Chapter 22

Ephemeral and Single-Threaded Sequences

This chapter covers implementations of sequences that support constant work updates.

1 Persistent and Emphemeral Implementations

Persistent Data Structures. The implementations and the cost models that we have discussed so far are "non-destructive" in the sense that if we use a sequence, by for example, passing the sequence to an operation such as map, $update$, or $inject$ the sequence remains the same after the operation completes. Such implementations are sometimes called ***pure*** or ***persistent***.

Persistence is generally a desirable property. Some algorithms benefit from persistence and it is safe for parallelism. But persistence does come with a cost, because we are not allowed to update data destructively in place. For example, in array sequences , the $update\ a\ (i, v)$ and $inject\ a\ b$ operations require $\Omega(|a|)$ work because they have to copy the sequence a. In tree sequences , $update\ a\ b$ and $inject\ a\ b$ require $\Theta(\lg |a|)$ and $\Theta(\lg |a| + \lg |b|)$ work, but in some algorithms this is still high.

Ephemeral Data Structures. Persistence is not always necessary. For example, an algorithm may use a data structure in a "linear" fashion, where it uses or more precisely "consumes" an instance of the data structure no more than once. Linearity is relatively common, especially in sequential algorithms. For example, an algorithm may, at each step, consume one instance of a data structure and create a new instance, which is then consumed in another step. In such use cases, we may employ destructive updates in the implementation of the data structure: because an instance is never consumed more than once, it is safe to destruct or reuse it when making a new instance. We refer to an implementation of a data structure that destroys existing instances as ***ephemeral***.

Disadvantages of Ephemeral Implementations. Even though they can be efficient, ephemeral implementations have one important disadvantage: they are generally not safe for parallelism. As an example, consider the following three sequences:

$$\begin{array}{lcl} a & = & \langle\, 0, 1, \ldots, n-1 \,\rangle \\ b & = & \langle\, (0,0), (1,2), (2,4), \ldots, (n-1, 2n-2) \,\rangle \\ c & = & \langle\, (0,1), (1,3), (2,5), \ldots, (n-1, 2n-1) \,\rangle \end{array}$$

Using ephemeral sequences, the result of the following piece of code, which injects the sequence b and c into another sequence a is non-deterministic:

$$\mathit{inject}\ a\ b\ \|\ \mathit{inject}\ a\ c.$$

This piece of code has as many as 2^n distinct outcomes: the element at position i is either the i^{th} even number of i^{th} odd number.

Ephemeral implementations can therefore make reasoning about the correctness parallel algorithms challenging, because we have to consider an exponential number of possibilities. An earlier chapter covers this topic in more detail. This does not mean, however, that emphemeral data structures should be avoided at all cost. They are usually acceptable in sequential algorithms. Even in parallel algorithms, it is sometimes possible to use them in a structured fashion and establish that they don't harm correctness.

2 Ephemeral Sequences

Constant Work Updates. We can create an ephemeral version of array sequences by changing the update, inject, and $\mathit{ninject}$ primitives to update the input array destructively. For an update sequence of length m, the resulting implementation has the following improved bounds:

- $O(1)$ work and span for update,
- $O(m)$ work and $O(\lg d)$ span for inject, where d is the degree of the of the update sequence,
- $O(m)$ work and $O(1)$ span for $\mathit{ninject}$.

Note that this implementation is significantly more work efficient than the persistent one, and thus can make a real difference in complexity if the algorithm performs many updates.

3 Single-Threaded Sequences

Single-threaded sequence data structure offers a specific interface that can in some cases, combine the best of ephemeral and persistence sequences. The data structure is persistent—its functions have no externally visible effects—but its implementation internally uses benign effects. These benign effects make the cost specification more subtle.

Example 22.1. Recall that the function $update\ a\ (i, v)$ updates sequence a at location i with value v returning the new sequence, and that $inject\ a\ b$ updates sequence a using a sequence b of index-value pairs (each value is written to the corresponding index. Using arrays costs, $update$ requires $\Theta(|a|)$ work, and $inject$ requires $\Theta(|a| + |b|)$ work.

We can implement $inject$ using $update$ as follows.

$$inject\ a\ b = \ iterate\ update\ a\ (reverse\ b)$$

This code iterates over a making each of the updates specified in b. The problem, beyond being completely sequential, is that each update does $\Theta(|a|)$ work so the total work is $O(|a| \cdot |b|)$ instead of $O(|a| + |b|)$. The problem is that it is a waste to copy the sequence for every update.

Data Type 22.1 (Single Threaded Sequences)**.** For any element type α, the α-***single threaded sequence*** (stseq) data type is the type $\mathbb{T}_\alpha$ consisting of the set of all α stseq's, and the following functions.

$$\begin{array}{lcl} fromSeq & : & \mathbb{S}_\alpha \ \to \ \mathbb{T}_\alpha \\ toSeq & : & \mathbb{T}_\alpha \ \to \ \mathbb{S}_\alpha \\ nth & : & \mathbb{T}_\alpha \ \to \ \mathbb{N} \ \to \ \alpha \\ update & : & \mathbb{T}_\alpha \ \to \ (\mathbb{N} \times \alpha) \ \to \mathbb{T}_\alpha \\ inject & : & \mathbb{T}_\alpha \ \to \ \mathbb{S}_{\mathbb{N}\times\alpha} \ \to \mathbb{T}_\alpha \end{array}$$

where $\mathbb{S}_\alpha$ are standard sequences, and nth, $update$, and $inject$ behave as they do for standard sequences.

An $stseq$ is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (nth), update a position of the sequence ($update$), or update multiple positions in the sequence ($inject$). To use other functions from the sequence library, one needs to covert an $stseq$ back to a sequence (using $toSeq$).

To define the cost specification we need to distinguish between the latest version of an $stseq$, and earlier versions. Whenever we update a sequence, we create a new version, and the old version is still around due to the persistence. The cost specification then gives different costs for updating the latest version and old versions. Here we only define the cost for updating and accessing the latest version, because this is the only way we will be using an $stseq$.

Cost Specification 22.2 (Single Threaded Array Sequence)**.**

	Work	Span
$fromSeq\ a$	$O\left(\lvert a\rvert\right)$	$O\left(1\right)$
$toSeq\ a$	$O\left(\lvert a\rvert\right)$	$O\left(1\right)$
$nth\ a\ i$	$O\left(1\right)$	$O\left(1\right)$
$update\ a\ (i, v)$	$O\left(1\right)$	$O\left(1\right)$
$inject\ a\ b$	$O(\lvert b\rvert)$	$O\left(\lg(\mathsf{degree}(d))\right)$

In the cost specification the work for both nth and $update$ is $O\left(1\right)$, which is asymptotically as good as we can get. Again, however, this is only when a is the latest version of

a sequence (i.e. no one else has updated it). The work for $inject$ is proportional to the number of updates. It can be viewed as a parallel version of $update$.

Example 22.2 (Inject with Update Revisited)**.** If we return to our previous example:

$$inject\ a\ b = \ iterate\ update\ a\ (reverse\ b)$$

Using single threaded array sequences, the work is now just $\Theta(|b|)$ since each of the $|b|$ updates take constant work, and every update is on the last version. The span, however, is also $\Theta(|b|)$ since iterate is fully sequential. Therefore the built in $inject$ is significantly better for parallelism, but they both take the same amount of work.

Applications in this Book. In this book we can use $stseq$'s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

3.1 Implementation

You might be curious about how single threaded sequences can be implemented so than they act purely functional but match the given cost specification. Here we will just briefly outline one approach.

The idea is to keep with each sequence a version number, and for each position in the sequence a version list. The version number is incremented each time the sequence is updated either with $update$ or $inject$. A version list is a linked list of all the updates to the corresponding position, each with the version number of when the update was made. The most recent version is kept at the head of the list, and the rest are kept in decreasing order. A mutable (impure) array is used to keep a pointer to the head of each list, and the version number is also mutable.

We now consider how to do lookups and updates on the most recent version. For a lookup we can just look into the given location and take the first value from the head of the linked list. This takes constant work. To do an update requires updating the version number, grabbing the appropriate list, adding the value and version number to the front of the list, and then writing back the new head of the list using mutation (if done right, this is a benign effect). The update also takes constant work. Looking up or updating an old version is more expensive. A lookup requires grabbing the list from the given position, and then looking through the list for the correct version. An update requires copying the whole array.

There are two tricky aspects. The first is ensuring that the lists do not get too large. This can be implemented by copying the whole array once the number of updates equals the length of the sequence. The second is to ensure safety in a parallel setting, which can be achieved by using atomic read-modify-write operations. We will briefly cover such operations in the final part of this course.