Simple Confluently Persistent Catenable Lists
(Extended Abstract)

Haim Kaplan¹, Chris Okasaki²*, and Robert E. Tarjan³**

¹ AT&T labs, 180 Park Ave, Florham Park, NJ. hkl@research.att.com
² School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. cokasaki@cs.cmu.edu
³ Department of Computer Science, Princeton University, Princeton, NJ 08544 and InterTrust Technologies Corporation, Sunnyvale, CA 94086. ret@cs.princeton.edu.

Abstract. We consider the problem of maintaining persistent lists subject to concatenation and to insertions and deletions at both ends. Updates to a persistent data structure are nondestructive—each operation produces a new list incorporating the change while keeping intact the list or lists to which it applies. Although general techniques exist for making data structures persistent, these techniques fail for structures that are subject to operations, such as catenation, that combine two or more versions. In this paper we develop a simple implementation of persistent double-ended queues with catenation that supports all deque operations in constant amortized time.

1 Introduction

Over the last fifteen years, there has been considerable development of persistent data structures, those in which not only the current version, but also older ones, are available for access (partial persistence) or updating (full persistence). In particular, Driscoll, Sarnak, Sleator, and Tarjan [5] developed efficient general methods to make pointer-based data structures partially or fully persistent, and Dietz [3] developed an efficient general method to make array-based structures fully persistent.

These general methods support updates that apply to a single version of a structure at a time, but they do not accommodate operations that combine two different versions of a structure, such as set union or list catenation. Driscoll, Sleator, and Tarjan [4] coined the term confluently persistent for fully persistent structures that support such combining operations. An alternative way to obtain persistence is to use strictly functional programming (By strictly functional we mean that lazy evaluation, memoization, and other such techniques are not

** Research at Princeton University partially supported by NSF Grant No. CCR-9626862.
allowed). For list-based data structure design, strictly functional programming amounts to using only the LISP functions CAR, CONS, CDR. Strictly functional data structures are automatically persistent, and indeed confluently persistent.

A simple but important problem in data structure design that makes the issue of confluent persistence concrete is that of implementing persistent double-ended queues (deques) with catenation. A series of papers [4, 2] culminated in the work of Kaplan and Tarjan [8], who developed a confluently persistent implementation of deques with catenation that has a worst-case constant time and space bound for any deque operation, including catenation. The Kaplan-Tarjan data structure and its precursors obtain confluent persistence by being strictly functional.

If all one cares about is persistence, strictly functional programming is unnecessarily restrictive. In particular, Okasaki [12, 11, 13] observed that the use of lazy evaluation in combination with memoization can lead to efficient functional (but not strictly functional) data structures that are confluently persistent. In order to analyze such structures, Okasaki developed a novel kind of debit-based amortization. Using these techniques and weakening the time bound from worst-case to amortized, he was able to considerably simplify the Kaplan-Tarjan data structure, in particular to eliminate its complicated skeleton that encodes a tree extension of a redundant digital numbering system.

In this paper we explore the problem of further simplifying the Kaplan-Tarjan result. We obtain a confluently persistent implementation of deques with catenation that has a constant amortized time bound per operation. Our structure is substantially simpler than the original Kaplan-Tarjan structure, and even simpler than Okasaki's structure: whereas Okasaki requires efficient persistent deques without catenation as building blocks, our structure is entirely self-contained. Furthermore our analysis uses a standard credit-based approach. As compared to Okasaki's method, our method requires an extension of the concept of memoization: we allow any expression to be replaced by an equivalent expression.

The remainder of this extended abstract consists of five sections. In Section 2, we introduce terminology and concepts. In Section 3, we illustrate our approach by developing a persistent implementation of deques without catenation. In Section 4, we develop our solution for deques with catenation. We conclude in Section 5 with some remarks and open problems.

### 2 Preliminaries

The objects of our study are lists. As in [8] we allow the following operations on lists:

- **MAKELIST(x):** return a new list containing the single element $x$.
- **PUSH(x, L):** return a new list formed by adding element $x$ to the front of list $L$. 