Catenable Double-Ended Queues

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

Abstract

Catenable double-ended queues are double-ended queues (deques) that support catenation (i.e., append) efficiently without sacrificing the efficiency of other operations. We present a purely functional implementation of catenable deques for which every operation, including catenation, takes \(O(1)\) amortized time. Kaplan and Tarjan have independently developed a much more complicated implementation of catenable deques that achieves similar worst-case bounds. The two designs are superficially similar, but differ in the underlying mechanism used to achieve efficiency in a persistent setting (i.e., when used in a non-single-threaded fashion). Their implementation uses a technique called recursive slowdown, while ours relies on the simpler mechanism of lazy evaluation.

Besides lazy evaluation, our implementation also exemplifies the use of two additional language features: polymorphic recursion and views. Neither is indispensable, but both significantly simplify the presentation.

1 Introduction

Purely functional programming is gradually gaining recognition in the data structures community as an excellent medium for designing persistent (i.e., immutable) data structures. Several other general techniques for designing persistent data structures exist [5, 4], but unfortunately, these other techniques break down when the data structure in question supports operations that combine two or more structures. Examples of such offending operations include catenating (i.e., appending) two sequences, unioning two sets, or merging two priority queues. Of these, sequence catenation has received the most attention [6, 1, 11, 17].

Two implementations of purely functional catenable lists have recently been proposed. Kaplan and Tarjan [11] described an approach that supports catenation and all other usual list operations in \(O(1)\) worst-case time. Okasaki [17] presented a much simpler implementation based on lazy evaluation that achieved similar amortized bounds. In this paper, we extend these results to the double-ended case, yielding a purely functional implementation of catenable deques that supports catenation and all other usual deque operations in \(O(1)\) amortized time. Kaplan and Tarjan [12] have independently developed an implementation of catenable deques that achieves the same bounds in the worst case. However, our design is much simpler than theirs.

Continuing in the tradition of [16, 17, 18], our implementation reinforces the important role of lazy evaluation in purely functional data structures. Our implementation also makes extensive use of two additional language features: polymorphic recursion [14, 7] and views [22, 2, 20]. Although neither is indispensable, both significantly simplify the presentation. We hope that this example will motivate more language designers to include these features in their languages.

Section 2 briefly reviews related work. Section 3 describes the notation and properties we will assume for deques and for lazy evaluation. Section 4 presents our implementation of catenable deques and its analysis. Section 5 shows how to implement our data structure in a language without views or polymorphic recursion. Section 6 compares our data structure to the catenable deques of Kaplan and Tarjan [12]. Finally, Section 7 concludes with a few open problems.

2 Related Work


None of the above structures support catenation efficiently. Myers [15] described an implementation of AVL trees that supports all relevant deque operations, including catenation, in \(O(\log n)\) time. Hughes [10] represented lists as functions in such a way that catenation becomes simple function composition, running in \(O(1)\) time. Unfortunately, his structure can only be inspected in toto — it no longer supports individual head and tail operations efficiently. Driscoll, Sleator, and Tarjan [6] presented the first implementation of catenable lists to support all operations in sublogarithmic time: catenation in \(O(\log \log k)\) time, where \(k\) is the number of list operations (which may be much smaller than \(n\)), and all other operations in \(O(1)\) time. Their implementation is persistent, but not purely functional. Buchsbaum and Tarjan [1] gave a purely functional implementation of catenable deques supporting deletion of the first or last element in \(O(\log^2 k)\) time, and all other operations in \(O(1)\) time.

Kaplan and Tarjan [11] finally achieved an implementation of catenable lists that supports all operations, including catenation, in \(O(1)\) time. Their implementation is based on recursive slowdown
and achieves its bounds in the worst case. Okasaki [17] gave a much simpler implementation based on lazy evaluation that also supports all operations in $O(1)$ time, but only in the amortized sense. The catenable deques in this paper are descended from Kaplan and Tarjan’s implementation of catenable lists, but use lazy evaluation instead of recursive slowdown. In independent work, Kaplan and Tarjan [12] have also extended their implementation of catenable lists to the double-ended case. Modulo the difference between lazy evaluation and recursive slowdown, their approach is very similar to ours. For comparison purposes, we present their data structure in Section 6. To make the comparison clearer, we have adapted their data structure to use lazy evaluation instead of recursive slowdown. This greatly simplifies their design, but also degrades its bounds from worst-case to amortized.

3 Preliminaries

3.1 Non-Catenable Deques

Our implementation of catenable deques uses non-catenable deques internally. We will henceforth refer to catenable deques as c-deques and to non-catenable deques as p-deques (primitive deques).

Let $[\alpha]$ denote the type of p-deques containing elements of type $\alpha$. Let $[\alpha]_k$ denote the subtype of p-deques of length $k$ or greater. We assume that we are given an implementation of p-deques that supports each of the following operations in $O(1)$ time (see, for example, [8, 3, 16]):

\[
\begin{array}{ll}
[\ ] & : [\alpha] \quad \text{(the empty deque)} \\
\lt ; & : \alpha \times [\alpha] \rightarrow [\alpha] \quad \text{(left cons)} \\
\gt ; & : [\alpha] \times \alpha \rightarrow [\alpha] \quad \text{(right cons)} \\
\lhd, \rhd & : [\alpha]_1 \rightarrow [\alpha] \quad \text{(left and right head)} \\
\ltl, \rtl & : [\alpha]_1 \rightarrow [\alpha] \quad \text{(left and right tail)} \\
\cdot & : [\alpha] \rightarrow \text{int} \quad \text{(size)} \\
\end{array}
\]

\lt and \gt are infix operators, and are right-associative and left-associative, respectively. Even though we treat p-deques as an abstract data type, we allow $[\ ]$, \lt, and \gt to be used in pattern matching. These kinds of abstract patterns are called views [22, 2, 20]. A \[\ ]\ pattern matches the empty deque. The pattern $\mathrm{ph} \lt \mathrm{pt}$ means “given a non-empty deque $\mathrm{d}$, match pattern $\mathrm{ph}$ against $1 \lt \mathrm{d}$ and pattern $\mathrm{pt}$ against $\lt \lt \lt \mathrm{d}$. The pattern $\mathrm{pt} \gt \mathrm{ph}$ is interpreted similarly. In both expressions and patterns, we use the abbreviation $\langle x_1, \ldots, x_n \rangle$ for $\langle x_1, \ldots, x_n \rangle$ [\or equivalently, $\langle x_1 \downarrow \ldots \downarrow x_n \rangle$].

P-deques do not support catenation efficiently, but given the above primitives, it is simple to implement a catenation operation that runs in time proportional to the shorter of the two arguments.

\[
\begin{array}{ll}
\mathrm{\&} & : \alpha \times [\alpha] \rightarrow [\alpha] \\
x \times \mathrm{\&} \langle \rangle & = x \\
\mathrm{\&} \langle \rangle \times ys & = ys \\
x \lt (x \lt \langle \rangle \times ys) \gt y & = x \lt (x \lt ys) \gt y \\
\end{array}
\]

Note that the last line is ambiguous, depending on the relative precedences of \lt and \gt, but both readings yield the same result since $x \lt (d \gt y) = (x \lt d) \gt y$.

3.2 Lazy Evaluation

We assume that all computation is strict except where explicitly indicated otherwise. To delay the evaluation of an expression $e$ of type $\tau$, we write $\tau e$. This returns a suspension of type $\tau$. To force the evaluation of a suspension $s$ of type $\tau$, we write $!s$, which returns a value of type $\tau$. Suspensions are memoized, meaning that the first time a suspension is forced, the value is saved so that the next time the suspension is forced, the value can be looked up instead of recomputed.

In addition, we provide a view for suspensions that allows forcing during pattern matching. When matching a suspension against a pattern $\tau e$, we first force the suspension, and then match the resulting value against $\tau$.

This style of notation for lazy evaluation is explored more thoroughly in [19], along with many examples of its use.

4 Catenable Deques

In this section, we present our implementation of c-deques. C-deques support exactly the same operations as p-deques, but improve the running time of $\otimes$ to $O(1)$ amortized time. Except for size, the remaining operations continue to run in $O(1)$ time, although for $\lt l \lt l$ and $\lt r \lt r$, this becomes an amortized bound, whereas for some implementations of p-deques it may be a worst-case bound. If desired, we could also make size run in $O(1)$ time by adding a size field to the root of each c-deque.

Except at the level of types, we use the same notation for both c-deques and p-deques. Whether a given occurrence of, say, $<$ refers to c-deques or p-deques will always be uniquely determined by context.

4.1 Representation

Let $[\alpha]_{\otimes}$ denote the type of c-deques containing elements of type $\alpha$. A c-deque is either a simple p-deque, written $(d)$, or a five-tuple $(f, a, m, b, r)$, where $f$, $m$, and $r$ are p-deques and $a$ and $b$ are suspended c-deques of compound elements. $f$ and $r$ must contain at least three elements each and $m$ must contain at least two elements. A compound element is either a simple p-deque, written $(d)$, or a three-tuple $(f, c, r)$, where $f$, $c$, and $r$ are p-deques containing at least two elements each, and $c$ is a suspended c-deque of compound elements. These types are summarized by the following equations:

\[
\begin{align*}
[\alpha]_{\otimes} & = [\alpha] \times (\langle [\alpha]_3^+ \times (\text{CE} \alpha \otimes [\alpha]_2^+ \times (\text{CE} \alpha \otimes [\alpha]_2^+)) \\
\text{CE} \alpha & = [\alpha]_2^+ \times (\langle [\alpha]_2^+ \times (\text{CE} \alpha \otimes [\alpha]_2^+))
\end{align*}
\]

Whether $(d)$ is a c-deque or a compound element will always be uniquely determined by context.

Note that $[\alpha]_{\otimes}$ is defined in terms of $[\text{CE} \alpha]_{\otimes}$. Supporting this kind of non-uniform type in a useful way requires polymorphic recursion [14, 7]. (See Section 5 for how to cope without polymorphic recursion.)

The order of elements is from left to right at every level. Hence, the first and last elements of $(f, a, m, b, r)$ are the first element of $f$ and the last element of $r$, respectively.

4.2 Operations

Next, we define the operations on c-deques. The empty c-deque is defined in terms of the empty p-deque.

\[
[\ ] = (\langle \rangle)
\]

Adding an element on the left or right simply adds the element to the p-deque on that side.

\[
\begin{align*}
x \lt \langle d \rangle & = \langle x \lt d \rangle \\
x \lt \langle f, a, m, b, r \rangle & = \langle x \lt f, a, m, b, r \rangle \\
\langle d \rangle \gt x & = \langle d \gt x \rangle \\
\langle f, a, m, b, r \rangle \gt x & = \langle f, a, m, b, r \gt x \rangle
\end{align*}
\]
Similarly, asking for the leftmost or rightmost element returns the appropriate element of the p-deque on that side.

\[
\begin{align*}
&1\text{hd} (x \triangleleft d) = x \\
&1\text{hd} (x \triangleleft f, a, m, b, r) = x \\
&\text{rhd} (d \triangleright x) = x \\
&\text{rhd} (f, a, m, b, r \triangleright x) = x
\end{align*}
\]

The auxiliary functions \text{ltl}' and \text{rtl}' remove the elements returned by \text{1hd} and \text{rhd}.

\[
\begin{align*}
&\text{ltl}' (x \triangleleft d) = \langle d \rangle \\
&\text{ltl}' (x \triangleleft f, a, m, b, r) = \langle f, a, m, b, r \rangle \\
&\text{rtl}' (d \triangleright x) = \langle d \rangle \\
&\text{rtl}' (f, a, m, b, r \triangleright x) = \langle f, a, m, b, r \rangle
\end{align*}
\]

These auxiliary functions sometimes violate the size requirements of the data structure by leaving the \text{f or r} fields with only two elements instead of three. We will use these functions only when we intend to immediately replace the missing element.

We next turn to catenation. First, we consider catenating two simple p-deques. If one or both p-deques contains fewer than four elements, we combine them using p-deque catenation.

\[
\langle d_1 \rangle \times \langle d_2 \rangle = \langle d_1 \Vdash \langle d_2 \rangle \rangle, \text{if } |d_1| < 4 \lor |d_2| < 4
\]

Otherwise, we place both p-deques in a five-tuple with empty \text{a} and \text{b} fields. We remove one element from each p-deque to form the middle field.

\[
\langle d_1 \triangleright x \rangle \times (y \triangleleft d_2) = \langle d_1, \overline{x}, [y], \overline{x}, d_2 \rangle, \text{if } |d_1| \geq 3 \land |d_2| \geq 3
\]

When catenating a p-deque with a five-tuple, we simply invoke p-deque catenation if the p-deque is short enough, and otherwise move the existing \text{f or r} field into \text{a or b} and install \text{d} as the new \text{f or r} field.

\[
\langle d \rangle \times \langle f, a, m, b, r \rangle = \langle d \Vdash \langle f, a, m, b, r \rangle \rangle, \text{if } |d| < 4
\]

\[
\langle f, a, m, b, r \rangle \times \langle d \rangle = \langle f, a, m, b, r \Vdash \langle d \rangle \rangle, \text{otherwise}
\]

The most interesting case is catenating two five-tuples. The \text{left half} of a five-tuple comprises the \text{f and a} fields, and half of the \text{m} field. The \text{right half} of a five-tuple comprises the \text{r and b} fields, and the other half of the \text{m} field. To catenate two five-tuples, we first fold the right half of the left tuple into the left half, and the left half of the right tuple into the right half. Then, we glue the two halves together.

\[
\langle f_1, a_1, m_1, b_1, r_1 \triangleright x \rangle \times (y \triangleleft f_2, a_2, m_2, b_2, r_2) = \langle f_1, \overline{a_1} \triangleleft (m_1, b_1, r_1), [x, y], \overline{f_2, a_2, m_2} \triangleleft \overline{b_2, r_2} \rangle
\]

The definitions of \text{ltl} and \text{rtl} use several auxiliary views. The first pair of views, \text{≤} and \text{≥}, are just like \text{≤} and \text{≥} except that they delay their tails. For example, matching \text{ph} \triangleleft \text{pt} against a non-empty c-deque \text{xs} first matches \text{ph} against \text{1hd} \text{xs} and then matches \text{pt} against \text{ltl} \text{xs}. The second pair of views, \text{◁} and \text{▷}, are just like \text{≤} and \text{≥} except that they call \text{ltl}' and \text{rtl}' instead of \text{ltl} and \text{rtl}.

Finally, we are ready to define \text{ltl} and \text{rtl}. We begin with \text{ltl}. The simplest cases are when \text{ltl} \text{xs} can discard the leftmost element of \text{xs} without violating the size restrictions. This happens when \text{xs} is a simple p-deque or when \text{xs} is a five-tuple whose \text{f} field contains more than three elements.

\[
\begin{align*}
&\text{ltl} \langle x \triangleleft d \rangle = \langle d \rangle \\
&\text{ltl} \langle x \triangleleft f, a, m, b, r \rangle = \langle f, a, m, b, r \rangle, \text{if } |f| \geq 3
\end{align*}
\]

In the remaining cases, \text{xs} is a five-tuple whose \text{f} field contains exactly three elements so removing the leftmost element leaves only two elements. To refill the \text{f} field, we first try to remove a compound element from the \text{a} field. If it is a simple p-deque \text{d}, we add it to the \text{f} field.

\[
\text{ltl} \langle [x, y, z], \langle d \rangle \triangleleft a, m, b, r \rangle = \langle y \triangleleft z, d, a, m, b, r \rangle
\]

Note the use of polymorphic recursion here — by matching against the \text{a} view, \text{ltl} on a c-deque of elements implicitly invokes \text{ltl} on a c-deque of compound elements. However, note that the recursive call to \text{ltl} is suspended by the \text{g} view. This use of lazy evaluation is critical if the data structure is to be efficient in a persistent setting [18].

We continue with the remaining clauses of \text{ltl}. If the first compound element from the \text{a} field is a three-tuple \langle \text{f}', \text{c}', \text{r}' \rangle, then we add \text{f}' to the \text{f} field and replace the three-tuple in \text{a} with \langle \text{r}' \rangle. Finally, we catenate \text{c}' and \text{a} to obtain the new \text{a} field.

\[
\text{ltl} \langle [x, y, z], \langle f', c', r \rangle \triangleleft a, m, b, r \rangle = \langle y \triangleleft z, a, m, b, r \rangle
\]

If the \text{a} and \text{b} fields are both empty, then we add the remaining elements of the \text{f} field to \text{m} and catenate the result with \text{r}.

\[
\text{ltl} \langle [x, y, z], \overline{m}, \overline{m}, \overline{r} \rangle = \langle y \triangleleft z \triangleleft m \rangle \times \langle r \rangle
\]

This completes the definition of \text{ltl}. \text{rtl} is defined symmetrically. The complete implementation of c-deques is summarized in Figure 1.

### 4.3 Analysis

We first argue informally that every operation runs in O(1) amortized time. Then we prove this formally using a debit argument in the style of [17, 18, 19].

First, note that only \text{ltl} and \text{rtl} call themselves recursively.

The remaining operations clearly run in O(1) time since none of them loop. Now consider \text{ltl} (the argument for \text{rtl} is similar).

The first two cases terminate immediately. Several of the remaining cases recursively call \text{ltl} on \text{a} or \text{b}. But note that at the end of each of these cases, \text{f} contains at least four elements: \text{y}, \text{z}, and two or more elements from the p-deque used to refill \text{f}. Therefore, the next call to \text{ltl} will terminate immediately in the second clause. This means that at most every other call to \text{ltl} can call itself recursively.

Extending this argument a few steps further, we note that at most every fourth call can make two recursive calls, at most every eighth call can make three recursive calls, and so on. Altogether then, the amortized cost of any one call at the top level is

\[
O(1 + 1 + 1 + 1 + \cdots) = O(1).
\]

As an aside, this argument explains the size restrictions we place on the various p-deques in five-tuples and compound elements. When we refill an \text{f} or \text{r} field that has dropped below the minimum size, we wish to raise it not just to the minimum size, but above it so that the next operation that removes an element from that field will terminate immediately. Therefore, \text{m} and the various p-deques in compound elements that are used to refill \text{f} and \text{r} must contain at
\[
[a]_\ast = [a] \mid ([a]_\ast \times \overline{CE} \alpha_\ast \times [a]_\ast \times \overline{CE} \alpha_\ast \times [a]_\ast_\ast)
\]
\[
CE \alpha = [a]_\ast_\ast \mid ([a]_\ast_\ast \times \overline{CE} \alpha_\ast \times [a]_\ast_\ast)
\]

\[
[\text{Figure 1: Catenable deques.}]
\]
least two elements each. The \( f \) and \( r \) fields in five-tuples must contain at least three elements, because during catenation one element is transferred to the \( m \) field and the remainder of the p-deque goes in a compound element (and thus must contain at least two elements).

Although the above informal argument provides a useful intuition, it fails to address two important concerns. First, what happens if there are other operations, such as \( lt \) and \( rt \), interleaved with the calls to \( lt l \)? Since \( lt l \) and \( rt l \) can both recurse on either the \( a \) or \( b \) field of a c-deque, we cannot blithely assume that they will not interfere with each other. Second, what happens if c-deques are used persistently? For example, if \( lt l \) \( xz \) recurses to depth \( k \), how can we be sure that repeating this call \( n \) times will not take \( O(nk) \) time?

We could satisfy the first concern using any of several formal techniques, such as the standard techniques of amortized analysis using credits or potential functions [21] or the non-standard debit techniques of Okasaki [17, 18, 19] for analyzing amortized data structures involving lazy evaluation. The basic approach under any of these methods is to establish an invariant and show that any individual call to \( lt l \) or \( rt l \) preserves the invariant, so any sequence of interleaved calls also preserves the invariant. However, of these various proof techniques, only debit arguments address the question of persistence. The key ingredient in this technique is the use of lazy evaluation to delay expensive computations. This allows the results of these computations to be shared via memoization among multiple “threads” of a non-single threaded computation.\(^1\) See [17, 18, 19] for a fuller discussion of the role of lazy evaluation in persistent, amortized data structures.

In a debit argument, every suspension is assigned a certain number of debits, which account for the cost of eventually executing the suspension. Every debit must be discharged before its corresponding suspension may be forced. There are three kinds of suspensions in our data structure: the \( a \) and \( b \) fields of five-tuples, and the \( c \) field of three-tuples. We limit the number of debits on each \( c \) field to four, and limit the number of debits on each \( a \) or \( b \) field according to the sizes of \( f \) and \( r \).

- If \(|f| > 3\) and \(|r| > 3\), then \( a \) and \( b \) are allowed five debits each.
- If \(|f| > 3\) and \(|r| = 3\), then \( a \) is allowed four debits and \( b \) is allowed one debit.
- If \(|f| = 3\) and \(|r| > 3\), then \( a \) is allowed one debit and \( b \) is allowed four debits.
- If \(|f| = 3\) and \(|r| = 3\), then \( a \) and \( b \) are allowed zero debits each.

The amortized cost of each operation is \( O(1 + \#\text{debits discharged}) \). We show that \( \Sigma \) discharges at most four debits and that \( lt l \) and \( rt l \) discharge at most five debits each.

\textbf{Proof:} (\textcircled{\textbullet}) The interesting case is catenating two five-tuples \( x_{s_1} = (f_1, a_1, m_1, b_1, r_1) \) and \( x_{s_2} = (f_2, a_2, m_2, b_2, r_2) \). We create and immediately discharge two debits to pay for the suspended \( \triangleleft \) and \( \triangleright \) onto \( a_1 \) and \( b_2 \). In addition, we discharge at most one debit from either \( a_1 \) or \( a_2 \), and at most one debit from either \( b_1 \) or \( b_2 \). Suppose \(|r_2| > 3\). Then \( a_2 \) might have five debits, one of which must be discharged as \( a_2 \) becomes the \( c \) field of a new three-tuple. Otherwise, if \(|r_2| = 3\) and \(|r_1| > 3\), then the allowance of \( a_1 \) might decrease by one, requiring the discharge of a single debit. A similar argument holds for \( b_1 \) and \( b_2 \). Altogether, we discharge no more than four debits.

\( (lt l \text{ and } rt l) \) Since \( lt l \) and \( rt l \) are symmetric, we present the argument only for \( lt l \). Consider a call to \( lt l \) that recurses to depth \( k \) and note that every call except the outermost is enclosed in a suspension. Five debits must be discharged before each of these calls, but only the debits for the outermost call must be discharged immediately. For each of the recursive calls, those five debits are charged to the enclosing suspension. These debits will then be discharged sometime before the enclosing suspension is forced and the recursive call in question is executed. We call this transfer of debits from one set of suspensions to another \textit{debit passing}. Now, there is one case for every clause of \( lt l \). We describe only the cases for clauses 2, 3, and 4. The other cases are similar.

\begin{itemize}
\item \( lt l \langle x, a, f, a, m, b, r \rangle = \langle f, a, m, b, r \rangle \), if \(|f| \geq 3\)
  This is a terminating call. If the length of the \( f \) field drops from four to three, then the debit allowance of \( a \) drops by four and the debit allowance of \( b \) drops by one. We pass these five debits to the enclosing suspension, or discharge them if this is the outermost call.
\item \( lt l \langle x, y, z, \langle d \triangleright a, m, b, r \rangle \rangle = \langle y < z \triangleleft d, a, m, b, r \rangle \)
  This is not a terminating call. Since we force the \( a \) field, we must pass or discharge any debits currently on that field. If \(|r| > 3\) then there is currently at most one debit on the \( a \) field. We pass this debit to the enclosing suspension or discharge it if this is the outermost call. In addition, the new suspension for \( a \) (the one create by the \( \triangleleft \) view) receives at most five debits from its recursive call to \( lt l \). However, the new allowance for \( a \) is five, so we do not pass on any of these debits. If \(|r| = 3\) then there are currently zero debits on the \( a \) field. The new suspension for \( a \) receives at most five debits from the recursive call, but the new allowance for \( a \) is four, so we pass on one of these debits. Either way, we pass a single debit (or discharge it if this is the outermost call).
\item \( lt l \langle x, y, z, \langle f', e', r' \rangle \triangleleft a, m, b, r \rangle = \langle y < z < f', \triangleleft e' \triangleright (r') \triangleleft a \rangle, m, b, r \rangle \)
  This is a terminating call. Since we force the \( a \) field, we must pass on any debits that are currently on that field. There is one such debit if \(|r| > 3\) and none if \(|r| = 3\). The new \( a \) field receives at most four debits from \( e' \), at most four debits from the call to \( \triangleleft \), and one newly created debit that accounts for the call to \( \triangleleft \). The new allowance is five if \(|r| > 3\) so we pass on the excess four credits, making five altogether. The new allowance is four if \(|r| = 3\) so we pass on the excess five debits. In either case, we pass or discharge a total of five debits.
\end{itemize}

\section{5 Restricting the Language of Implementation}

The code presented in Section 4 takes advantage of both views and polymorphic recursion. However, few current languages support these features, so we briefly sketch how the implementation changes without them.

\subsection{5.1 Without Views}

Views [22, 2, 20] are a language mechanism allowing pattern matching on abstract datatypes. As with pattern matching in general, views are a syntactic convenience that can be replaced by explicit calls to case predicates (such as \texttt{null}) and access functions (such as \texttt{lhd} and \texttt{ltl}).

We use views in two ways. First, we use patterns such as \( x < d \) and \( d > x \) on p-deques to both recognize and decompose non-empty p-deques. The use of these patterns reveals nothing about the representation of p-deques, which is held abstract. Second, we use patterns such as \( x < a \) and \( x < a \) on c-deques to decompose these deques in non-standard ways. Note that we also use ordinary

\footnote{This terminology can be somewhat confusing. Here the term \textit{threads} refers not to concurrent threads of execution, but rather to multiple paths through the graph of data dependencies. Reusing a given deque induces a branch in the graph of data dependencies and hence creates a new thread.}
\begin{align*}
ltl \langle x < d \rangle & = \langle d \rangle \\
ltl \langle x < f, a, m, b, r \rangle & = \langle f, a, m, b, r \rangle \text{, if } |f| \geq 3 \\
ltl \langle x, y, z, \underline{[d]} \triangleleft a, m, b, r \rangle & = \langle y \triangleleft d, a, m, b, r \rangle \\
ltl \langle x, y, z, \langle f', c', r' \rangle \triangleleft a, m, b, r \rangle & = \langle y \triangleleft f', \underline{[c']} \times ((r') \triangleleft a), m, b, r \rangle \\
ltl \langle x, y, z, \underline{[m]} \triangleleft \underline{[d]}, b, r \rangle & = \langle y \triangleleft m, \underline{[d]}, b, r \rangle \\
ltl \langle x, y, z, \underline{[m]}, \underline{[f', c', r']}, b, r \rangle & = \langle y \triangleleft m, \langle f' \rangle \triangleleft c', r', b, r \rangle \\
ltl \langle x, y, z, \underline{[m]}, \underline{[f]}, a, m, b, r \rangle & = \langle y \triangleleft m \times (r) \rangle
\end{align*}

Figure 2: The ltl function, written with and without views.

\begin{align*}
[a]_{\infty} & = [a] \mid ([a]_{3+} \times [\text{CE } a]_{\infty} \times [a]_{2+} \times [\text{CE } a]_{\infty} \times [a]_{3+}) \\
\text{CE } a & = [a]_{2+} \mid ([a]_{2+} \times [\text{CE } a]_{\infty} \times [a]_{2+}) \\
\text{CE } [a]_{\infty} & = [\text{CE } a] \mid ([\text{CE } a]_{3+} \times [a]_{\infty} \times [\text{CE } a]_{2+} \times [a]_{\infty} \times [\text{CE } a]_{3+}) \\
\text{CE } a & = a \mid [\text{CE } a]_{2+} \mid ([\text{CE } a]_{2+} \times [a]_{\infty} \times [\text{CE } a]_{2+})
\end{align*}

Figure 3: The type of c-deques, with and without polymorphic recursion.
pattern matching on c-deques to distinguish between, for instance, \( \langle d \rangle \) and \( \langle f, a, m, b, r \rangle \). Views are not necessary for this last class of patterns because they match the concrete representation of c-deques, which is visible within the implementation.

To remove the dependency on views, we replace each view pattern with appropriate calls to \texttt{null}, \texttt{1hd (rh2)}, and \texttt{1tl (rlt1)}. For example, Figure 2 contrasts versions of the \texttt{1tl} function written with and without views. The version with views is clearly more concise, but more importantly, it is also easier to understand, at least for a reader comfortable with views. Even for a reader not comfortable with views, the version with views is probably easier to read for the gist of the implementation, although for such a reader the second version may be preferable for understanding the details.

5.2 Without Polymorphic Recursion

Polymorphic recursion [14, 7] allows one to write recursive functions on non-uniform recursive datatypes. Without polymorphic recursion, recursive functions can be written only for uniform recursive types. The type of c-deques, as presented in Section 4, is non-uniform because \( \alpha [\alpha_0] \) is defined in terms of \([ CE \alpha [\alpha_0] \) rather than \( \alpha [\alpha_0] \). If polymorphic recursion is not available, then we must modify this type definition to be uniform.

Consider the elements in the various p-deques in the representation of a c-deque. These elements have type \( \alpha \) at the first (top) level, type \( CE \alpha \) at the second level, type \( CE (CE \alpha) \) at the third level, and so on. To make \( \alpha [\alpha_0] \) uniform, we must collapse all of these types into a single type. First, we allow a simple element to be used anywhere a compound element can be used by extending the definition of \( CE \alpha \) with a third summand of type \( \alpha \) (i.e., \( CE \alpha = \alpha | \ldots \)). Next, we allow a compound element to be used anywhere a simple element can be used by replacing each p-deque of type \( \alpha [\alpha_0] \) with a p-deque of type \( CE \alpha \). With these changes, we can finally replace each \( CE \alpha [\alpha_0] \) with \( \alpha [\alpha_0] \). Figure 3 shows the final type definitions.

The rest of the implementation is mostly unaffected by these changes. We need only provide wrapper functions for \( a (b) \) and \texttt{1hd (rh2)} to inject and project elements of type \( \alpha \) to and from type \( CE \alpha \). These wrapper functions are exported to the user, while \texttt{Bd}, \texttt{ltl}, and \texttt{rtl} continue to call the original versions.

Although these changes are all relatively minor, we feel that the original implementation using polymorphic recursion is far superior. Not only does the non-uniform type provide much better documentation of the invariants of the data structure (i.e., that the first level p-deques contain elements of type \( \alpha \), the second level p-deques contain elements of type \( CE \alpha \), and so on), it also allows the type system to catch many more accidental violations of these invariants.

6 An Alternative Implementation of Catenable Deques

The catenable deques of Kaplan and Tarjan [12] share a superficially similar structure with ours, but the two implementations are difficult to compare because of differences in their underlying mechanisms. To facilitate comparison, we adapt their implementation to our framework. This greatly simplifies many details of their structure, but also degrades its bounds from worst-case to amortized. For the opposite view, see [12], where Kaplan and Tarjan have adapted our implementation to their framework.

In Kaplan and Tarjan’s design, a \textit{left pair} is a pair \( \langle f, a \rangle \), where \( f \) is a p-deque containing at least two elements and \( a \) is a c-deque of right pairs. A \textit{right pair} is a pair \( \langle b, r \rangle \), where \( r \) is a p-deque containing at least two elements and \( b \) is a c-deque of left pairs. A c-deque is either a simple p-deque, written \( \langle d \rangle \), or a four-tuple \( \langle f, a, b, r \rangle \) containing both a left pair and a right pair. The type definitions and operations for this data structure are summarized in Figure 4.

Kaplan and Tarjan’s design replaces our five-tuples with four-tuples, and our three-tuples with left pairs and right pairs. In addition, they reduce the minimum size of the \( f \) and \( r \) fields in a c-deque from three to two. On the other hand, our structure is more regular, having no need to distinguish between left pairs and right pairs.

The operations on both structures are mostly similar. The largest difference occurs in the \( 1tl \) and \( rtl \) functions. The question is: when the \( f \) field becomes too small, how do we refill it from the \( a \) field? In Kaplan and Tarjan’s design, this is accomplished by the following two rules:

\[
\begin{align*}
\langle x, y, \langle \langle d, a, b, r \rangle \rangle \rangle \Rightarrow \langle y \langle r', a, b, r \rangle \rangle \\
\langle x, y, \langle \langle f', a \rangle, b, r \rangle \rangle \Rightarrow \langle y \langle f', \langle a \rangle, b, r \rangle \rangle
\end{align*}
\]

The second rule, in particular, is rather involved, containing a total of four views that force two suspensions and remove the first elements of two c-deques. In contrast, these are the equivalent rules for our implementation:

\[
\begin{align*}
\langle x, y, z, \langle d \rangle \rangle \Rightarrow \langle y \langle z \rangle \rangle \\
\langle x, y, \langle f', r' \rangle \rangle \Rightarrow \langle y \langle f', \langle r' \rangle \rangle \rangle
\end{align*}
\]

Here the second rule contains only two views, one to force the suspension and one to remove the first element of the inner c-deque.

All in all, as long as our design and Kaplan and Tarjan’s design are both implemented using lazy evaluation or both using recursive slowdown, there is little reason to prefer one over the other on the grounds of simplicity or aesthetics. Which is to be preferred in practice can only be decided by a suitable empirical study. Unfortunately, we do not yet have enough experience with catenable deques—especially persistent ones—to determine an appropriate instruction mix for such a study.

7 Open Problems

The catenable deques of Kaplan and Tarjan [12] are asymptotically optimal. However, they are rather complex, so one might hope that a simpler structure with equivalent asymptotic bounds would run faster in practice. The catenable deques described in this paper are simpler, but achieve only amortized rather than worst-case bounds. It is still an open problem whether the catenable lists of Okasaki [17] can be extended to the double-ended case. Such a data structure would likely be simpler yet, but would also achieve amortized rather than worst-case bounds. Is a simpler worst-case approach possible?

A second area of further research involves extending catenable deques with additional operations. For example, it is relatively easy to extend both our data structure and Kaplan and Tarjan’s to support both \textit{reverse} and \textit{findMin} in \( O(1) \) time [12]. Can either design be extended with efficient primitives for random access, such as looking up or updating the \( i \)th element, or inserting or deleting the \( i \)th element? Kaplan and Tarjan [13] have described a related data structure supporting these operations, but catenation in that design requires \( O(\log \log (\min(n_1, n_2)) \) time. Is it possible to achieve constant-time catenation for such a data structure?

References

Figure 4: Catenable deques based on a design by Kaplan and Tarjan [12], adapted to use lazy evaluation.


