suffixes. We can instead make the prefixes and suffixes be deques with fixed maximum sizes. As mentioned in the introduction, we have also been able to extend our ideas to handle catenable double-ended queues in constant time per operation. We can also extend our data structures to support an additional heap order without affecting the constant time bound or the purely functional implementation. For related work see [2, 3, 16, 24].

In independent work, Okasaki has obtained a persistent implementation of catenable stacks with a constant time bound per operation [27]. His method is much simpler than ours, but it has two technical drawbacks: It is not purely functional but uses memoization, and the time bound is amortized rather than worst case. It is still open whether his approach can be extended to the double-ended problem.

Acknowledgements

We thank Adam Buchsbaum for extensive and fruitful discussions concerning our ideas and David Wagner for contributing the idea of the color invariant as an alternative to the explicit use of binary counting as a work allocation mechanism. We also thank Chris Okasaki for his comments on an earlier draft of this paper.

References

[1] J. Allen. *Anatomy of LISP* McGraw-Hill Computer Science Series. McGraw-Hill Book Co., New York, 1978.

[2] A. L. Buchsbaum, R. Sundar and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues, *Proceedings of the 33rd IEEE Symp. on Foundations of Computer Science*, 1992, 40–49.

[3] A. L. Buchsbaum and R. E. Tarjan. Confluently persistant deques via data structural bootstrapping, *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, 155–164.

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

[5] B. Chazelle. How to search in history, *Information and Control* **77** (1985), 77–99.

[6] B. Chazelle and L. J. Guibas. Fractional cascading: I. A. data structure technique. *Algorithmica*, **1**(2):133–62, 1986.

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

[8] R. Cole. Searching and storing similar lists, *Journal of Algorithms* **7** (1986), 202–220.

[9] D. P. Dobkin and J. I. Munro. Efficient uses of the past, *Journal of Algorithms* **6** (1985), 455–465.

[10] P. F. Dietz. Fully persistent arrays, Proceedings of the 1989 Workshop on Algorithms and Data Structures, Ottawa Canada, *Lecture Notes in Computer Science* 382, Springer-Verlag, 1989, 67–74.

[11] J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent, *Journal of Computer and Systems Science* **38** (1989), 86–124.

[12] J. Driscoll, D. Sleator, and R. Tarjan. Fully persistent lists with catenation. In *Proc. 2nd ACM-SIAM SYMP. on Discrete Algorithms*, pages 89–99, 1991. Submitted to J. ACM.

[13] M. Felleisen. A calculus for assignments in higher-order languages. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 180–190, 1988.

[14] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: a mathematical semantics for handling full functional jumps. *Proc. Conference on Lisp and Functional Programming*, 52–62, 1988.

[15] Patrick C. Fisher, Albert R. Meyer, and Arnold L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the ACM*, **19**(4):590–607, October 1972.

[16] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, **12**(4):197–100, April 1986.

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

[18] Robert Hood. *The efficient implementation of very-high-level programming language constructs.* PhD thesis, Department of Computer Science, Cornell University, 1982.

[19] R. Hood and R. Melville, Real-time queue operations in pure LISP, *Information Processing Letters* **13** (1981), 50–54.

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

[21] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 158–168, 1988.

[22] D. E. Knuth. *The Art of Computer Programming*, volume 1: *Fundamental Algorithms.* Addison-Wesley, Reading, MA, second edition, 1973.

[23] S. R. Kosaraju. Real-time simulation of concatenable double-ended queues by double-ended queues. In *Proc. 11th ACM Symp. on Theory of Computing*, pages 346–51, 1979.

[24] S. R. Kosaraju. An optimal RAM implementation of catenable min double-ended queues. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 195–203, 1994.

[25] Benton L. Leong and Joel I. Seiferas. New real-time simulations of multihead tape units. *Journal of the ACM*, **28**(1):166–180, January 1981.

[26] Chris Okasaki. Simple and efficient purely functional queues and deques. *J. Functional Progamming*, to appear.

[27] Chris Okasaki. Private Communication, February 1995.

[28] M. H. Overmars, Searching in the past, I. *Information and Computation*, in press.

[29] M. H, Overmars, "Searching in the past, II: General Transofrms," Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utricht, The Netherlands, 1981.

[30] T. Reps, T. Etitelbaum, and A. Demers, Incremental context-dependent analysis for language based editors, *ACM Transactions on Progamming Languages and Systems* **5** (1983), 449–477.

[31] N. Sarnak. *Persistent Data Structures.* Ph.D. thesis, Department of Computer Sciences, New York University, 1986.

[32] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, **29**(7):669–79, 1986.

[33] D. Sitaram, M. Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*: An International Journal, **3** (1990), 67–99.

[34] J. M. Spitzer, K. N. Levitt, and L. Robinson. An example of hierarchical design and proof. *Communications of the ACM*, **21**(12):1064–75, 1978.

[35] Hans-Jörg Stoss. K-band simulation von k-Kopf-Turing-maschinen. *Computing*, **6**(3):309–317, 1970.

[36] G. F. Swart, "Efficient Algorithms for Computing Geometric Intersections," Technical Report 85-01-02, Department of Computer Science, University of Washington, Seattle, WA, 1985.