J. Functional Programming **7** (5): 487–513, September 1997. Printed in the United Kingdom

A new method for functional arrays

MELISSA E. O'NEILL and F. WARREN BURTON
School of Computer Science, Simon Fraser University, Canada

Abstract

Arrays are probably the most widely used data structure in imperative programming languages, yet functional languages typically only support arrays in a limited manner, or prohibit them entirely. This is not too surprising, since most other mutable data structures, such as trees, have elegant immutable analogues in the functional world, whereas arrays do not. Previous attempts at addressing the problem have suffered from one of three weaknesses, either that they don't support arrays as a persistent data structure (unlike the functional analogues of other imperative data structures), or that the range of operations is too restrictive to support some common array algorithms efficiently, or that they have performance problems. Our technique provides arrays as a true functional analogue of imperative arrays with the properties that functional programmers have come to expect from their data structures. To efficiently support array algorithms from the imperative world, we provide $O(1)$ operations for single-threaded array use. Fully persistent array use can also be provided at $O(1)$ amortized cost, provided that the algorithm satisfies a simple requirement as to uniformity of access. For those algorithms which do not access the array uniformly or single-threadedly, array reads or updates take at most $O(\log n)$ amortized time, where n is the size of the array. Experimental results indicate that the overheads of our technique are acceptable in practice for many applications.

Capsule Review

Arrays typically receive second-class treatment in functional programming languages because of the difficulty of combining efficient updates with functional semantics. The authors present an implementation of functional arrays that support reads and updates in $O(1)$ amortized time for most common access patterns, and in $O(\log n)$ amortized time for worst-case access patterns. Their solution is a skillful combination of off-the-shelf components from the algorithms community, such as splay trees and an efficient solution to the List Order Problem. Although their data structure has functional semantics, its implementation is decidedly imperative, so it must either be provided by the compiler writer as a primitive or built on top of imperative arrays in a language like Standard ML.

1 Introduction

Data structures in the functional world and the imperative world are different. The imperative style of programming involves assignment and the mutation of existing data structures, whereas the functional model does not. To describe the ramifications of these differences, we will use the terminologies of Schmidt (1985) and Driscoll *et al.* (1989). Updates to a mutable data structure destroy the old

version of the data structure in creating the new – thus in Driscoll's terminology mutable data structures are *ephemeral*. In contrast, updates to an immutable data structure need to leave the original version of the data structure accessible and unchanged, and so must create a separate new version where the relevant change has been made – thus, in Driscoll's terminology, immutable data structures are *persistent*. Driscoll also introduced a concept of *partial persistence*, where the original version remains readable after an update, but cannot itself be updated. In Schmidt's terminology, algorithms that only ever refer to the most recent version of a data structure are *single-threaded*, and those that do not are *non-single-threaded*. From these definitions, single-threaded algorithms only require data structures that are ephemeral, and ephemeral data structures can only support single-threaded algorithms.

Functional languages typically spurn explicit mutation of data structures, which means that data structures in functional languages tend to be persistent. This typically means functional programs do more copying than their imperative counterparts, but for most data structures, the additional cost is small enough to be outweighed by the elegance and safety of functional solutions. However, while mutable trees, lists and other linked data structures have pleasing functional analogs, arrays do not. The cost of having to copy a whole array, of arbitrary size, at every update seems grotesque and unacceptable.

In a functional context, then, while we could construct a vector data structure that allowed rapid indexing, as imperative arrays do, we cannot so easily provide a functional analog of the array element update that is the cornerstone of imperative array use.

1.1 Ignoring and sidestepping the problem

Since providing efficient functional arrays seems awkward, we shouldn't be too surprised that usually they aren't provided. Approaches vary from providing no arrays at all, to providing them but restricting the kinds of accesses that can be done efficiently, to offering no functional array support but providing imperative arrays instead.

Miranda (Turner, 1986) takes the first approach, offering no support for arrays, and the Standard ML definition omits them too (although most Standard ML implementations take the third approach and provide imperative arrays as an impure feature).

Haskell (Hudak *et al.*, 1992) opts for the second approach. It does provide functional arrays, but the functions provided seem designed to process groups of updates at once. In current implementations these mass update operations have $O(n+u)$ cost, where n is the size of the array and u is the number of updates performed together. While some algorithms can use this technique efficiently, by collecting groups of $O(n)$ updates and performing them en masse, others cannot. Unless *update analysis* (Bloss, 1989) is employed by the Haskell compiler, this technique cannot even offer good performance for common single-threaded algorithms from the imperative world.

Concurrent Clean (Huitema and Plasmeijer, 1992) opts for the third approach, but unlike Standard ML implementations, manages to do so quite elegantly, providing safe access to ephemeral imperative structures, including arrays, by the use of *unique types* (Achten *et al.*, 1993), which are similar to the *linear types* proposed by Wadler (1990b). Unique types allow access to mutable structures by enforcing a restriction that there can only be one copy of any pointer to such structures. The mechanism allows array algorithms from the imperative world to be ported to Concurrent Clean (although observing the 'single reference' property may make the ports rather more awkward than it would first appear).

Corresponding work has also been done on providing Haskell with safe support for imperative features. Here the work has not so much been on incorporating linear types into the language, but instead on programming using monads (Wadler, 1990a; Wadler, 1992). In Haskell, one could provide imperative arrays by using an array monad. Again this would allow imperative array algorithms that have no obvious functional counterpart to be ported to Haskell.

The problem with this third approach is that it does not address the issue of providing a functional analog of imperative arrays. If either unique types or monads are used to implement a comprehensive array package that can support all imperative array algorithms, we also end up with a package that also allows back into the language the very bugs that functional programming style is supposed to eliminate. Also, by providing arrays as an ephemeral and not a persistent data structure, arrays become special and unlike other functional data structures, which can pose problems because ephemeral data structures cannot be used in non-single-threaded algorithms.

For these reasons, others have looked at the issue of providing a true functional analog of imperative arrays that allows them to be used as persistent structures.

1.2 Previous attempts at functional arrays

Perhaps one of the most commonly used techniques to provide functional arrays is to use a balanced binary tree with integer keys (Myers, 1984), or a tree where the bit pattern of array index is used to determine the position of the element in the tree (Hoogerwoord, 1992). This method has $O(\log n)$ time complexity for element read, and $O(\log n)$ time and space complexity for element update, where n is the size of the array. It is typically used when programmers have needed arrays and had to 'roll their own', especially in languages like Miranda (Turner, 1986) where it is not possible to extend the runtime environment to provide new functions which internally use some imperative structures, making trees practically the only choice.

Holmström (1983), Hughes (1985) and Aasa *et al.* (1988) have proposed a technique called *version tree arrays* or *trailer arrays*. The technique, which is in fact a rebirth of Henry Baker's *shallow binding* method (1978; 1991), provides excellent performance for single-threaded algorithms. However, while it does provide for non-single-threaded access, its performance when used as a persistent data structure can be arbitrarily bad, being linear in the number of updates between the old version accessed and the most recent version.

Chuang (1992) extended the version tree array technique to provide periodic cuts to the version tree, ensuring that reads of any array element costs at most $O(n)$, where n is the size of the array, in the fully persistent case, while continuing to provide $O(1)$ performance for single-threaded algorithms. Chuang's method also provides an operation to perform r reads, where r is $\Omega(n)$, in $O(r)$ amortized time. These *voluminous* reads are further restricted to be of versions that form a linear path in the version tree. Chuang states that this voluminous read operation may prove useful in practice for some non-single-threaded algorithms, but this still leaves the technique with an unfortunate $O(n)$ worst case performance.

Chuang later developed an alternative method for providing cuts to the version tree array based on randomization (1994). The idea is that instead of making cuts to ensure reads of any array element takes at most $O(n)$ steps, cuts are performed with a probability of $1/n$ during read operations. This method has expected worst case performance of $O(u + r + nu)$ for u updates and r reads of an initial array of size n, which we can restate as an expected amortized performance of $O(1 + nu/(r + u))$ per access. Chuang proves that this is within a factor of two of optimal for any strategy involving cuts to a version tree array. Chuang doesn't consider the theoretical space overheads of his algorithm, but it appears that the expected amortized space requirements are $O(1)$ space per element read, until the array versions take $O(nu)$ space in total; in other words, the upper bound on space is the same as the upper bound for naïvely copying the array at every update. This contrasts sharply with other techniques, which almost universally require no space consumption for element reads.

Prior to Chuang's work, Dietz (1989) presented, in extended abstract form, a technique that supports fully persistent arrays in $O(\log \log n)$ expected amortized time for read and update, and $O(1)$ space per update. Dietz's technique is, however, particularly complex, and it seems unlikely that it could be implemented without a large constant factor overhead, making it more theoretically interesting than useful in practice. Dietz's work in this area seems to have been largely overlooked by those working in this field (Baker, 1991; Chuang, 1992), perhaps because the method was never published as a complete paper. Dietz's technique is interesting however, since it is broadly similar to ours in its early stages, both being inspired by the work of Driscoll and company on persistent data structures (1989). However, beyond the common inspiration of Driscoll's work, our work and Dietz's differ significantly.

More overlooked than Dietz's work is that of Cohen (1984), being ignored by all the above authors, and almost overlooked by ourselves[1]. Cohen's method is similar to both our method and that of Dietz, being based on the idea of storing changes at the level of elements, rather than at the level of the array, and using a version counter. However, Cohen's method is less sophisticated than either Dietz's or our own, and claims[2] a worst case performance for read of $O(u_e)$, where u_e is the number of updates performed on element e (although it does also achieve updates in $O(1)$ time).

[1] Thanks to Chris Okasaki for bringing it to our attention.

[2] From our understanding of Cohen's work, his result only holds for partially persistent updates – a pathological fully persistent update sequence can actually cause accesses to take $O(u)$ time, where u is the total number of updates made to the array.

1.3 Improvements with our method

The weaknesses of existing schemes for implementing functional arrays have prompted us to develop an array method which can also provide $O(1)$ performance for single-threaded algorithms but has better typical use and worst case performance than prior solutions.

Many array algorithms access array elements *evenly*, where no element is read or updated (both of which we generically call *accessed*) more than k times the average for all elements in the array, where k is a constant. If an algorithm uses arrays evenly, we guarantee that the accesses will take $O(1)$ amortized time. The accesses do not even need to be on a single version of the array, but can be on any array in a realm of $O(n)$ *adjacent* versions (what we mean by this term is explained in section 2.1).

If an algorithm uses an array unevenly, the worst case performance of our technique is $O(\log n)$ amortized time for an access, with $O(1)$ amortized space cost for an update. This worst case performance is much better than Chuang's $O(n)$ performance for read accesses, and also better than binary trees, which are $O(\log n)$ time for all accesses and require $O(\log n)$ space for each update. Our technique is also likely to perform better than Dietz's method for most real cases, despite the better expected amortized asymptotic performance of his technique.

2 The fat elements method

In our opinion, most previous attempts at providing functional arrays have failed to achieve good worst-case bounds because they failed to capitalize on the structure of arrays[3]. The popular technique of *trailers*, for example, stores change lists, a technique that can in fact be applied to any data structure (Overmars, 1981; Overmars, 1983). The fact that each change can only be to a single element of the array is lost, and changes are only thought of as being applied to the array as a whole. Our approach, on the other hand, does notice that reads and updates are done on single elements, and so stores changes at the element level.

We use a scheme similar to the *fat node method* suggested by Driscoll *et al.* (1989), which was for implementing persistent linked data structures of bounded in-degree, but our scheme is adapted for use with arrays, and refined. As Driscoll's technique used *fat nodes*, so ours uses *fat elements*.

In our method[4], each array version receives a unique *version stamp*, which is used, along with fat elements, to store multiple array versions in a single master array. A fat element maps version stamps to values, being able to return the element's value for any version of the array. The master array is simply an array of fat elements. To ensure good performance, we guarantee that we never store more than $\Theta(n)$ versions

[3] Cohen's (Cohen, 1984) and Dietz's (1989) techniques do capitalize on the structure of arrays and thus do achieve good worst case bounds.

[4] The source for which is available though the JFP Internet home page (`http://www.dcs.gla.ac.uk/jfp/code/oneill-arrays.tar.gz`).

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

in a master array, where n is the size of the array, breaking it into two independent master arrays when necessary to preserve this condition.

2.1 Version stamps

The master array stores several array versions, and so requires some form of tagging mechanism so that data corresponding to a particular array version can be retrieved or stored. These tags are known as *version stamps*. We arrange for versions stamps to have an ordering, whereby an update on some array version with version stamp x_v creates a new array version with version stamp y_v, such that $x_v < y_v$.

If we only allowed the most recent version of an array to be updated (a restriction known as *partial persistence*), issuing version stamps would be simple; we could issue versions stamps from a simple integer counter. But while partial persistence would be sufficient for some algorithms, it wouldn't be satisfying in general, since the array interface would not be referentially transparent and the programmer would be left having to worry about which array version was the most recent and be sure to only update that version and no others.

Allowing any version of the array to be updated requires a more complex arrangement. If x, y and z are array versions, with version stamps x_v, y_v and z_v, respectively, and where both y and z are arrays derived from x though some sequence of updates, it is clear from the rule given earlier that $x_v < y_v$, and that $x_v < z_v$, but no ordering is defined between y_v and z_v. Thus, our requirements, as stated so far, only dictate a partially ordered version stamping scheme.

A partially ordered version stamping scheme would cause problems, however, because it would preclude the use of efficient data structures for storing data keyed by array version stamp. Also the generation of n partially ordered version stamps seems to require $O(n^2)$ space and time in the worst case, which is unacceptable.

Instead, we impose some additional structure that allows us to allocate version stamps under a totally ordered scheme. The additional structure results from saying if $x_v < y_v$ and $x_v < z_v$, and y_v was created before z_v, then $y_v > z_v$. More formally, we can state the rule as follows: let V be the set of all currently existing version stamps, and x be an array version, with version stamp x_v, if we update x to form a new version y, y will be given a version stamp y_v such that $x_v < y_v$ and $\forall v \in V$ s.t. $\mathrm{v} > \mathrm{x_v}, \mathrm{v} \geq \mathrm{y_v}$ – in other words, y_v is the least version greater than x_v.

This version scheme corresponds to an existing and well studied problem, the *list order problem*. The technique above inserts version y_v after some version, x_v, but before all the versions that follow x_v, which parallels inserting items into an ordered list. One solution to this problem would be to maintain a linked list of versions, and for efficient comparisons give each version a real number tag, where later versions in the list have greater tags. With this scheme a version inserted between two other versions would be given a tag that lies between the tags of those two versions. A problem with this version stamping method is that it requires arbitrary accuracy real arithmetic, which cannot be done in constant time. Practical solutions that take constant time for insertion, deletion, successor and predecessor queries, and comparisons do exist, however (Dietz and Sleator, 1987; Tsakalidis, 1984). We

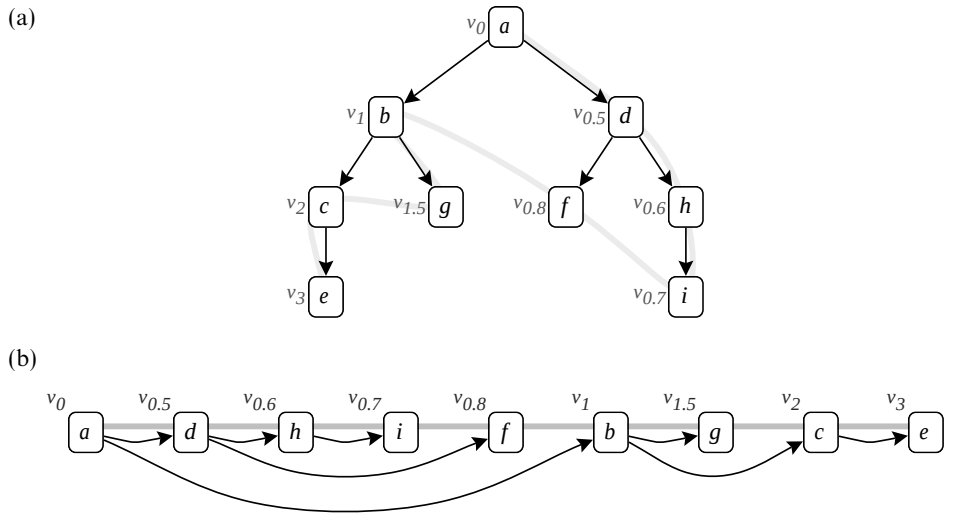

Fig. 1. Laying a total order on a version tree. (a) A version tree and the totally ordered version stamps that are applied to it. In the diagram arrows represent updates, and further, the array versions were created in alphabetical sequence, thus array version *c* was created before version *d* was, etc. (b) The same version tree flattened to better show the imposed total ordering.

include a description of Dietz and Sleator's method for version stamping in constant amortized time in the appendix to this paper.

The naïve scheme we have just outlined has some worth in its simplicity, however. While it may not be practical for efficient implementations, its directness is very useful for explaining the concepts of our method. In examples we will write specific version stamps as v_{tag} where *tag* is a real number following the naïve scheme (with subscripts rounded). Figure 1 gives an example of the naïve version stamp scheme, showing how our version stamping rules coerce the natural partial order of array versions into a total order.

Figure 1(b) also helps explain the concept of *adjacent versions*, which we will refer to later. Put simply, two versions are adjacent if there is no version between them. It should be clear from the discussion above that in the case of partially persistent (or single-threaded) updating, versions which are adjacent at one time will stay adjacent, because any new versions will always come after all previous versions, but with fully persistent updates, the new version may lie between two previous versions, making those two no longer adjacent.

2.2 The fat element data structure

As outlined earlier, a fat element is a data structure able to support a mapping from version stamps to values. Abstractly, we can imagine it storing a set of version stamp/value pairs.

Often array elements have the same value over several array versions, and our data structure will capitalize on this. The nature of array update is that it operates

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

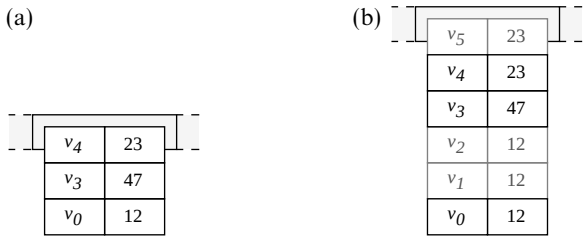

Fig. 2. Fat elements don't contain entries for every version stamp. (a) The value of this fat element changes at v_0, v_3 and v_4. Entries are omitted for versions v_1, v_2 and v_5, since their values can be inferred from the versions that preceded them. (b) Here we show not only the actual values stored in the fat element, but also the values that are inferred (shown in grey).

on a single element of the array – all other elements remain the same. We can capture this property by storing not a complete set of version stamp/value pairs, but just the entries for times when the element was changed, and inferring those that are missing.

Finding the correct value for a particular version stamp in this sparse set of pairs is still relatively straightforward. The value corresponding to some desired version stamp, v_d, can be found by finding the closest version stamp, v_c, in the fat element F, where

$$v_c = \max \{v \mid (v, x) \in F \wedge v \leq v_d\}$$

and retrieving the value corresponding to that version stamp. In other words, v_c is the greatest version stamp less than or equal to v_d. This technique is shown graphically in figure 2, which introduces the abstract diagrammatic notation we'll be using when representing fat elements.

Since we impose a total order on version stamps, we can use a relatively efficient data structure to provide the necessary insert and lookup operations, such as a height balanced mutable tree. Further, by using a *splay tree* (Sleator and Tarjan, 1985) instead of an ordinary balanced binary tree, we can gain the property that the most recently read or updated version in a fat element is at the top of the tree, and can thus be accessed in $O(1)$ time. It is this property which guarantees that accesses done by single-threaded algorithms will execute in $O(1)$ time (since every read will be of the root of the tree, and every update will make the new fat element entry the root, and the previous root its left child; thus creating a tree with no right children). The use of splay trees can also provide useful locality to some non-single-threaded algorithms, while guaranteeing $O(\log e)$ amortized worst case performance, where e is the number of entries in the fat element[5]. Later, we will ensure that $e \in O(n)$.

Putting fat elements together with our version stamping system developed in the preceding section, we have the basic data structure used for our functional array

[5] Splay trees are actually more powerful than is required to provide single-threaded algorithms with $O(1)$ performance – we need only cache the most recent addition to the tree for fast retrieval. Thus the additional complexity that accompanies splay trees can be eliminated, if that is desired.

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

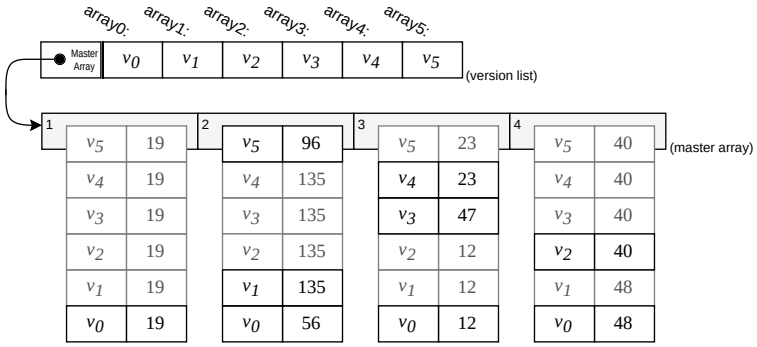

Fig. 3. Putting the pieces together.

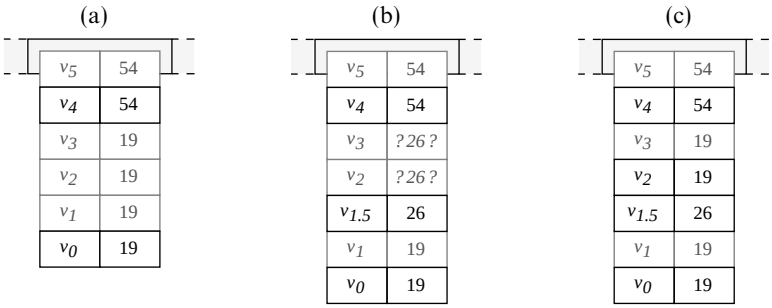

Fig. 4. Potential difficulties in using linear version stamps. (a) A fat element with gaps. The entry for v_0 is used to provide values for v_1, v_2 and v_3; similarly v_5 is inferred using the value for v_4. (b) Naïvely inserting a value for $v_{1.5}$ inside a gap, upsets the values of v_2 and v_3. Note that v_1. v_4 and v_5 are unaffected. (c) To ensure correct behaviour, we need to also insert an entry corresponding to v_2 (which was previously just inferred using the v_0 entry), before inserting $v_{1.5}$.

implementation (see figure 3). Note that the version list is set up such that, given any entry in the list, the pointer to the master array can be read in constant time, and updated in at most $O(v)$ time, where v is the number of versions.

There is, however, one flaw in our method, which we must rectify before we move on. It relates to *fully persistent* updates, which are updates done to array versions which have already received one update, and to an interaction between our totally ordered version stamping scheme and our method for slimming down fat elements[6] outlined above. The problem arises because fat elements need not contain entries for every version stamp – in other words they may contain 'gaps'. If an update to an element causes a version stamp/value pair to be added where there is a gap, the values inferred for some of the entries in the gap may change, when they should have remained the same. An example of this problem is shown in the first two parts of figure 4, where we see that adding elements into a gap can change the value of any inferred entries that lie immediately after it.

[6] Perhaps we should be calling them reduced fat elements?

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

We cannot prevent new fat element entries from being added in the places they are, but we can take steps to prevent the addition of new entries from affecting the values inferred for entries that lie after the one we've added. If we will be adding an entry, with version stamp v, to a fat element, we need to check whether there are any version stamps greater than v and, if so, whether v's immediate successor (in the version list) has no explicit entry in the fat element. If this is the case, we need to turn the implicit entry for v's successor into an explicit entry before adding v's entry (see figure 4(c)). This does mean that in this case we add two entries to the fat element, one for v's successor and one for v, but we never need to add more than two, no matter how big the gap.

We now have a working method for functional arrays. In our scheme, as presented so far, accesses to the most recently read or written value of an array element take constant time and, in general, reading or writing an array element which has been updated u_e times takes $O(\log u_e)$ amortized[7] time, and updates require $O(1)$ space. In the next section, we will improve on this result.

2.3 Breaking up is easy to do

Our goal now is to make things faster. We saw in the preceding section that a fat element is just an ordered tree of version stamp / value pairs and that if the tree has u nodes (as it will after $\Theta(u)$ updates to the element), reading or updating the element at a particular version will take $O(\log u)$ time, as we'd expect for a tree. Currently u is unbounded, and thus the more updates the element receives, the slower accesses become. If we could limit u, we would achieve a better worst case bound.

Our scheme for achieving a better worst case bound involves splitting the master array into two independent master arrays whenever the fat elements might be growing too fat. We will guarantee a master array hold no more than $O(n)$ versions, where n is the size of the array. More specifically, if c_* is the number of fat element entries in the master array, we impose the condition that $c_* \leq (1 + k_l)n$, where k_l is a positive constant[8]. When the master reaches its fatness limit, we split it in half, making two independent master arrays. Since it requires $\Theta(n)$ updates to take a master array from half full to full, we can amortize the cost of splitting the array of size n over those updates (a full amortized time analysis is presented in section 3).

To know when to break the array into two, we need to maintain some housekeeping information. With each version stamp, we'll associate a counter, c_i, holding the number of fat element entries that are associated with that version and a counter, c_*, for the total number of entries stored in the master array. When it comes time to break up the array, we'll split things so that entries corresponding to all versions

[7] This time result can be made $O(\log u_e)$ real time if a balanced tree is used instead of a splay tree, and a real time version stamping algorithm is used.

[8] The exact value of k_l is a matter of choice – testing has shown that when reads and writes are equally balanced, a value of around 4 is appropriate, with a lower value being better if reads are likely to outnumber writes.

up to the mth are placed in one master array, and all entries from versions $m+1$ onwards are put in another. We define m to be an integer such that

$$\sum_{i=0}^{m} c_i \leq \frac{c_* + n}{2} \geq n + \sum_{i=m+2}^{v} c_i$$

holds, where n is the size of the array, v is one less than the number of versions held by the master array, $c_0, \ldots, c_v$ are the counters for each entry in the version list, and c_* their sum (also, note that c_0 will always be equal to n). The left-hand and right-hand sums will be the sizes of the two master arrays after the split – the right-hand sum is $n + \sum_{i=m+2}^{v} c_i$ and not $\sum_{i=m+1}^{v} c_i$ because the first version in the second array will have to have entries for all n elements, regardless of the number of entries it had before. If there is more than one possible m, we choose the m such that $c_{(m+1)}$ is maximized, thus reducing the number of fat element entries that have to be added during the split.

The formula above can be implemented very straightforwardly as an iterative loop, and could also be generalized for splitting the master array into k master arrays, if that were desired.

Thus an upper bound on the number of fat element entries in each of the two new master arrays resulting from the split is $(c_* + n)/2$ (where c_* refers to the pre-split master array) and an obvious[9] lower bound on the number of entries is n. Since splitting only takes place when $c_* = (1 + k_l)n$ we can see that another upper bound on the number of fat element entries in each of the new master arrays is $(2 + k_l)n/2$.

Lemma 2.1
Splitting the master array into two independent master arrays takes $O(n)$ real time if splitting is done when $c_* = (1 + k_l)n$.

Proof
The steps involved in splitting the master array are:

- Allocating space for the second array, which requires $O(n)$ time.
- Finding the split point and chopping the version list into two at that point, which from the description above takes $O(v)$ time. Since $v < c_*$ and $c_* \in O(n)$, a loose bound on the time taken to find the split point is $O(n)$.
- Changing one of the two version lists so that it points to the new master array, which (as we learned at the close of section 2.2) takes at most $O(v)$ time, or more loosely $O(n)$ time.
- Splitting all the fat elements, which takes $O(n)$ time since splitting the splay tree for element i takes $O(\log e_i)$ amortized time and at most $O(e_i)$ real time[10], where e_i the number of fat element entries stored in that fat element. For simplicity, let's take the looser bound of $O(e_i)$ time to split. Thus the time to split all the elements is $O(\sum_{i=0}^{n-1} e_i) = O(e_*) = O(c_*) = O(n)$. (Note that when

[9] One can also manipulate the inequality above to show a tighter lower bound of $k_l n/2$ when $k_l \geq 2$, but we do not require this, with the added conditionality it brings.
[10] Some search tree structures might require $O(e_i)$ time if they were used instead of the splay tree, although AVL trees (Myers, 1984) can be split in $O(\log e_i)$ real time.

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

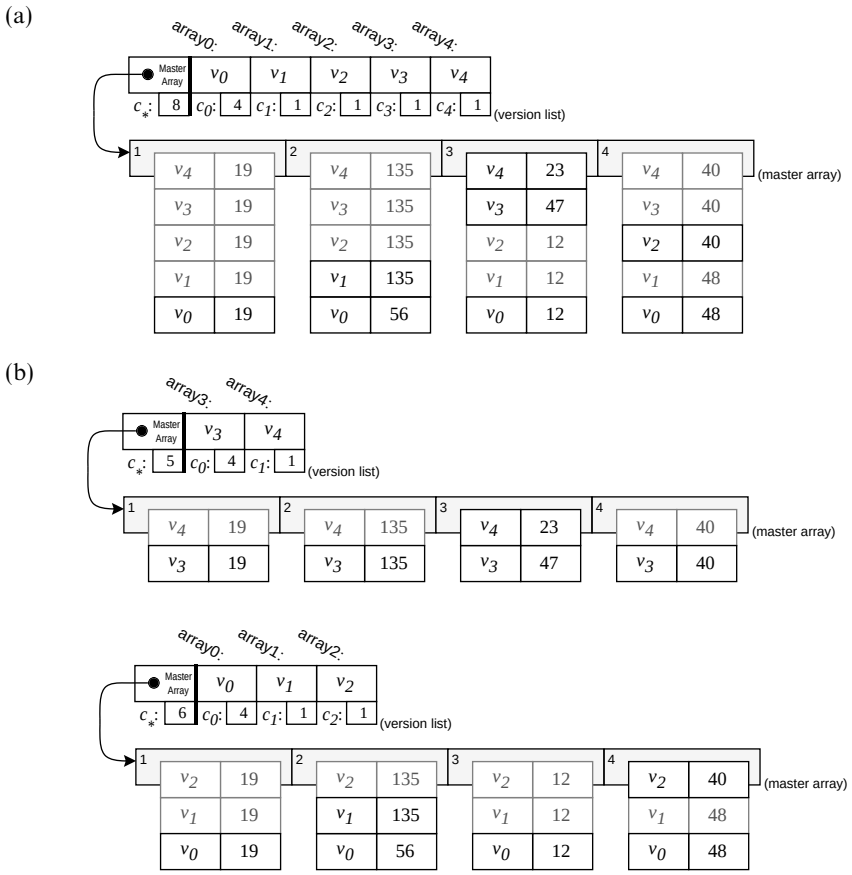

Fig. 5. Splitting a master array. (a) Here the master array holds a total of 8 fat element entries, which when $k_l = 1$ is the maximum number allowed. (b) If the master array receives another update, we need to split the array into two before applying the update in order that our invariants be preserved (i.e. in order that $c_* \leq (1 + k_l)n$). Here the midpoint, m, is 2, thus the first three versions form the first array (recall that numbering starts at zero), and the remaining two form the second array.

we split a fat element we need to ensure that it has an explicit entry for the version that is the split point, making an implicit entry into an explicit one if necessary.)

Since each of the steps takes is bounded by $O(n)$, the whole operation has this same bound. □

An example of splitting, with $c_* = 2n$, is shown in figure 5.

3 Amortized time analysis

We shall examine the amortized time complexity of our method using *potential functions* (Tarjan, 1985). The idea is that each configuration of the data structure is given a real number value called its *potential* (we may thus conceive of a function

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

which takes a data structure configuration and returns us this value). We may think of potential as representing stored energy which can be used to compensate for expensive operations, and thus define amortized work as follows:

$$\text{(amortized time)} \equiv \text{(time taken)} + \text{(increase in potential)}$$

or, using symbols to represent the words above $T_A \equiv T_R + \Delta_\Phi$.

We define the potential of our data structure in terms of *primitive potential*, ϕ, where:

$$\phi = k_\Phi \left(c_* - \frac{(k_l + 2)n}{2} \right)$$

where n is the size of the array, c_* is the total number of fat element entries in the master array, and k_Φ is a constant, k_l is the constant governing the size of the master array (i.e. $c_* \leq (1 + k_l)n$). The potential of our data structure, Φ is defined as

$$\Phi = \max(\phi, 0).$$

Since we will often be discussing changes in potential it is worth noting that when Δ_ϕ is positive, $\Delta_\Phi \leq \Delta_\phi$ and that since n is invariant, $\Delta_\phi = k_\Phi \Delta_{c_*}$. Both of these are obvious consequences of the equations above.

Lemma 3.1
Splitting a master array A at, or before, its maximum fullness (i.e. when $c_* \leq (1 + k_l)n$), using the algorithm of section 2.3, produces two independent arrays A' and A'' with primitive potential of at most 0, and hence potential of exactly 0.

Proof
The algorithm of section 2.3 splits the master array A into two arrays, A' and A''. Without loss of generality consider the first of these. Our algorithm of section 2.3 ensures that $c'_* \leq (c_* + n)/2$, and since $c_* \leq (1 + k_l)n$, we know that $c'_* \leq (k_l + 2)n/2$. Thus $\phi' \leq 0$, and hence $\Phi' = 0$. □

Lemma 3.2
Splitting a master array A at its maximum fullness (i.e. when $c_* = (1 + k_l)n$) takes zero amortized time, for a suitably chosen value of k_Φ.

Proof
The real time taken to perform the split T_R has the bound $T_R \leq k_s n$ real time (from Lemma 2.1).

After the split, we have two arrays, neither of which can be larger than $(c_* + n)/2$. Recall from Lemma 3.1 that the potential of the two arrays produced in the split (Φ' and Φ'') is 0. Thus the net change in potential, Δ_Φ, can be defined as

$$\begin{aligned} \Delta_\Phi &= (\Phi' + \Phi'') - \Phi \\ &= 0 - \Phi \\ &= -\frac{k_\Phi k_l n}{2} \end{aligned}$$

since $\Phi = k_\Phi k_l n/2$ when $c_* = (1 + k_l)n$. Since the amortized time, T_A is defined as

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

$T_A = T_R + \Delta_\Phi$, the amortized time taken T_A has the bound

$$\frac{k_\Phi k_l n}{2} T_A \quad \leq \quad k_s n - \frac{k_\Phi k_l n}{2}$$

If we define k_Φ such that $k_\Phi \geq 2k_s/k_l$ we find that $T_A \leq 0$. □

Lemma 3.3

Adding an entry to a fat element, x, takes at most $k_i \log e_x + k_\Phi$ amortized time, where e_x is the number of entries stored in fat element x, k_i is a constant of the tree insertion algorithm.

Proof

There are two cases to consider, one where splitting occurs, and one where splitting does not.

Case 1 : The master array is not full.

In this case the actual time taken to insert the entry will be the time taken to perform an insertion in the tree used to represent fat elements. We'll assume the tree insertion takes time bounded by $k_i \log e_x$, where k_i is a constant of the tree insertion algorithm. Both splay trees (Sleator and Tarjan, 1985) and balanced binary trees fit this assumption – although in the case of splay trees, it is amortized time, but the amortization analysis for splay trees can be considered independent of this analysis (since the potential of the splay trees has no effect on the potential of the master array that contains them); it just means that the resulting complexity contains two amortizations rather than one.

Now let us consider the increase in potential from the insertion. As we noted earlier, $\Delta_\Phi \leq \Delta_\phi$ and $\Delta_\phi = k_\Phi \Delta_{c_*}$. In the case of inserting one element, $\Delta_{c_*} = 1$, and thus $\Delta_\Phi \leq k_\Phi$.

Thus, the amortized time in this case is $T_A \leq k_i \log e_x + k_\Phi$.

Case 2 : The master array is full.

In this case we need to split the array into two arrays before performing the insertion, using the process outlined in section 2.3. This takes zero amortized time (Lemma 3.2).

After the split, we have two arrays, each with at most $(c_* + n)/2$ fat element entries. We'll be inserting a new fat element entry in just one of them.

Inserting a new fat element takes at most $k_i \log e'_x$ time, where e'_x is the size of the fat element in that half. Since $e'_x \leq e_x$, we can also say that the time taken is bounded by $k_i \log e_x$, and as in the previous case causes a change in potential for that array of at most k_Φ. The potential of the other array remains the same, and thus has no change in potential.

Thus, the amortized time $T_A \leq k_i \log e_x + k_\Phi$.

□

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

Lemma 3.4
Updating array version element, x, takes at most $2(k_i \log e_x + k_\Phi)$ amortized time, where e_x is the number of entries stored in fat element e of the array version's master array, and k_i is a constant of the tree insertion algorithm.

Proof
This trivially follows from Lemma 3.3, since in the worst case we may have to add two fat element entries for a single update (section 2.2). □

Lemma 3.5
Updating any element of an array version takes $O(\log n)$ amortized time. (Specifically it takes at most $2(k_i \log(k_l n + 1) + k_\Phi)$ amortized time).

Proof
The maximum size of a fat element e_x, in the master array is $e_x \leq c_* - n + 1$, and c_* has the bound $c_* \leq (1 + k_l)n$, thus $e_x \leq k_l n + 1$. Thus the Lemma follows from Lemma 3.4. □

Lemma 3.6
Reading any element, i, of an array version takes at most $k_r \log e_i$ amortized time, where e_i is the number of entries stored in fat element i of the array version's master array, and k_r is a constant of the tree lookup algorithm.

Proof
Finding the value of an element of an array version requires looking up that version in the master array associated with that array version. The fat element holds e_i entries, and we assume that our tree lookup algorithm takes at most $k_r \log e_i$ amortized time. Since reading the array does not change its potential, that bound is the total bound on the amortized time taken to perform the operation. □

Lemma 3.7
Reading any element of an array version takes at most $O(\log n)$ amortized time. (Specifically, it takes at most $k_r \log(k_l n + 1)$ amortized time, where k_r is a constant of the tree lookup algorithm.)

Proof
Analogously to Lemma 3.5. □

Lemma 3.8
Any sequence of read accesses that occur within a master array, in which no array element is accessed more than k_a times the average for elements of that master array, where k_a is a constant, takes $O(1)$ amortized time each. (Specifically, it takes at most $k_r \log(k_a(1 + k_l))$ amortized time, where k_r is a constant of the tree lookup algorithm – see Lemma 3.6.)

Proof
Rather than use the method of potential functions, we shall consider an arbitrary sequence of read accesses, in which each element, i, of the array, is accessed a_i times, with the total number of accesses being referred to as a_* (in other words,

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

$a_* = \sum_{i=0}^{n-1} a_i$), and each fat element corresponding to array element i contains e_i entries, with e_* being the sum of all the e_is (also note that $e_* = c_*$, since both count the total number of fat element entries in the master array).

The amortized time per access can be defined as

$$\begin{aligned} \text{amortized time} &= \frac{\text{time for a}_* \text{ accesses}}{a_*} \\ &\leq \frac{1}{a_*} \sum_{i=0}^{n-1} (a_i k_r \log e_i) \\ &= k_r \sum_{i=0}^{n-1} \left(\frac{a_i}{a_*} \log e_i \right) \end{aligned}$$

Since log is a convex function, we can at this point use Jensen's inequality (described in most texts on convex functions (e.g. Pecaric *et al.*, 1993)):

$$\sum_{i=0}^{n-1} \left(\frac{a_i}{a_*} \log e_i \right) \leq \log \left(\sum_{i=0}^{n-1} \left(\frac{a_i}{a_*} e_i \right) \right)$$

Now, recall that our condition was that each element was accessed no more than k_a times the average, i.e.

$$\forall (0 \leq i < n), \quad a_i \leq \frac{k_a a_*}{n}$$

thus, we can now say:

$$\begin{aligned} \text{amortized time} &\leq k_r \log \left(\sum_{i=0}^{n-1} \left(\frac{\left(\frac{k_a a_*}{n} \right)}{a_*} e_i \right) \right) \\ &= k_r \log \left(\frac{k_a}{n} \sum_{i=0}^{n-1} e_i \right) \\ &= k_r \log \left(\frac{k_a}{n} e_* \right) \end{aligned}$$

But $e_* = c_*$ and $c_* \leq (1 + k_l)n$, thus:

$$\text{amortized time} \leq k_r \log (k_a (1 + k_l))$$

□

Notice that the conditions of the above lemma do not require that all the elements be accessed, some percentage may be ignored completely.

Lemma 3.9

The number of updates, u, required to take an initial array to the point where it needs to be split is $\Theta(n)$ (specifically $(k_l n + 1)/2 \leq u \leq k_l n$).

Proof

Initially, the number of fat element entries e_* (also c_*) is n, but at the point when the array is split $c_* = (1 + k_l)n$, thus $k_l n$ fat element entries must have been added. In the upper bound case, every update adds exactly one fat element entry, meaning that $u \leq k_l n$. In the lower bound case every update, except the first, adds two fat element entries, thus $2(u - 1) + 1 \geq k_l n$, which simplifies to $u \geq (k_l n + 1)/2$. □

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

Lemma 3.10
The number of updates, u, required to take an array which has just been split to the point where it needs to be split again is $\Theta(n)$ (specifically $k_l n/4 \leq u \leq k_l n$).

Proof
An array which has just been split will contain $n \leq e_* \leq (2 + k_l)n/2$ entries, but at the point when the array is split $c_* = (1 + k_l)n$. Thus, the upper bound case is the same as that of Lemma 3.9, thus $u \leq k_l n$. In the lower bound case, we assume that the split array has greatest possible size, $(2 + k_l)n/2$, thus to reach the splitting size of $(1 + k_l)n$, only $k_l n/2$ fat element entries have to be added. Since in the worst case, every update may add two fat element entries, $2u \geq k_l n/2$, or $u \geq k_l n/4$. □

Lemma 3.11
If an initial array receives $\Theta(n)$ updates (with the updates being made to any array in the version tree stemming from that initial array), this will create $\Theta(1)$ master arrays.

Proof
Trivially from Lemma 3.9 and Lemma 3.10. □

Lemma 3.12
Any sequence of read accesses that occur within m master arrays, in which no array element is accessed more than k_a times the average for elements of those m master arrays, where k_a is a constant, takes $O(\log m)$ amortized time. (Specifically amortized time bounded by $k_r \log(k_a(1 + k_l)m)$, where k_r is a constant of the tree lookup algorithm – see Lemma 3.6.)

Proof
This proof is analogous to that of Lemma 3.8, except that we need to consider multiple array versions. In this case, we shall consider an arbitrary sequence of read accesses, in which each element i of master array j is accessed $a_{i,j}$ times, with the total number of accesses being referred to as a_{**} (in other words, $a_{**} = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} a_{i,j}$), and each fat element corresponding to element i of master array j contains $e_{i,j}$ entries, with e_{**} being $\sum_{j=0}^{m-1} \sum_{i=0}^{n-1} e_{i,j}$.

The amortized time per access can be defined as

$$\begin{aligned}
\text{amortized time} &= \frac{\text{time for } a_{**} \text{ accesses}}{a_{**}} \\
&\leq \frac{1}{a_{**}} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \left(a_{i,j} k_r \log e_{i,j}\right) \\
&= k_r \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \left(\frac{a_{i,j}}{a_{**}} \log e_{i,j}\right) \\
&\leq \log \left(\sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \left(\frac{a_{i,j}}{a_{**}} e_{i,j}\right)\right)
\end{aligned}$$

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

Now, recall that our condition was that each element was accessed no more than k_a times the average, i.e.

$$\forall(0 \leq j < m, 0 \leq i < n), \quad a_{i,j} \leq \frac{k_a a_{**}}{n}$$

thus, we can now say:

$$\begin{aligned} \text{amortized time} &\leq k_r \log \left(\sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \left(\frac{\left(\frac{k_a a_{**}}{n}\right)}{a_{**}} e_{i,j} \right) \right) \\ &= k_r \log \left(\frac{k_a}{n} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} e_{i,j} \right) \\ &= k_r \log \left(\frac{k_a}{n} e_{**} \right) \end{aligned}$$

But $e_{**} \leq m((1 + k_l)n)$, thus:

$$\text{amortized time} \leq k_r \log (k_a(1 + k_l)m).$$

□

Corollary 3.1
A collection of read accesses, across a set of versions created from an initial array version by $O(n)$ updates, in which no element is accessed more than k_a times the average for all elements takes $O(1)$ amortized time per access.

Proof
From Lemma 3.11, the $O(n)$ versions created by the updates will reside in a constant number of master arrays. Hence this follows from Lemma 3.12. □

Corollary 3.2
A collection of arbitrary read or write accesses, across set of versions created from an initial array version by $O(n)$ updates, in which no element is accessed more than k_a times the average for all elements and no more than $O(n)$ updates are made takes $O(1)$ amortized time per access.

Proof
This is analogous to Lemma 3.12 and Corollary 3.1, except that we have the issue that splitting may occur because we allow updates. However, the $O(n)$ updates can only cause a constant number of splits (from Lemma 3.11), meaning that there is a constant upper bound on the number of master arrays. This constant bound on the number of master arrays and thus $O(n)$ bound on the total number of fat element entries allows us to generalize Corollary 3.1 to cover updates as well as reads in this case. □

Thus, our data structure offers constant amortized time performance when array versions are accessed evenly or single-threadedly, and even when they are not, it turns in a quite reasonable $O(\log n)$ worst case amortized performance.

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

4 Real world performance

In the previous section we showed that the fat element array method has nice theoretical properties, but whether a data structure is useful depends not only on its theoretical complexity but on the constant factor overheads involved in using the data structure in the real world and how they compare to those of other data structures aimed at doing the same job. Our intention in this section is to briefly examine how a Standard ML implementation of our data structure fared against ML implementations of competing techniques.

Thus, our goal is not to try to deduce the asymptotic complexity of our technique from experimental results – we have already found these results in the preceding section. In fact to try to make such inferences would be difficult, since our experimental results often have small anomalies that result from the complex interactions found on a modern computer system, with processor caches, virtual memory, and running a large language such as ML. These anomalies however, affect neither the deductions we make about usability, nor the fundamental properties of the algorithms found in the preceding section. The interested reader is welcome to investigate these performance issues themselves by downloading our code for performance testing and conducting their own runs of the tests described here, the source being available through the JFP Internet home page (`http://www.dcs.gla.ac.uk/jfp/code/oneill-arrays.tar.gz`).

We compared the performance of our data structure against that of an implementation of functional arrays using binary trees and an implementation using the trailers technique. Our tree based implementation was based on the implementation in *ML for the Working Programmer* (Paulson, 1991), which is inspired by the work of Hoogerwoord ((1992)), and our implementation of Trailers was based on that of Aasa *et al.* (1988).

Lacking well known benchmarks for functional arrays, we used three fairly simple tests to examine the behaviour of the data structures, one that treated arrays as a fully persistent data structure, one that treated them as a partially persistent data structure, and one that treated them as an ephemeral data structure.

Our first test created v different versions of an array of size n, with each version depending on three randomly selected prior versions of the array. To do this it repeatedly updated a random element of a randomly selected array version with the sum of random elements of two other array randomly selected array versions. Figure 6 shows the results of this test.

Figure 6(a) shows how the performance of the techniques vary as we increase both the number of different versions and the size of the array (with $n = v$). Notice that while the Fat Element technique has a worst case $O(\log n)$ performance, the random element accesses cover the array evenly, causing us to expect $O(1)$ performance. The graph mostly echos these expectations, but in any case, the fat element method turns in performance which is better in real time than the competing techniques.

Figure 6(b) shows how the performance of the techniques vary as we increase versions while holding the size of the array constant (at 2^{16} elements). From theory, we would expect the performance of binary trees to depend only on the size of the array,

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

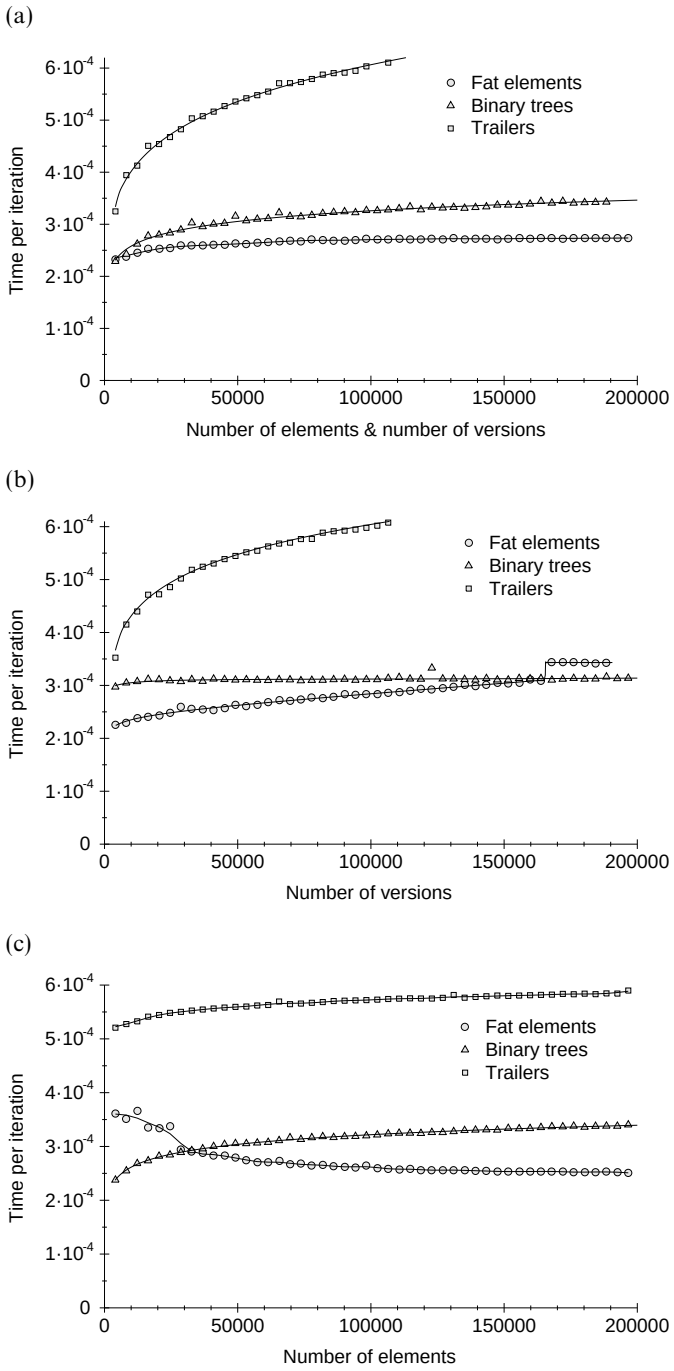

Fig. 6. Results from the multi-version test. (a) Varying n and v. (b) Varying only v ($n = 2^{16}$). (c) Varying only n ($v = 2^{16}$).

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

and we would expect the time per access to stay constant. The graph mostly echos this, but has some anomalous behaviour for small arrays. From theory we might also expect the performance of fat elements to take a constant time per iteration. However, while there is a constant upper bound, we see that actual times increase as we increase versions until splitting begins, which is the source of the discontinuity on the graph. Unfortunately, with this array size our technique is slower than binary trees for large numbers of versions. However, it's still very close in speed and one should bear in mind that for larger arrays trees will take longer, and require $O(\log n)$ space per update compared to $O(1)$ for fat elements. This does show, however, that for the particular case of small arrays with many versions, binary trees may be a preferable technique.

Figure 6(c) shows how the performance of the techniques vary as we increase the size of the array while holding the number of versions constant (at 2^{16}). Again, while the performance of fat elements is bounded by a constant, we see variations in times, and see the overheads of splitting cause it to turn in worse performance than binary trees for small n. As array size increases, we see fat elements turn in better performance than the other techniques.

The remaining two tests test the array data structures in simple ephemeral and partially persistent situations. For this we used the simple test of reversing an array. The first of the two (shown in figure 7(a)) uses the standard imperative array reversal algorithm, swapping leftmost and rightmost elements and working inwards until the whole array is reversed. The second (shown in figure 7(b)) uses a partially persistent method, performing updates from left to right, always reading from the original array, but updating the most recently changed array.

We can see in figure 7(a) that trailers significantly outperform both fat elements and binary trees for imperative algorithms. This comes as no real surprise, since trailers are primarily designed to support imperative algorithms. We can also see, however, that while fat elements are not the fastest technique in this case, they outperform binary trees, and fit our theoretical expectations of $O(1)$ performance.

Figure 7(b) reveals the weaknesses of the trailers method, however, while binary trees and fat elements turn in virtually the same performance for this slight variation in the algorithm, we know from theory that trailers will fall into a terrible $O(n^2)$ performance hole (it doesn't even make it onto the graph, taking 0.04 seconds per iteration when $n = 4096$). Even if we had used the first of Chuang's (1992) techniques for speeding up trailers, it would have still seen the same $O(n^2)$ behaviour (Chuang's second, probablistic, method (1994) would have expected $O(n)$ performance for this test, but as we noted earlier, this method has its own problems when it comes to space usage.)

In this section, then, we have seen that, as well as having better theoretical asymptotic performance, our technique is competitive with other techniques and does offer better real time performance in many cases. In particular, unlike trailers and balanced trees, fat elements yield reasonable performance for every case, making them the most suitable general purpose functional array implementation.

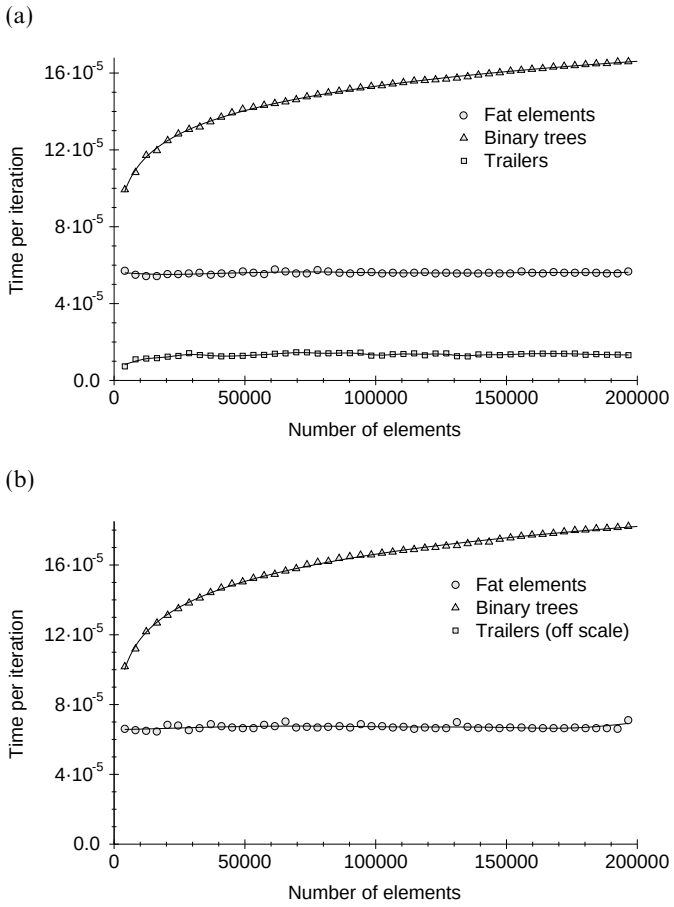

Fig. 7. Results from the two tests based on array reversal. (a) Ephemeral array reversal. (b) Partially persistent array reversal.

5 Conclusion

Our goal was to develop a functional array data structure which offers good theoretical performance, and also is not prohibitively complex and slow to use in practice. We feel we have achieved our goal, offering better performance than either tree based, or trailer based arrays. Obviously, our functional array scheme cannot ever be as fast as imperative arrays, as we are solving a more complex and general problem. Thus, even though we have constant time performance for single-threaded algorithms, our technique should not be viewed as a drop in replacement for imperative arrays – they still have their place.

Functional arrays belong in situations where either persistence is required, or where one cannot be certain that persistence isn't required. Thus our technique can act as a nice adjunct to other techniques for arrays, such as update analysis (Bloss, 1989). In cases where abstract interpretation can determine that arrays are used single-threadedly, raw imperative arrays may be used, but when that determination fails, fat element arrays can be used. In fact, such an analysis can even be used for

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

programs which are only partially single-threaded, and the information gained can be used to allow update in place on the fat element arrays when that would be safe. Runtime techniques that can catch safe opportunities for update-in-place, such as single bit reference counting (Wise and Friedman, 1977; Stoye *et al*., 1984) can also be integrated with fat element arrays.

While it would seem that no technique is likely to provide all the answers, the method we have developed here does seem to be an improvement over previous work. It is probably the first kind of array where the programmer is free to use arrays in whatever way they like without worrying too much about whether they are going to pay a terrible penalty for that freedom.

A Appendix: The list order problem

In this appendix we present an algorithm which addresses the list order problem, which, as we have seen, corresponds exactly to our version stamping scheme. The algorithm that we present has been presented previously by Dietz and Sleator (1987), but we include it here because the generation and comparison of version stamps is a fundamental part of our method, and to show that version stamp generation can be done relatively easily.

Before we begin, we should note a few things about our choice of algorithm. We have chosen to present the simplest practical solution to the list order problem, but other solutions do exist. The technique we present requires $O(1)$ amortized time and space for insertion, deletion and comparison. The other solutions to the list order problem (Dietz and Sleator, 1987; Tsakalidis, 1984) include an $O(1)$ real time algorithm, but this offers little additional benefit in our application.

Although the algorithm has been presented before, our presentation of it may be of some interest to those who might encounter it elsewhere, since we present it from a slightly different perspective, and reveal some properties that may not have been obvious in the original presentation[11]. Note, however, that for brevity we omit proofs of the complexity of this algorithm, referring the interested reader to the original paper (Dietz and Sleator, 1987) for such matters.

We will start by presenting an $O(\log n)$ amortized time solution to the list order problem, which we will then refine to give the desired $O(1)$ amortized time solution.

An $O(\log n)$ solution to the list order problem

The algorithm maintains the ordered list as a circularly linked list. Each node in the list has an integer label, which is occasionally revised. For any run of the algorithm, we need to know N, the maximum number of versions that might be created. This upper limit could be decided for each run of the algorithm, or, more typically, be

[11] In particular, we show that it is not necessary to refer to the 'base' when performing insertions; it is only necessary for comparisons. Also, some of the formulas given by Dietz and Sleator would, if implemented as presented, cause problems with overflow (in effect causing ' mod M' to be prematurely applied) if M is chosen, as suggested, to exactly fit the machine word size.

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

fixed by an implementation. The value N should be influenced by the fact that the larger the value of N, the *faster* the algorithm runs (because it operates using an interval subdivision technique), but that real world considerations[12] will likely preclude the choice of an extremely large value for N. In cases where N is fixed by an implementation, it would probably be the largest value that can fit in a machine word, or perhaps a machine half-word (see below).

The integers used to label the nodes range from 0 to $M-1$, where $M > N^2$. In practice, this means that if we wished N to be $2^{32}-1$, we would need to set M to 2^{64}. If it is known that a large value for N is not required, it may be useful for an implementation to fix M to be 2^w, where w is the machine word size, since much of the arithmetic needs to be performed modulo M, and when M is the machine word size this will happen automatically.

In the discussion that follows, we shall use $l(e)$ to denote the label of an element e, and $s(e)$ to denote its successor in the list. We shall also use the term $s^n(e)$ to refer to the nth successor of e, for example, $s^3(e)$ refers to $s(s(s(e)))$. Finally, we define two 'gap' calculation functions, $g(e,f)$ and $g^*(e,f)$, that find the gap between the labels of two elements:

$$g(e,f) = (l(f) - l(e)) \mod M$$

$$g^*(e,f) = \begin{cases} g(e,f) & \text{if } e \neq f \\ M & \text{if } e = f. \end{cases}$$

To compare two elements of the list for order, we require the base, as well as the elements which are to be compared. If we are comparing two elements, x and y, we perform a simple integer comparison of $g(base,x)$ with $g(base,y)$, where *base* is the first element in the list.

Comparison of elements is trivial then, as is deletion, which is done just by removing the element from the list. The only issue that remains is that of insertion. Suppose that we wish to place a new element, i, so that it comes directly after some element, e. For most insertions, all that needs to be done is to select a new label that lies between $l(e)$ and $l(s(e))$. The label for this new node can be derived as follows:

$$l(i) = \left(l(e) + \left\lfloor \frac{g^*(e,s(e))}{2} \right\rfloor \right) \mod M.$$

This approach is only successful, however, if the gap between the labels of e and its successor is greater than 1 (i.e. $g(e,s(e)) > 1$), since there needs to be room for the new label. If this is not the case, it is necessary to relabel some of the elements in the list to make room. Thus we relabel a stretch of j nodes, starting at e, where j is chosen to be the least integer such that $g(e,s^j(e)) > j^2$. (The appropriate value of j can be found by simply stepping through the list until this condition is met). In fact, the label for e is left as is, and so only the $j-1$ nodes that succeed e need

[12] Such as the fact that arithmetic on arbitrarily huge integers cannot be done in constant time. In fact, if we *could* do arithmetic on arbitrary sized rationals in constant time, we wouldn't need this algorithm, since we could use the naïve version stamping scheme outlined in Section 2.1.

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

have their labels updated. The new labels for the nodes $s^1(e), \ldots, s^{j-1}(e)$ are assigned using the formula below:

$$l(s^k(e)) = \left(l(e) + \left\lfloor \frac{k \times g^*(e, s^j(e))}{j} \right\rfloor \right) \mod M.$$

Having relabeled the nodes to create sufficient gap, we can then insert a new node following the procedure outlined earlier.

Refining the algorithm to $O(1)$ performance

The algorithm, as presented so far, takes $O(\log n)$ amortized time to perform an insertion (Dietz and Sleator, 1987), but there is a simple extension of the algorithm which allows it to take $O(1)$ amortized time per insertion(Tsakalidis, 1984; Dietz and Sleator, 1987). To do this, we use a two-level hierarchy that uses an ordered list of ordered sublists.

The top level of the hierarchy is represented using the techniques outlined earlier, but each node in the list contains an ordered sublist which forms the lower part of the hierarchy. An order list element, e, is now represented by a node in the lower (child) list, $c(e)$, and a node in the upper (parent) list, $p(e)$. Nodes that belong to the same sublist will share the same node in the upper list. In other words

$$p(e) = p(f), \ \forall e, f \text{ s.t. } c(e) = s_c(c(f))$$

where $s_c(e_c)$ is the successor of sublist element e_c. We also define $s_p(e_p)$, $l_c(e_c)$ and $l_p(e_p)$ analogously.

The sublists have their order maintained using a simpler algorithm. Each sublist initially contains $\lceil \log n_0 \rceil$ elements, where n_0 is the total number of items in the ordered list we are representing. This means that the parent order list contains $n_0 / \log n_0$ entries.

Each sublist element receives an integer label, such that the labels of the elements are, initially, $k, 2k, \ldots, \lceil \log n_0 \rceil k$, where $k = 2^{\lceil \log n_0 \rceil}$. When a new element, n_c, is inserted into a sublist, after some element e_c, we choose a label in between e_c and $s_c(e_c)$. More formally,

$$l_c(n_c) = \left\lceil \frac{l_c(e_c) + l_c(s_c(e_c))}{2} \right\rceil$$

Under this arrangement, the sublist can receive at least $\lceil \log n_0 \rceil$ insertions before there is any risk of there not being an integer label available that lies between e_c and $s_c(e_c)$.

To insert an element i after e in the overall order list, if the sublist that contains $c(e)$ has sufficient space, all that needs to be done is to insert a new sublist element i_c after $c(e)$, and perform the assignments $c(i) \leftarrow n_c$ and $p(i) \leftarrow p(e)$. If the sublist contains $2\lceil \log n_0 \rceil$ elements, it may not be possible to make insertions after some of its elements, however. In this case, we split the sublist into two sublists of equal length, relabeling both sets of $\lceil \log n_0 \rceil$ nodes following the initial labeling scheme. The nodes of the first sublist are left with the same parent, e_p, but nodes of the

https://doi.org/10.1017/S0956796897002852 Published online by Cambridge University Press

second sublist are given a new parent, i_p that is inserted in the upper order list immediately after e_p.

These techniques are used for insertions until the number of nodes, n, in the overall order list is greater than $2^{\lceil \log n_0 \rceil}$, since at that point $\lceil \log n \rceil > \lceil \log n_0 \rceil$. When this happens (every time n doubles), we must reorganize the list so that we now have $n/\lceil \log n \rceil$ sublists each containing $\lceil \log n \rceil$ nodes, rather than having $n/\lceil \log n_0 \rceil$ sublists of $\lceil \log n_0 \rceil$ nodes.

Since this new scheme only creates $n/\lceil \log n \rceil$ entries in the upper order list, M, can be slightly lower. Recall that previously we imposed the condition $M > N^2$. Now we have a slightly smaller M, since it need only satisfy the condition:

$$M > (N/\lceil \log N \rceil)^2$$

In practice, this would mean that if we required up to 2^{32} list entries, we would need an arena size of 2^{54} (instead of 2^{64}). Similarly, if we wished all labels to fit in a machine word, and so wished M to be 2^{32}, we would be able to have a little over 2^{20} items in an order list at one time (instead of 2^{16} items).

Following this scheme then, we can implement efficient ordered lists and by simple derivation, a quick and effective scheme for totally ordered version stamps.

References

Aasa, A., Holmström, S. and Nilsson, C. (1988) An efficiency comparison of some representations of purely functional arrays. *BIT*, **28**(3): 490–503.

Achten, P. M., van Groningen, J. H. G. and Plasmeijer, M. J. (1993) High-level specification of I/O in functional languages. In: Launchbury *et al.*, editors, *Proceedings of the Glasgow Workshop on Functional Programming*. Springer-Verlag.

Baker Jr., H. G. (1978) Shallow binding in lisp 1.5. *Comm. ACM*, **21**(7): 565–569.

Baker Jr, H. G. (1991) Shallow binding makes functional arrays fast. *SIGPLAN Notices*, **26**(8): 145–147.

Bloss, A. (1989) Update analysis and the efficient implementation of functional aggregates. In: *Functional Programming Languages and Computer Architecture, London.* ACM.

Chuang, T.-R. (1992) Fully persistent arrays for efficient incremental updates and voluminous reads. In: Krieg–Brückner, B., editor, *ESOP '92: 4th European Symposium on Programming: Lecture Notes in Computer Science 582* pp. 110–129. Springer-Verlag.

Chuang, T.-R. (1994) A randomized implementation of multiple functional arrays. *Conference Record of the 1994 ACM Conference on LISP and Functional Programming*, pp. 173–184.

Cohen, S. (1984) Multi-version structures in Prolog. *International Conference on Fifth Generation Computer Systems*, pp. 265–274.

Dietz, P. F. (1989) Fully persistent arrays (extended abstract). In: Dehne, F., Sack, J.-R. and Santoro, N., editors, *Proceedings of the Workshop on Algorithms and Data Structures: WADS'89*, pp. 67–74. *Lecture Notes in Computer Science 382.* Springer-Verlag.

Dietz, P. F. and Sleator, D. D. (1987) *Two algorithms for maintaining order in a list. To appear.* (A preliminary version appeared in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, New York, May 25–27 1987.)

Driscoll, J. R., Sarnak, N., Sleator, D. D. and Tarjan, R. E. (1989) Making data structures persistent. *J. Computer & System Sci.*, **38**: 86–124.

Holmström, S. (1983) How to handle large data structures in functional languages. *Proceedings of the SERC Chalmers Workshop on Declarative Programming Languages.*

Hoogerwoord, R. R. (1992) A logarithmic implementation of flexible arrays. *Proceedings of the 2nd International Conference on Mathematics of Program Construction*, pp. 191–207. *Lecture Notes in Computer Science 669*. Springer-Verlag.

Hudak, P., Peyton Jones, S. L., Wadler, P. L., Arvind, Bontel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughs, J., Johnson, T., Kieburtz, R, Partain, W. and Peterson, J. (1992) Report on the functional programming language Haskell. version 1.2. *SIGPLAN Notices*, **27**, July.

Hughes, J. (1985) An efficient implementation of purely functional arrays. *Technical Report*, Department of Computer Sciences, Chalmers University of Technology.

Huitema, H. S. and Plasmeijer, M. J. (1992) The Concurrent Clean System Users Manual, version 0.8. *Technical Report 92-19*, University of Nijmegen.

Myers, E. W. (1984) Efficient applicative data types. In: Kennedy, K., editor, *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pp. 66–75. Salt Lake City, UT: ACM Press.

Overmars, M. H. (1981) Searching in the past II: General transforms. *Technical Report RUU–CS–81–9*, Department of Computer Science, University of Utrecht.

Overmars, M. H. (1983) *The design of dynamic data structures: Lecture Notes in Computer Science 156*, pp. 153–157. Springer-Verlag.

Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

Pecaric, J. E., Proschan, F. and Tong, Y. L. (1993) *Convex functions, partial orderings, and statistical applications*. Mathematics in Science and Engineering, vol. 187. Academic Press.

Schmidt, D. A. (1985) Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, **7**: 299–310.

Sleator, D. D. and Tarjan, R. E. (1985) Self-adjusting binary search trees. *J. ACM*, **32**(3): 652–686.

Stoye, W. R., Clarke, T. J. W. and Norman, A. C. (1984) Some practical methods for rapid combinator reduction. *ACM symposium on LISP and functional programming*, pp. 159–166. Austin, Texas: ACM.

Tarjan, R. E. (1985) Amortized computational complexity. *SIAM J. Algebraic and Discrete Methods*, **6**(2): 306–318.

Tsakalidis, A. K. (1984) Maintaining order in a generalized linked list. *Acta informatica*, **21**(1): 101–112.

Turner, D. A. (1986) An overview of Miranda. *SIGPLAN Notices*, **21**(12): 158–166.

Wadler, P. L. (1990a) Comprehending monads. *Proceedings of the ACM conference on Lisp and functional programming, Nice*. ACM.

Wadler, P. L. (1990b) Linear types can change the world! In: Broy, M. and Jones, C., editors, *Programming Concepts and Methods*. North Holland.

Wadler, P. L. (1992) The essence of functional programming. *Conference record of the 19th annual ACM symposium on principles of programming languages*. New Orleans, Louisiana. ACM.

Wise, D. S. and Friedman, D. P. (1977) The one-bit reference count. *BIT*, **17**(3): 351–359.