A Data Structure for More Efficient Runtime Support of Truly Functional Arrays

by

Melissa Elizabeth O'Neill

B.Sc., University of East Anglia, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Melissa Elizabeth O'Neill 1994
SIMON FRASER UNIVERSITY
April 1994

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Melissa Elizabeth O'Neill

Degree: Master of Science

Title of thesis: A Data Structure for More Efficient Runtime Support of Truly Functional Arrays

Examining Committee: Dr. Bill Havens
Chair

____

Dr. F. Warren Burton
Senior Supervisor

____

Dr. Robert Cameron
Supervisor

____

Dr. Fred Popowich
Examiner

Date Approved: _______________________________

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A Data Structure for More Efficient Runtime Support of Truly Functional .

Arrays.

Author: ______________________
(signature)

Melissa O'Neill
(name)

April 11, 1994
(date)

Abstract

Functional languages often neglect the array construct because it is hard to implement nicely in a functional language. When it is considered, it is usually from the context of converting imperative algorithms into functional programs, rather than from a truly functional perspective.

In an imperative language, changes to an array are done by modifying array elements, destroying their original values. Unless special measures are taken to support destructive update, an array update in a functional language must produce a new array, without destroying the old one. Since most other functional data structures do not support destructive update, it would be desirable not to have to make a special case of arrays, especially since many kinds of algorithms (backtracking algorithms being a simple example) may find having multiple versions of a data-structure useful. Our goal is to be able to provide a functional array interface where array operations are reasonably cheap.

We will look existing techniques that have been used in the past to address this problem area, and then present a new runtime technique that offers very good all-round performance, and can be used where other array mechanisms fail.

Acknowledgments

The author is extremely grateful for the continuing support and guidance of her committee and in particular her supervisor, Dr. F. Warren Burton. His constructive comments have helped shape this thesis, and his enthusiasm for the topic has made this reseach a pleasure to undertake.

About the Author

Melissa O'Neill was born in England, growing up in the peaceful sea side resort town of Worthing in Sussex. She undertook her undergraduate degree at the University of East Anglia, in Norwich, England, gaining a First Class honours degree. It was at UEA where she first came into contact with functional programming languages, which she has made the focus of her graduate research studies here at Simon Fraser University.

Contents

Abstract iii
Acknowledgments iv
About the Author v
List of Tables ix
List of Figures x
1 Introduction 1
1.1 Functional vs. Imperative Languages 2
1.2 Arrays in Functional Languages 5
1.3 Thesis Outline 6
1.4 A Word on Notation 7
2 Review of Earlier Work on Functional Arrays 9
2.1 Single Threaded Approaches 9
2.1.1 Trailers 10
2.1.2 Monads 12
2.1.3 Linear Types 15
2.1.4 Compile Time Analysis 16
2.2 Non Single Threaded Approaches 17
2.2.1 Monolithic Approaches 17
2.2.2 Faking it, with Trees 18
2.3 Non-Functional Approaches 20
2.4 Conclusion 20
3 The Essence of Our Method 22
3.1 Basic Operations for Arrays 22
3.2 Origins of Our Method 23

3.3 Data Structure Overview 25
3.4 Element Histories 25
3.5 Array Values and the Master Array 27
3.6 Conclusion 28
4 Providing Full Persistence 30
4.1 The Problem 30
4.2 Partially and Totally Ordered Timestamping 30
4.2.1 Partially Ordered Timestamping 32
4.2.2 Totally Ordered Timestamping 33
4.3 Algorithms for Partially Ordered Timestamps 34
4.3.1 A Simple Timestamping Algorithm 34
4.3.2 Improving the Algorithm 35
4.3.3 Conclusions 36
4.4 Algorithms for Totally Ordered Timestamps 38
4.4.1 A Simple Timestamping Algorithm 39
4.4.2 Refining the Algorithm 41
4.4.3 Conclusions 43
4.5 Correctly Implementing Element Histories 43
4.6 Conclusion 47
5 Performance Issues 48
5.1 The Issue of Performance 48
5.2 A Simple Test, Reversing an Array 49
5.3 The Multiple Versions Test 55
5.4 Conclusion 55
6 Case Study, Implementing Heaps 58
6.1 Introduction 58
6.2 A Straightforward Heap Implementation 59
6.3 Problems over Referential Transparency 61
6.4 Optimising the Implementation of Heaps 63
6.5 Conclusion 65
7 Conclusion 66
Appendices
A Source Listings 68

A.1 Functional Arrays using totally ordered timestamps . 68
A.1.1 Structure Signatures 68
A.1.2 Structure Implementations 71
A.1.3 Implementations of Other Array Techniques . 93
A.1.4 Functions used for Performance Timing . . . 95
A.1.5 Array Test Code 97

List of Tables

1.1 Comparing Functional and Imperative Code. 3

2.1 An example monadic interface for arrays. 13
2.2 Two implementations of max 14

3.1 Semantics for array operations. 23

5.1 Specification of three implementations of array reversal. . . 50

List of Figures

1.1 A linked list in an Imperative Language. 4
1.2 A linked list in an Functional Language. 4

2.1 Understanding Trailers. 11
2.2 Understanding the Tree based Array representation. 19

3.1 How Element Histories are Represented. 26
3.2 Element Histories don't contain entries for every Timestamp. 27
3.3 The array update process. 29

4.1 Handling full persistence. 31
4.2 Partially Ordered vs. Totally Ordered Timestamps. 32
4.3 A simple timestamping scheme. 35
4.4 The principles of compressing timestamp chains. 35
4.5 The lurid details of efficient partially ordered timestamping. 37
4.6 Potential difficulties in using linear timestamps. 44
4.7 The gap problem shown in the context of the array data structure. 46

5.1 Performance for the three implementations of array reversal. 52
5.2 Worst case performance for Element Histories. 56

6.1 Implementing heaps using Functional Arrays. 60
6.2 Optimization for heaps. 64

Chapter 1

Introduction

Functional languages are often weak when it comes to supporting arrays, even though they are one of the most widely used data structures outside of the functional language community. Neglect of arrays shows up in both *Miranda*[27] and *Standard ML*[19], which have no support for arrays at all[1], and even the more recent language of *Haskell*[10] only provides functions for updating a whole array en masse, rather than allowing the update to be performed on a single array element. There are some functional programming advocates who would claim that functional languages have little need of arrays, as they have other data structuring mechanisms that are more useful, so this sort of weakness does not matter, but that sort of justification is poor at best.

This neglect of arrays is not because functional languages don't need them, but because they present some awkward problems. In this thesis we look at these problems, and at some of the partial solutions proposed in prior works, and we then go on to develop an array representation technique that provides a better solution when truly functional arrays are required.

[1] The Definition of Standard ML does not include any special features for arrays and does not include any array functions in the standard environment, but most popular implementations do provide support for non-functional array use. If one wishes to write purely functional algorithms in ML, these extensions are of little direct help.

1.1 Functional vs. Imperative Languages

Imperative languages have named storage locations, into which values can be placed, destroying any previously existing value that was held in that location. This act of *assignment* can be performed more than once on each storage location, with different values each time. Thus, different values may be read from a storage location at different points during a program's execution. For these languages, an array is an indexed collection of storage locations, usually occupying a contiguous area of computer memory. Although the collection of locations can be considered as a whole, the process of array subscripting singles out one array element to be read or destructively updated like any other storage location.

Functional languages, on the other hand, disallow assignment. Values are manipulated directly and their storage is handled implicitly. Variables in a functional language differ from their counterparts in imperative languages in that their role is purely denotational; lambda-bound or let-bound variables of functional languages simply denote a particular value, albeit one whose value may be unknown until runtime. Thus, functional languages do not make explicit use of storage locations, and so a functional array cannot be quite the same as an imperative one. It cannot be a collection of storage locations, it can only be an indexed collection of values, and must itself be a value.

Traditional imperative programming languages often blur the distinction between storage locations and values, especially when it comes to aggregate structures, so we'll clarify the difference here. Values are immutable, whereas storage locations tend to be seen as a chunk of computer memory that can be modified as required. In almost all languages, functional and non-functional alike, it is unreasonable to destructively change values themselves; an assignment such as '$1 := 2$' is disallowed, since its semantics are unclear[2]. Similarly, since the variables of functional languages do not represent storage locations, but actual values,

[2]If it were allowed, it would presumably have a deleterious effect on arithmetic, since after such an assignment it would presumably be the case that $1 \times 1 = 4$, but it is unclear what the result of $1 \div 2$ would be.

```
procedure insert(item, headptr) =
        var itemptr : list-pointer;
        while not( isnull(headptr^.next) or headptr^.next^.item > item) do
                headptr := headptr^.next;
        allocate(itemptr);
        itemptr^.item := item;
        itemptr^.next := headptr^.next;
        headptr^.next := itemptr
```

(a) An imperative implementation of insert.

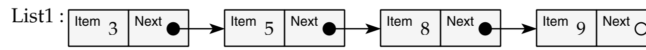
```
insert(item, list) = cons (item, list),                         if isempty list ∨ first list > item
                   = cons (first list, insert(item,rest list)), otherwise
```

(b) A functional implementation of insert.

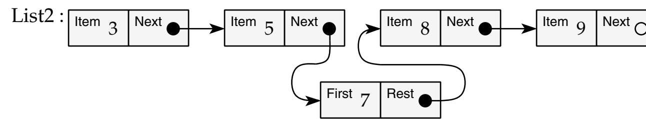
Table 1.1: Comparing Functional and Imperative Code.

they cannot have their values altered. Functional languages would not allow a definition such as '$f(x) = (x := 2)$' because evaluating '$f(3)$' would be identical to evaluating '$(3 := 2)$'. Likewise, because functional languages do not have aggregates of storage locations, only aggregates of values, one cannot redefine or modify the components of a compound value. One can only create new values, possibly derived from existing ones.

The differing styles of imperative and functional languages not only cause different coding styles to be adopted (see Table 1.1), but also result in differing data structure usage. Typically in an imperative language, we'll perform destructive updates on a data structure, modifying it as necessary. Figure 1.1 shows what this means for imperative languages in practice — namely that updates destroy prior versions of a data structure, causing us to describe such data structures as *ephemeral*[8]. In imperative languages, this isn't seen as a problem since typically a programming style is used where we never do try to refer to refer to a previous version of a data structure. We describe such algorithms as being *single threaded*[22]. The lack of storage locations and destructive update in functional languages means that previous versions of a data structure

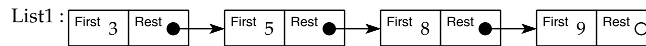


(a) A linked list in an imperative language. The list is made up of storage locations, that can have their values destructively modified.

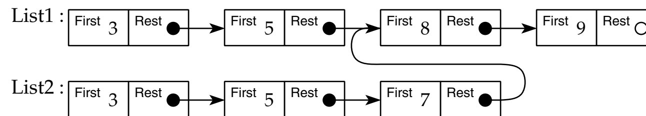


(b) Typically, insertions into a list will be done destructively. Having performed the update, we can no longer reference the old version of the list. Because updates destroy prior versions of the list, we call this kind of data structure *ephemeral*.

Figure 1.1: A linked list in an Imperative Language.



(a) A linked list in a functional language. The list is a value that can be decomposed into two components, the first element and the rest of the list, both of which are themselves values.



(b) Insertions cannot be performed destructively, thus in order to perform the update we must copy elements of the list until we reach the point where the element is to be inserted. Notice that the old version of the list can still be accessed, for this reason we call it *persistent*.

Figure 1.2: A linked list in an Functional Language.

cannot be destroyed by a program's execution[3] (see Figure 1.2) causing us to describe them as *persistent*[8].

1.2 Arrays in Functional Languages

Since functional programming languages do not make use of explicit storage locations, and imperative arrays are a vector of storage locations, something needs to be changed to allow arrays to be used in functional languages. The most obvious approach, if we wish to provide a fairly traditional interface, is to present arrays to the programmer as just another kind of value. If arrays are presented as values, which cannot be destructively updated, the array update operation must become a function that returns an entirely new array with the relevant change made. This leaves the original array on which we were performing the update both accessible and unchanged.

Such a scheme, however, while fitting semantics of functional languages, does seem rather inefficient, since copying the whole array for every update would be very costly for any array of reasonable size.

Functional language designers and implementors have come up with a variety of techniques that attempt to implement array operations efficiently, without violating the constraints for functional behaviour. Their motivations often lean towards providing good performance for some of the 'classic' array-based algorithms (such as Quicksort, various text processing algorithms, graph algorithms that use adjacency matrices and so forth[23]). Thus, a major concern is being able to run array algorithms originally written for imperative languages in a functional context without an increase in space and time complexity.

The arrays of imperative languages are, like other data structures in such languages, *ephemeral*. As we learned earlier, the original contents of locations that have been updated do not persist and thus cannot be accessed afterwards, which restricts data structure use to a *single threaded* pattern. When a non-functional algorithm that uses its data

[3] They can however cease to be referenced, and thereby be forgotten about.

structures single threadedly is re-written in a functional language, the single threaded property can be preserved. Much effort has therefore been directed towards providing good support for arrays when they are used single threadedly.

Concentrating on the ‘classic’ mode of use of arrays, and working to provide adequate performance for that case, may have helped rebut the sneers of those who scoff at the efficiency of functional languages, and allow us to re-implement some popular algorithms in a functional context, but is the price we pay for this too high? In functional languages we can write single threaded algorithms, but we are have not been required to do so, until now. Focusing only on single-threaded array access means that non-single-threaded array algorithms will be either prohibited, or incur strong runtime penalties. In cases where the semantics suggest that the original array persists after an update, we may be encouraged or even required to ignore that, and treat arrays as ephemeral structures just as we would in an imperative language. Some might argue that this doesn’t matter, and point to the paucity of array algorithms that require persistence. However, this isn’t a particularly valid argument, since few algorithm designers have had the option of efficient arrays that have efficient persistence as an option.

A *truly functional* array would allow us to use arrays non single threadedly and treat them as a persistent data structure, without huge penalties, if we so desired, or employ them as we would an ephemeral structure, accessing them single threadedly. The choice would be ours. One would hope that in such a scheme, we could still use ‘classic’ array algorithms without their having a considerably worse time or space complexity, and yet have reasonable time and space complexity for persistent usages.

1.3 Thesis Outline

In this thesis we develop and present an array representation and associated access methods that can be used to achieve the goal of providing

truly functional arrays effectively. Chapter 2 reviews many of the methods currently used to provide support for arrays in functional languages. As we have already alluded, these existing methods are far from perfect, and in this chapter, we cover some of their problems. In subsequent chapters we develop data structures and associated access methods that can be used to implement truly functional arrays effectively.

Chapter 3 presents the basic details of our method, developing *partially persistent* arrays. In Chapter 4, we extend that to provide *fully persistent* functional arrays, looking at two different schemes that can provide full persistence. We go on to examine performance issues in Chapter 5, comparing our method to some of the techniques covered in Chapter 2. Our final conclusions are presented in Chapter 7.

1.4 A Word on Notation

When describing the interfaces to functions, we will use a notation similar to the type specifications of current functional languages, such as Standard ML, Miranda and Haskell. An example function specification is shown below:

$$\textsf{allocate} \;:\; (int, int \to \alpha) \to array(\alpha)$$

The use of α indicates a type variable, $array$ is a parametric type, ':' is read as 'has type', brackets are used to indicate a tuple and '$\to$' indicates the type of the value returned by the function. Thus, in this case, the allocate function receives as parameters an integer and a function which given an integer returns an item of some unspecified but consistent type, and returns an array filled with elements of that type. In Miranda, we would have written:

```
allocate :: (num, num -> *) -> array *
```

in Standard ML, we would write:

```
allocate : int * (int -> 'a) -> 'a array
```

and in Haskell, we would write:

```
allocate  :  Num a => (a, a -> b) -> Array b
```

Finally, note that when we say $\log n$, we are referring to $\log_2 n$, not $\log_{10} n$.

Chapter 2

Review of Earlier Work on Functional Arrays

Many others have proposed solutions to the problem of providing a suitable interface for arrays in a functional languages. In this chapter, we shall present an overview of some of the more foundational work in this area, and discuss the advantages and shortcomings of their approaches. The approaches can be roughly divided into two categories, *single threaded* and *non-single threaded*[22].

2.1 Single Threaded Approaches

When an algorithm always accesses only the most recently updated version of an array, the algorithm is said to access the array *Single Threadedly*. In imperative languages, because updates are done in place, only the most recent version exists, so that is the only version that can be accessed. Thus traditional array algorithms created for imperative algorithms always access arrays single threadedly.

Since most array-based algorithms originated in imperative languages and use arrays single threadedly, an implementation technique that is optimised for, and perhaps only allows, single threaded array access will almost certainly offer good performance for the vast majority of current

array algorithms. There are, however, two general problems with the single threaded approach. The most obvious drawback is that it requires restrictions on the ways we can use arrays, as we are now preferred or required to only use arrays single threadedly. Since functional languages do not require that any other data structures be used single threadedly, it is a little restrictive to constrain our use of arrays[1]. A second problem is that by forcing our algorithms into a single threaded mold, we can impose an execution order that is not strictly necessary, and thus deny opportunities for parallelism.

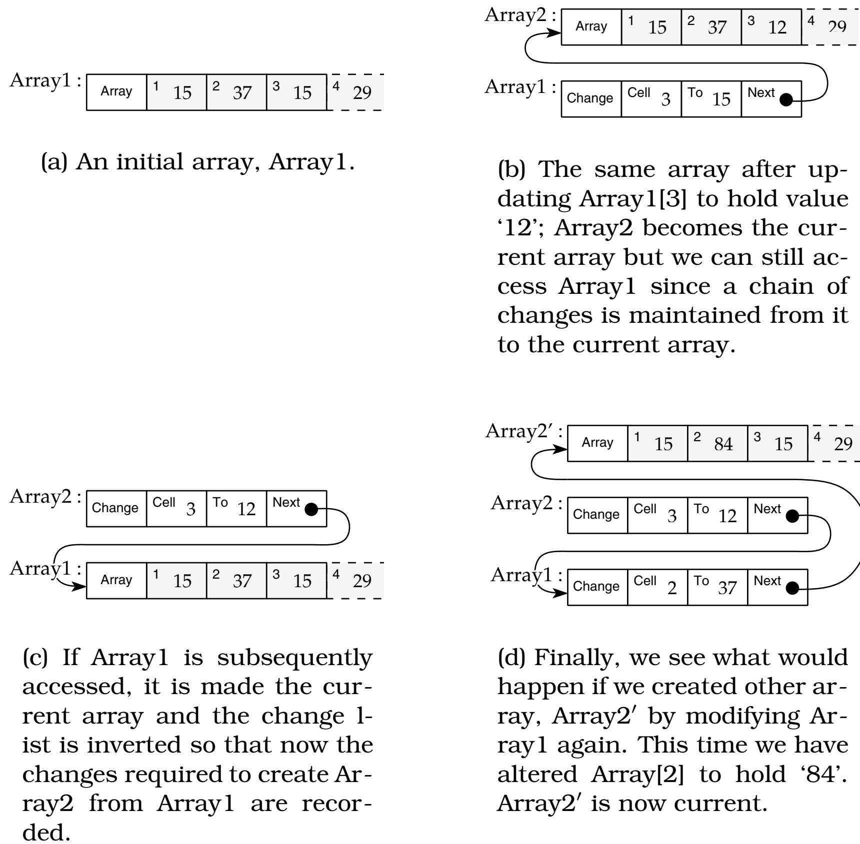
2.1.1 Trailers

Trailers[1, 14, 15] are a runtime technique developed to provide reasonably efficient support for the single threaded use of arrays. The technique also allows non-single threaded use of arrays, but such use incurs a performance penalty; the size of the penalty depends on the pattern of use, but could easily be $O(u)$ time to access an array, where u is the number of updates that have been performed (which may typically be larger than the size of the array), instead of $O(1)$ time.

Figure 2.1 shows how the data-structures used in the Trailers technique work. The method represents multiple versions of an array internally using a single 'current' array, and difference lists that trail from that array to indicate how a particular version differs from the 'current' version. Whenever a particular array version is accessed, the usual first step is to rearrange things so that the the version in question becomes the one that uses the internal array, adjusting and creating trailers for the other versions as necessary[2]. Figure 2.1(c) shows how a subsequent read access to Array1 would make it the current array.

[1] Although arrays are the only widely used data structure that tends to be viewed as requiring single threaded access, IO operations also tend to be viewed from a single threaded perspective, and so share many of the same issues as arrays.

[2] Not all implementations of trailers always rearrange things so the array being accessed becomes the 'current' array for every access. In general, knowing whether to make an array version 'current' for a particular access requires knowledge of the future access patterns the array will have, information which is rarely available or determinable.



(a) An initial array, Array1.

(b) The same array after updating Array1[3] to hold value '12'; Array2 becomes the current array but we can still access Array1 since a chain of changes is maintained from it to the current array.

(c) If Array1 is subsequently accessed, it is made the current array and the change list is inverted so that now the changes required to create Array2 from Array1 are recorded.

(d) Finally, we see what would happen if we created other array, Array2′ by modifying Array1 again. This time we have altered Array[2] to hold '84'. Array2′ is now current.

Figure 2.1: Understanding Trailers.

2.1.2 Monads

Monads[28, 30] are a very general mechanism, coming originally from the branch of mathematics known as category theory[17, 18], that can be used to incorporate non-functional features safely into a functional context. In the case of arrays, a monad can be used to bring traditional imperative arrays into a functional language, setting things up in such a way as to enforce single threaded access to arrays and thereby not violate the properties of functional programming languages. The monadic approach hides the array from the programmer (it is referenced implicitly) and severely restricts operations that can be performed on it and the way those operations can be used. One way to view an array monad is to say that it does not actually encapsulate an array in an abstract data type, it encapsulates an operation on an array. An *array operation*[3] has a parametric type, that being the type of the result that the operation will return when executed.

Table 2.1 shows an interface for a monadic implementation of an array. The array operations are shown in Table 2.1(a). Notice that each operation will be applied to an array (via the execute function which we will cover shortly), but that array reference is implicit; they do not take an array as an argument. An array operation is applied to an array by the execute function. Execute does not take an array as an argument, rather it creates a transient array to which a supplied array operation is applied. This transient array exists only during the execution of the execute function; the array itself cannot be captured by the operation being executed and returned since none of the supplied array operations provide such a feature.

To be worthwhile, the array operation passed to execute should embody a particular array algorithm. The array operations provided in the array interface only perform simple actions, but more complex operations may be created using the function compose, which allows one to create a composite array operation which will perform the one array operation, then a second. The second operation is generated by a function

[3]The term *array operation* is used here as a synonym for *array monad*. The former term is used because it is likely to create a more familiar picture for most readers.

$$\begin{aligned}
\textsf{size} &: \textit{array-op}\,(\alpha, int) \\
\textsf{read} &: int \rightarrow \textit{array-op}\,(\alpha, \alpha) \\
\textsf{update} &: (int, \alpha) \rightarrow \textit{array-op}\,(\alpha, \epsilon) \\
\textsf{return} &: \beta \rightarrow \textit{array-op}\,(\alpha, \beta)
\end{aligned}$$

(a) The functions above provide monadic operations for arrays. An $\textit{array-op}\,(\alpha, \beta)$ is an operation that can be performed on an array whose elements are of type α, with the result of the operation being of type β. When no result is returned, we use the null type ϵ.

$$\begin{aligned}
\textsf{compose} &: (\textit{array-op}\,(\alpha, \beta),\ \beta \rightarrow \textit{array-op}\,(\alpha, \gamma)) \rightarrow \textit{array-op}\,(\alpha, \gamma) \\
\textsf{execute} &: (int, int \rightarrow \alpha, \textit{array-op}\,(\alpha, \beta)) \rightarrow \beta
\end{aligned}$$

(b) These functions allow monadic operations to be combined and executed. Composing array operations is vital to creating array based algorithms, because the execute function takes a single array operation, creates an array, applies the operation, and returns the result of the operation while discarding the array that was used.

Table 2.1: An example monadic interface for arrays.

which is called with the result of the first array operation. This allows information to be passed along a chain of operations and thus allows useful algorithms to be specified. Those familiar with *continuation-passing style*[21, 6] may see some broad parallels here between that area and the way we use monadic arrays. Table 2.2 shows how a simple monad based algorithm might be written.

Monads do cleverly allow pure functional languages to make avaliable features that are usually found only in impure languages but they are not without their problems. Creating a composite array operation to perform a particular algorithm often results in complicated looking and unintuitive expressions, and algorithms that involve more than one array (such as, say, merging the contents of two arrays) cannot be implemented without specific support from the array data type (such as a two-array monad). Also, because the array is always handled implicitly,

```
max = compose size init
      where
         init arraysize       = compose (read last) (loop1 last)
                                where
                                   last = arraysize - 1
         loop1 posn best      = return best, if pos = 0
                              = compose (read prev) (loop2 prev best), otherwise
                                where
                                   prev = posn - 1
         loop2 posn best this = loop1 posn this,  if this > best
                              = loop1 posn best, otherwise
```

(a) A monad based implementation of max.

```
max array = loop last (read array last)
            where
               last           = (size array) - 1
               loop posn best = best,            if posn = 0
                              = loop prev this,  if this > best
                              = loop prev best, otherwise
                                where
                                   this  = read array pos
                                   prev = pos - 1
```

(b) A more traditional implementation of max.

Table 2.2: Two implementations of max. The first implementation uses monadic array operations, the latter uses a more traditional array interface. Both implementations follow the same algorithm, finding the size of the array, and then working from back to front, keeping track of the largest value found so far. The implementations are a little more wordy than is perhaps necessary; this has been done to facilitate the comparison of the two approaches.

it may not be embedded within any other data structures; this also prevents multi-dimensional arrays, since arrays of arrays are not possible without explicit support. Similar problems occur when monads are used to support other language features such as monadic IO or continuations, as the programmer may be forced into having to make a choice between using arrays or IO, but not both at once.

With careful design, many of these problems are not insurmountable. Language syntax can make the construction of monadic expressions more palatable, and if the various kinds of monads available are well designed and properly integrated, we can eliminate the need to make a choice between using different monads and also allow ourselves to use more than one array[4].

Monads are an excellent tool in any programmer's toolbox, having a much wider applicability than just the problem of supporting arrays. However, the problem of how to proceed when more than one monad is needed is still an open research area. Given this drawback, it seems that, current monadic approaches would appear to be less elegant than they first appear, despite their mathematical foundations.

2.1.3 Linear Types

Linear Types[29] (and also Unique Types[2]) seem to be a very promising solution to the problem of values that must be used in a single threaded way. To quote from the Philip Wadler's paper on Linear types[29]:

> Values belonging to a linear type must be used exactly once: like the world, they cannot be duplicated or destroyed. Such values require no reference counting or garbage collection, and safely admit destructive array update. Linear types extend Schmidt's notion of single threading; provide an alternative to

[4] Combining monads is an active research area[16, 13], and we have developed a technique which can enable the use of multiple monads which has not been described elsewhere. Our work in will probably appear along with our further research into functional arrays[20], or in a separate paper.

> Hudak and Bloss' update analysis; and offer a practical complement to Lafont and Holmström's elegant linear languages.

By using the type system to enforce the single use property, this approach allows algorithms to be coded in a straightforward functional style (c.f. Monads above). Linear types seem like a good solution for single-threaded array operations (as well as being a fairly general solution other interesting problems, such as file I/O), but there are some areas related to linear types where more work is needed. Specifically, some work remains to be done with regard to integrating linear types Hindley-Milner-Damas[9, 5] style polymorphism and type inference used in most popular functional languages. Also, current functional languages would require wide-ranging changes to their type systems to support linear types, and at the time of writing, with the exception of Clean[12], none of the major functional languages have been retrofitted to support them. Finally, while linear types provide a good solution to those who require an ephemeral data structure and do not restrict their applicability to arrays, it is of little use to those who are interested in persistent data structures.

2.1.4 Compile Time Analysis

An alternative to forcing the programmer to code their algorithms following some alternative style that enforces single threadedness, is for a compiler to spot single threaded use of objects, such as arrays, and thus determine when update in place can be used. This technique is usually called *Update Analysis*[3, 11], and relies on *Abstract Interpretation* to determine when update in place can safely be used.

The key problem with this approach is that for an arbitrary functional program, the task of statically determining whether it will use arrays single-threadedly is undecidable. The best any technique can do is determine whether a program appears to be single-threaded, erring on the side of caution and flagging some programs that would execute single threadedly as non single threaded. Since the programmer would no doubt

wish his single threaded code to be understood by the programming language as being single threaded, she has to know something of the tests applied and their limitations. If the programmer isn't concerned with how things work inside the compiler, we enter a new realm of unpredictabilty where a simple change to program can dramatically alter it's execution behaviour if that change happens to upset update in place optimizations being applied.

2.2 Non Single Threaded Approaches

As was alluded to earlier, single threaded approaches, while useful when converting 'classic' array algorithms into functional languages, have their limitations. Techniques have been devised that are suitable for applying to algorithms that do not fit neatly into the single threaded mold, but it is an area that appears to have received less study.

2.2.1 Monolithic Approaches

The monolithic approach to array operations does not try to find ways of performing small operations on arrays efficiently, rather it looks at ways of operating on the array as a whole. Rather than operating at the level of the array elements, the monolithic approach operates at the level of the whole array; obviously there is a need to work at the element level, but the array operation is considered to operate 'all in one go', across all the elements. A very simple monolithic array operation would be

$$\textsf{arraymap} \ : \ (\alpha \to \beta, array(\alpha)) \to array(\beta)$$

Usually, people want to perform more complex operations on the entire array, and thus most monolithic approaches use a special syntax to denote the monolithic array operations.

Having a separate syntax obviously adds complexity, not only for the user, but also for the language, which may have to perform some checks

on the correctness of the operation specified; however, monolithic specification does have some useful properties, being better suited to supporting parallelism in array access.

An intriguing thought is the combination of monadic and monolithic approaches, since one could use a monad as a means to specify the array operation to be performed.

2.2.2 Faking it, with Trees

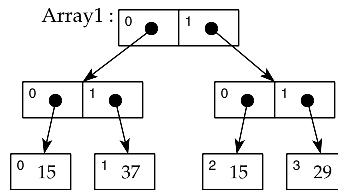
Height balanced binary trees are perhaps the most basic[5], and often used, methods for 'faking' arrays in functional languages. A reason for the popularity of this approach is that tree-based arrays can be implemented in almost any functional language. In some languages, such as Miranda[27], the programmer has little other option but to use them.

A simple tree based method would be to allocate a binary tree, and use the leaves of the tree to hold the array contents. The correct leaf of the tree for a particular integer index can be found by using the binary representation of the number to choose between taking the left or right branch. Figure 2.2(a) shows a small array represented using a binary tree[6]. Figure 2.2(b) shows how the data structure is affected by an array update operation. While it is possible to use n-ary trees; for an n-ary tree, of size k, the tree will have branch nodes of size n and height, $\log_n k$. These properties mean that having large values of n isn't usually sensible, since $\log_n k$ branch nodes of size n must be replaced for every update operation[7]. The only advantage of large n is that it speeds read access, in the limit providing $O(1)$ read access for $n = k$.

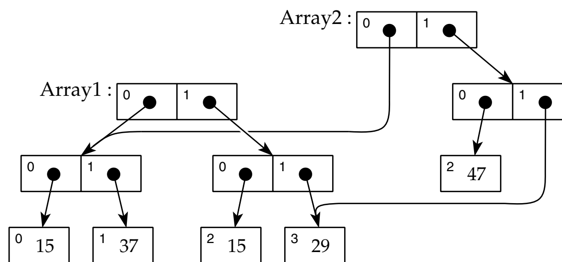
[5] A more basic approach than binary trees is to use an association list (or just a plain list) to simulate arrays. Using lists is horribly inefficient for an array of non-trivial size; the only redeeming feature of the approach being that it is incredibly simple to implement, which may explain why it is seen more often than it deserves.

[6] The representation could be optimized to eliminate an unnecessary level of indirection, by storing the contents of the leaves directly in the final branch node. This is possible because the tree has constant height, for a particular array size. This optimization isn't shown because it would unnecessarily complicate the picture.

[7] Curiously, at least to those who haven't made a study of trees, $n \log_n k$ has a minimum, independent of k, when $n = 3$ not when $n = 2$, thus using ternary trees to support arrays is slightly more efficient than using binary trees.



(a) An initial array, Array1.



(b) The same array after updating Array1[2] to hold value '47'; notice how Array2 shares much of the tree structure of Array1.

Figure 2.2: Understanding the Tree based Array representation.

The obvious problem with using trees as an array representation is that both read and update operations take a time of $O(\log_n k)$, in contrast with the $O(1)$ behaviour of the arrays found in non-functional languages. For many applications, especially those that, at least potentially, use arrays of non-trivial size this is unacceptable. Trees do, however, have the advantage of being a simple data structure, that can be used non-single-threadedly without additional complications.

2.3 Non-Functional Approaches

A final possibility is for the language to abandon the requirements for arrays to be functional, and support arrays that allow assignment. If the language is already an impure functional language and already supports assignable polymorphic references, the is no reason not to support traditional ephemeral arrays. However, functions that use such arrays could violate referential transparency, making programs that used such functions hard to reason about. While this may be acceptable in some cases, directly supported ephemeral arrays are of no use to those who need persistence.

2.4 Conclusion

None of the methods we have covered have provided good support for persistent arrays, and many are far from perfect for implementing 'classic' array algorithms, sometimes requiring cumbersome code sequences, or disallowing simple and useful operations. Some methods work successfully for some cases, but minor code changes can cause a drop to a terrible worst case performance. Finally, if the technique demands that program code access arrays in a single threaded[22] manner, that can impose an unnecessary serialisation of access that denies the potential for parallel execution.

In subsequent chapters, we shall present techniques that can provide persistent and thus truly functional arrays without increased space or

time complexity for ‘classic’ array algorithms.

Chapter 3

The Essence of Our Method

In the preceeding chapter, we looked at previously available options for those who wish to use arrays in functional languages. In this chapter we begin the presentation of our alternative to these techniques, laying down the foundations for our method. Here we will develop *partially persistent* functional arrays, which we will extend in subsequent chapters to provide fully persistent arrays. In other words, we will describe our technique, while imposing the restriction that only the most recent version of an array may be updated, but all array versions may be read. Later we will remove this restriction.

3.1 Basic Operations for Arrays

As we mentioned in the Chapters 1 and 2, there is more than one way of integrating arrays into functional languages. Our goal is not provide a special mechanism to support them, but rather to make them just another data type to be used as the programmer sees fit. Our interface is therefore based around an abstract array data type with a few simple access functions, shown below:

$$\begin{aligned} \textsf{create} &: (int, int \to \alpha) \to array(\alpha) \\ \textsf{subscript} &: (array(\alpha), int) \to \alpha \\ \textsf{update} &: (array(\alpha), int, \alpha) \to array(\alpha) \end{aligned}$$

subscript (create (s, f), i)	=	f(i),	if $0 \leq i < s$
	=	$\perp$,	otherwise
subscript (update (a, j, v), i)	=	v,	if $(i = j) \wedge ok$
	=	subscript (a, i),	if $(i \neq j) \wedge ok$
	=	$\perp$,	otherwise
		where	
		ok = inrange (a,i)	
		inrange (create(s, f), i) = $0 \leq i < s$	
		inrange (update (a, j, v), i) = inrange (a, i)	

Table 3.1: Semantics for array operations.

These functions for the most part mirror the array operations present in imperative languages, the only exception being array update. As mentioned in Chapter 1, the semantics of a functional update operation require that the result of an update be a new array value, while the original array value supplied to the update function remains unchanged and accessible.

The interface shown above there is no fundamental problem in applying all the techniques developed in this thesis to dynamically sized arrays.

Axioms for functional arrays are shown in Table 3.1.

3.2 Origins of Our Method

Our approach to the functional array problem came originally from an unrelated programming effort in which functional arrays were a prerequisite (a functional implementation of a unification based algorithm). Having written an implementation, we then realized that our approach had not been presented elsewhere, and so began looking into the subject more closely. In researching the matter, we found parallels between our work and pre-existing work by Driscoll et al.[8], and incorporated a few refinements based on their work. Given that, it is worth noting how our work differs from and builds upon that of Driscoll et al.[8].

Driscoll et al.'s work[8] is based on providing support for persistent

data structures in imperative languages, not functional languages. While this may seem like a trivial difference, we will discover in Chapter 6, when we examine heaps, that one cannot automatically assume that it is possible to transport these features into a functional language. On a similar note, in the imperative context of Driscoll et al.'s work, timestamps are a visible part of the system, whereas in a functional context it is necessary to keep their existence hidden.

Another issue with Driscoll et al.'s work on persistent data structures[8] is that their techniques relate specifically to linked data structures (or to be more precise, linked data structures of bounded in-degree). This has two ramifications, the first is that their work has been largely ignored by the functional language community, since as we learned in Chapter 1, linked data structures are already persistent in functional languages and so Driscoll et al.'s techniques for providing persistence aren't seen as being applicable. The second ramification is that arrays are not discussed by Driscoll et al., since arrays are not linked data structures. It is perhaps not surprising, then, that much of their work cannot be applied to arrays.

In examining the problem, we have developed techniques that *can* be applied to arrays, albeit with worse time complexity than Driscoll et al. manage for linked data structures of bounded in-degree[8]. We should also make it clear that we have not just applied a subset of Driscoll et al. to the area of functional arrays, but we have rather gone beyond what they present making refinements — for example, our technique guarantees $O(1)$ performance for single threaded array use, while their techniques that most closely approximate ours do not.

Finally, we also take a broader look at the issue of generating timestamps (see Chapter 4), and cover the merits and demerits of partially ordered timestamping, presenting an algorithm for generating such timestamps.

3.3 Data Structure Overview

Our technique uses timestamps along with 'element histories' to store multiple arrays, which we shall call *array values*, in a single data structure. Several array values are represented internally using a single master array, each element of which is a 'history' that holds all values that are associated with that array element. Every array value has its own unique timestamp, which is used to retrieve the values of its particular elements from the shared master array.

3.4 Element Histories

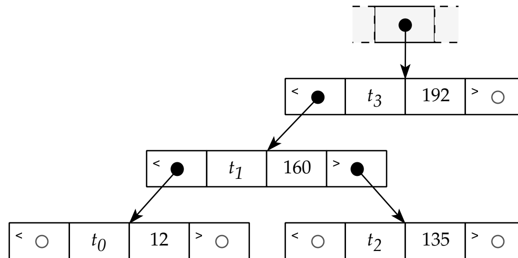
As we outlined above, each element of the array may contain more than one value. Specifically, each element of the array is a data structure which maps array timestamps to values. Figure 3.1(a) and Figure 3.1(b) show two different structures that could be used to store such information, portraying a stack and a tree respectively. Factors influencing our choice of data structure are covered later, and for much of our discussion, the exact data structure used is irrelevant. In such cases, we will use the abstract diagrammatic representation shown in Figure 3.1(c).

A key feature of the representation of element histories is that they will not, in all likelihood, contain values for every timestamp in existence. They only hold a history of changes. The only time a new timestamp/value pair is added to the element history is at a 'cusp', when that particular element is changed. Values for timestamps not held explicitly in the element history can be determined implicitly.

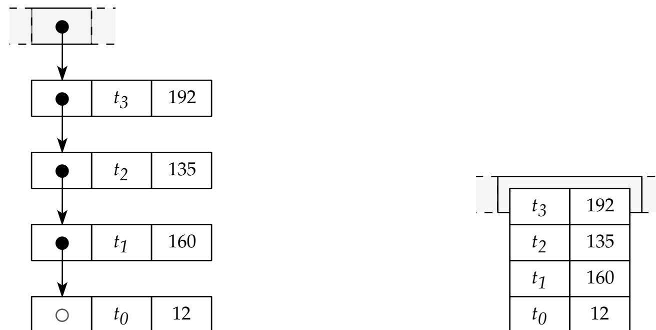
If we consider the element history to be a set of timestamp/value pairs, H, and suppose that there is some ordering relation on timestamps, then the value corresponding to some desired timestamp, t_d, can by found by finding the closest timestamp in the set, t_c, where

$$t_c = \max\{t \mid (t, v) \in H \land t \le t_d\}$$

and retrieving the value corresponding to that timestamp. In other words,



(a) A tree-based element history, containing four different values, one for each of four possible times. Initially, at time t_0 the element held the value 12, and even though it has been accessed and updated since then to hold first 160, then 135, and then to 192, it is still possible to retrieve the value it had at time t_0.



(b) Here we have the same history entries as above, but stored in a stack. The factors to consider when choosing a particular representation are dealt with in Chapter 4.

(c) Because there is more than one way to represent element histories, and because often the exact representation is less interesting than what is actually being stored, we will usually represent them as shown here. In this way, we convey what is being stored, without actually specifying the representation used.

Figure 3.1: How Element Histories are Represented.

t_4	23
t_3	47
t_0	12

(a) In this element history, cusps exist at times t_0, t_3 and t_4. Entries are omitted for timestamps t_1, t_2 and t_5, because no changes were made to this particular element at those times.

t_5	23
t_4	23
t_3	47
t_2	12
t_1	12
t_0	12

(b) Here we show not only the actual values stored in the element history, but also the values that are inferred (shown in grey).

Figure 3.2: Element Histories don't contain entries for every Timestamp.

t_c is the greatest timestamp less than or equal to t_d. This is expressed more clearly in Figure 3.2.

3.5 Array Values and the Master Array

As we outlined earlier, the master array, which is used to store multiple array versions, is simply an array of element histories. Each individual array value represents a particular version of the array, and consists of a reference to the shared master array, and a unique timestamp.

Subscripting a particular array value is done by simply retrieving the element history for the desired array element, and then retrieving the value corresponding to the array value's timestamp. Array update is achieved by creating a new array value, with a timestamp obtained by incrementing the timestamp of the array we are updating[1], and adding new entries corresponding to the new timestamp to the element history(s) of the element(s) being updated.

Figure 3.3 shows how multiple array versions are supported, showing

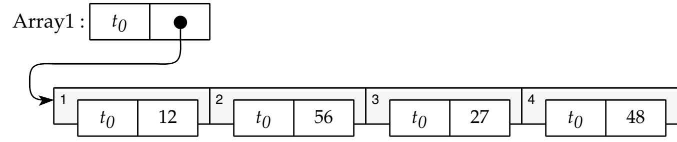
[1] Remember that we are assuming, for the time being, that only the most recent version of the array will be updated.

a sequence of four updates on an array, leading to the creation of five different array values. Each element history is shown with both the values it contains, and the values that are inferred (distinguished by being shown in grey).

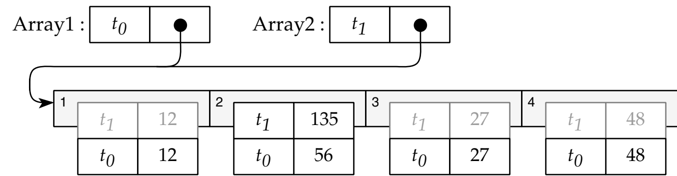
3.6 Conclusion

The techniques we have presented so far can only provide partial persistence, since we have assumed that, while every array value can be read, only the most recently changed version of the array will be updated. This would be sufficient for many applications, but this restriction means that our arrays so far could not be described as functional. In the following chapter, we will examine the ways in which this scheme can be extended to provide full persistence.

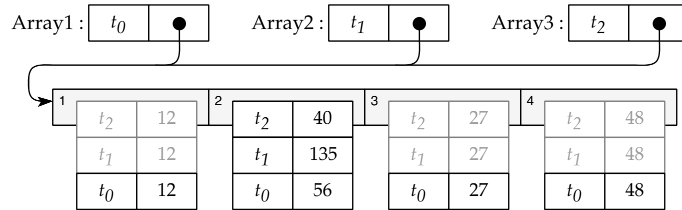
Before moving on however, we should take a final look at what we have accomplished so far. In our current scheme, update operations and read accesses on the most recently created array value take $O(1)$ time, if a stack or splay tree[24] is used to store element history entries. Reads of array elements that have been updated u times since the time of the current array take $O(\log u)$ time, if the element history is arranged as a search tree (such as a splay tree). Thus, single threaded access takes constant time, and while partially persistent access doesn't take constant time, it is still faster than the other techniques available. Users of imperative languages should not be able to sneer at this kind of functional arrays because the only access that does not take constant time is an access not usually available to them. There is, however, a larger constant factor in the access time, but as the author's subsequent work shows, even this can be minimised by certain optimizations [20].



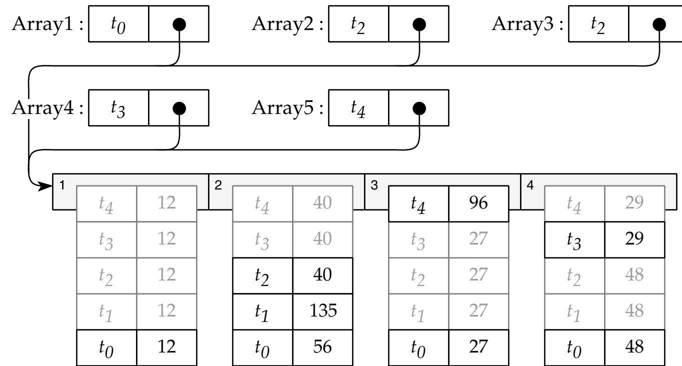
(a) An array data structure supporting a single array value.



(b) The array data structure after an update has been performed on the second element of the array. Thus, the master array now holds two array values.



(c) Another update has been performed on the second element array, so it now holds three array values.



(d) The array after two more updates, this time on the fourth element of the array and then the third element.

Figure 3.3: The array update process.

Chapter 4

Providing Full Persistence

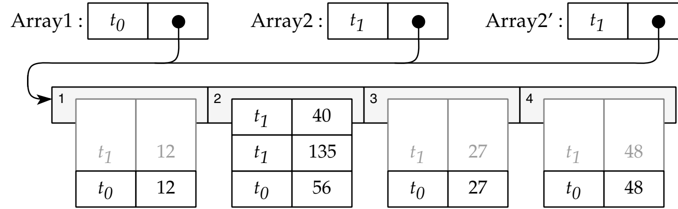
The methods we have developed so far provide us with *Partial Persistence*[8], whereby the original array values persist after an update, but those values cannot be updated again. In this chapter we shall look at how to extend our array algorithms and data structures to provide the capacity for *Full Persistence*[8].

4.1 The Problem

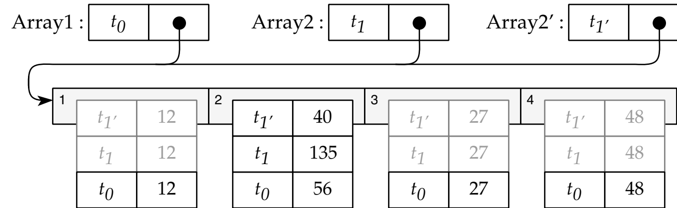
The reason the array implementation we presented in Chapter 3 cannot provide full persistence is that the simple timestamping method we used isn't up to the job. Figure 4.1(a) shows what would happen if we did try to update an array that had already been updated once. Our timestamping scheme generates the timestamp for the new array by simply incrementing the timestamp of the original array value. Thus, if we perform multiple updates, each new array will receive the same timestamp. This is clearly a problem. Possible solutions to this problem are shown in Figures 4.1(b) and 4.1(c), and discussed below.

4.2 Partially and Totally Ordered Timestamping

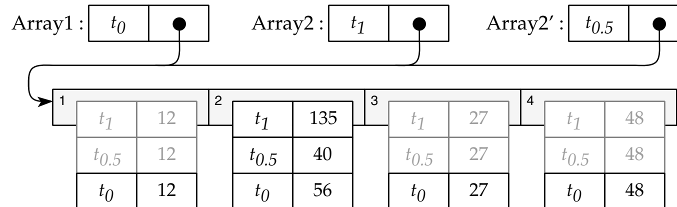
There are two methods we could use to provide fully persistent arrays, and so we shall briefly described both, before covering each in more detail.



(a) The approach that worked for the partially persistent case is not sufficient when full persistence is required. Array2 and Array2' have both been given the same timestamp, resulting in the wrong value being returned if the updated element of Array2 is accessed.

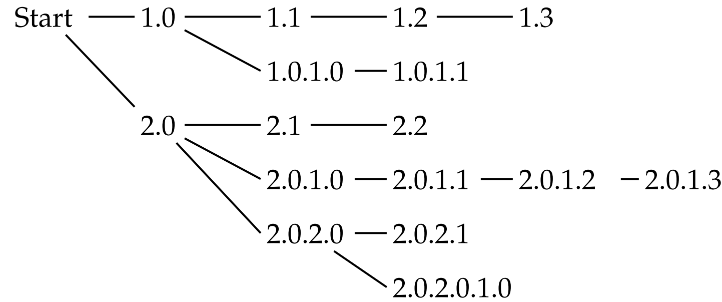


(b) Giving time-stamps a partial order allows fully persistent updates to work correctly. Under this scheme, t_1 and $t_{1'}$ are distinct, and both 'after' t_0 but are incomparable with each other.

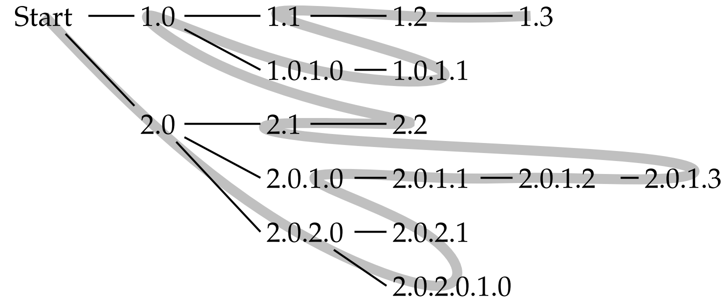


(c) Totally ordered timestamps can also support fully persistent updates, provided sufficient care is taken. In this case, Array2' gets allocated a timestamp that is after the timestamp of Array1 and before the timestamp of Array2.

Figure 4.1: Handling full persistence. In the situation shown above, Array1 has been updated twice; Array2 is the first array derived from an update on Array1 and Array2' is the second.



(a) The partially ordered timestamping scheme used by RCS.



(b) It is quite straightforward lay a total order over the partial order of RCS. In integrating different and incomparable branches into a single time line, we are seeing things somewhat differently, however, for example previously we saw no timestamps between 1.0 and 1.1 whereas in the totally ordered view, several timestamps lie between them.

Figure 4.2: Partially Ordered vs. Totally Ordered Timestamps.

4.2.1 Partially Ordered Timestamping

The first option shown in Figure 4.1(b) gives timestamps a *Partial Order*, whereby the timestamp for an array resulting from an update still comes from a kind of 'increment' operation, in that the resulting timestamp comes after that of the original array, but where multiple updates to the same array value generate different and incomparable timestamps. In other words, we allow time to branch, forming 'alternate timelines' when necessary.

Some readers will have already encountered partially ordered time-stamping schemes, since they typically appear in systems for source code control (such as *RCS*[25]). Looking at the timestamping scheme used by RCS is helpful since we can look at it and gain a general understanding of this kind of timestamping without worrying too much about implementation details (which we present later).

Figure 4.1(b) shows a situation where we initially had an array value with timestamp t_0, which was updated once, with the resulting array value receiving timestamp t_1. Then, another update was performed on the original array requiring another timestamp after t_0 — the timestamp $t_{1'}$. Under RCS we would allocate timestamps as follows: t_0 would be 1.0, t_1 would be 1.1, and $t_{1'}$ would start a new branch and thus be $1.0.1.0$. The first timestamp generated after t_0 is done via a simple increment, but subsequent timestamps need to form a 'branch'. A more complex example how of branching occurs, which includes the above example, is shown in Figure 4.2(a).

4.2.2 Totally Ordered Timestamping

In totally ordered timestamping, shown in Figure 4.1(c), we recognize that while updates can generate arrays which are unrelated to each other, a-part from a common ancestor, we do not need necessarily to transport this relationship into our timestamping scheme. Instead we impose a total order on top of this partial order, to gain a single timeline. Figure 4.2(b) gives an example of how we could take branched time, characterized by the example from RCS we used earlier, and impose a total order on it.

In this flattened model, we do not need to form branches, we only need to be able to allocate a new timestamp 'just after' an existing one (i.e., after that timestamp, but before any others that are after it). One way to envision such a scheme would be to imaging timestamps represented by real numbers. If an array value had a timestamp with value 17, a derived array value might have value 18. If another on the original array was performed, the resulting array would get a timestamp between 17 and 18,

say 17.5.

Having introduced both techniques, we shall now cover each in more detail, presenting implementation techniques, discussing their properties and covering problems that arise in the use of each scheme.

4.3 Algorithms for Partially Ordered Timestamps

In this section, we develop a straightforward mechanism for generation partially ordered timestamps. We shall begin by specifying the interface for such a timestamping scheme:

$$\begin{aligned} \textsf{create} &: \epsilon \to \textit{timestamp} \\ \textsf{increment} &: \textit{timestamp} \to \textit{timestamp} \\ \textsf{order} &: (\textit{timestamp}, \textit{timestamp}) \to \{\text{before}, \text{equal}, \text{after}, \text{incomparable}\} \end{aligned}$$

In this scheme, create creates an initial timestamp (create takes a null argument, represented by ϵ), with further timestamps being created by insert, which creates a new timestamp that lies after the one passed. Finally, the order function exists to compare two timestamps. If two timestamps lie on different time lines, an order comparison will say that they are incomparable.

We shall begin by introducing a very simple partially ordered timestamping scheme, which we will later refine.

4.3.1 A Simple Timestamping Algorithm

Our simple partially ordered timestamping scheme is based around object identity and a simple ‘cons’ing operation (see Figure 4.3). The ‘increment’ operation just adds a cell to the front of the chain and returns the new top of the chain. The ‘order’ function compares two timestamps by searching down each timestamp chain to find whether both lie on the same time line and if so, where they lie in relation to each other. Thus t_2 in our example is before both t_3 and $t_{3'}$ but t_3 nor $t_{3'}$ are both incomparable with each other.

This scheme is certainly simple and does provide fast allocation of new timestamps. But the length of the timestamp chains grows with every new

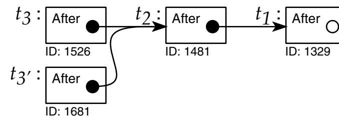
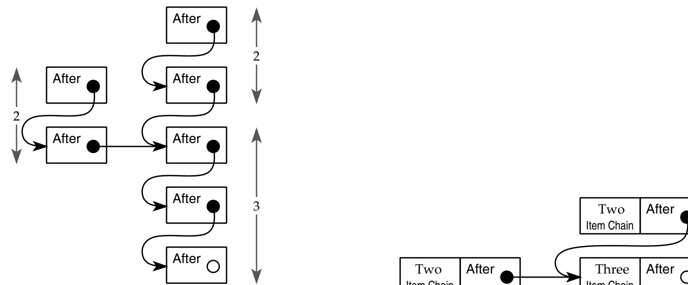


Figure 4.3: A simple timestamping scheme.



(a) A timestamp chain, without compression.

(b) A timestamp chain, with compression.

Figure 4.4: The principles of compressing timestamp chains.

timestamp, which seems something of a problem. Worse perhaps is the fact that the 'order' function cannot be implemented efficiently, since the timestamp chains can grow quite long.

4.3.2 Improving the Algorithm

If we assume that long unbranched chains of timestamps will be fairly common, we can improve our timestamping algorithm to take account of this and offer improved performance. Figure 4.4 shows a slightly simplified branching chain of timestamps and how we can reduce the chains into single cells that record how long the chain is. In expressing the basic idea, we have omitted a few details, but in essence, this method is the basis of our timestamping scheme.

Having presented the basic ideas underlying this refinement, we shall now proceed to fill in the details. Timestamps have two components, one which is unique to each timestamp, and another that is shared by all

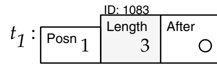
timestamps in a chain. The shared component keeps track of where this chain fits onto other timestamp chains, and also contains a chain length counter that is used to spot branching. The unshared part contains an integer indicating where the timestamp is in the chain, if this integer is equal to the chain length, then we know that no other timestamps exist after this one this chain.

If we are asked to generate a new timestamp that lies after the last entry in a chain, all we need to do is increment the chain length counter for the chain, and return a new timestamp whose position field is equal to the chain length. If, however, we are asked to generate a timestamp after a timestamp that is not at the end of a chain, we know that at least one timestamp lies after it, and so we must form a new branch. So in this case need to create a new chain, and link it to that timestamp. Figure 4.5 gives an example of an update sequence involving both simple additions to a timestamp chain and the creation of new branches.

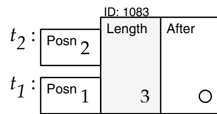
The chief advantage to this more complex scheme is that we reduce the length of the chains we have to search. For example, in the Figure 4.5 we can determine very quickly that both t_1 and t_2 are earlier than t_3, because they all lie on the same chain and differ only by their position. However, to see that t'_2 is after t_1, we need to go to a little more work, and we need to trace back along its branch in an analogous way to the very basic scheme we began with. This unfortunately does give partially ordered timestamping a rather poor worse case time complexity, even though practical applications may rarely cause branching to occur.

4.3.3 Conclusions

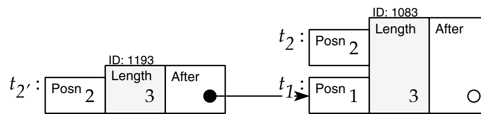
The poor worst case performance of the partially ordered approach is a clear drawback, and it is not the only problem that arises with this technique. In the context of functional arrays, we run across another problem that comes from the very nature of having partially ordered timestamps. If timestamps have a partial order we cannot use them in an order based data structure. This means that we cannot be able to use a tree to store



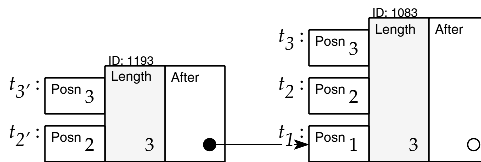
(a) A single (initial) timestamp, t_1. Notice that there are two parts to the structure, a 'Position' field which is unique to this timestamp, and a common part which will be shared by all timestamps in this chain. This latter part consists of 'Chain Length' counter and an (empty) 'After' pointer.



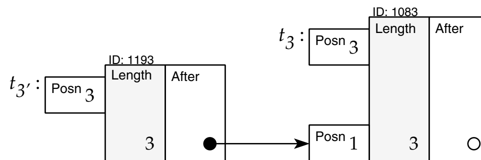
(b) Two timestamps; t_2 is the result of incrementing t_1. Notice that while t_1 and t_2 both reference the same common data structure, they do not explicitly reference each other. (Note that we use a pointer to link the unshared part of a timestamp to the shared part.)



(c) Three timestamps; both t_2 and $t_{2'}$ result from asking for a timestamp after t_1. Notice that the 'After' field in $t_{2'}$ references t_1. While t_2 was the result of a simple increment, $t_{2'}$ was the result of a branching increment.



(d) Five timestamps; t_3 is after t_2 and $t_{3'}$ is after $t_{2'}$.



(e) Three timestamps; t_1, t_2 and $t_{2'}$ have ceased to be referenced by the program.

Figure 4.5: The lurid details of efficient partially ordered timestamping.

element histories, instead we must to use some other data structure, such as a stack. Not being able to use order to speed the search for an entry in an element history reduces further the worst case time complexity of methods based on partially ordered timestamps.

Things aren't completely bleak, however. Despite it poor worst case performance, this may be a viable technique for algorithms that mostly perform single threaded accesses, since single threaded accesses will always create timestamps that lie on a single chain and the most recently version of an element will always be on the top of the stack of element histories.

So, while noting that applications that are mostly single threaded there may find work efficiently with partially ordered timestamps, we will turn our attention to totally ordered timestamping and see what advantages it holds.

4.4 Algorithms for Totally Ordered Timestamps

The problem of generating timestamps that follow a total order can also be viewed as the problem of 'maintaining order in a list'. This latter problem is one of maintaining a list through a sequence of insert and delete operations, while answering order queries (to determine which of two entries comes first in the list). An interface for ordered list maintenance functions is shown below:

$$\begin{aligned}
\textsf{create} &: \alpha \to \mathit{ordered}(\alpha) \\
\textsf{insert} &: (\mathit{ordered}(\alpha), \alpha) \to \mathit{ordered}(\alpha) \\
\textsf{order} &: (\mathit{ordered}(\alpha), \mathit{ordered}(\alpha)) \to \{\text{before}, \text{equal}, \text{after}\} \\
\textsf{succ} &: \mathit{ordered}(\alpha) \to \mathit{ordered}(\alpha) \\
\textsf{value} &: \mathit{ordered}(\alpha) \to \alpha
\end{aligned}$$

Note that the parametric type $\mathit{ordered}(\alpha)$ refers to the point in an ordered list where a particular element (of type α) is stored.

The order maintenance problem presumes that we will be storing something in this ordered list, but if we don't actually store any data in it, the problem can be viewed as a timestamp generation problem. We can therefore

apply some simplifications and produce and interface for timestamping operations, shown below[1]:

$$\begin{array}{rcl} \textit{timestamp} & = & \textit{ordered}(\epsilon) \\ \textsf{create} & : & \epsilon \rightarrow \textit{timestamp} \\ \textsf{insert} & : & \textit{timestamp} \rightarrow \textit{timestamp} \\ \textsf{order} & : & (\textit{timestamp}, \textit{timestamp}) \rightarrow \{\text{before}, \text{equal}, \text{after}\} \\ \textsf{succ} & : & \textit{timestamp} \rightarrow \textit{timestamp} \end{array}$$

Although the techniques for addressing the order maintenance problem with $O(1)$ time taken for insert, delete, order have been published elsewhere[7, 26], it is sensible for us to outline the algorithm here, since the generation and comparison of timestamps is a fundamental part of our method.

Our particular implementation follows the first algorithm presented by Dietz and Sleator[7], which is a relatively straightforward algorithm. This algorithm, which we will describe below, takes $O(1)$ amortized time for insert and $O(1)$ real time for order, succ and delete. Dietz and Sleator's other algorithm provides $O(1)$ real time for all operations, but at the cost of some algorithmic complexity. There is no fundamental problem with the use of this second algorithm, however, we have merely chosen to use the first one for the sake of simplicity.

Although the algorithm is presented elsewhere, our presentation of it may prove of some interest, since we present it from a slightly different perspective, and reveal some properties that aren't obvious in the original presentation[2].

4.4.1 A Simple Timestamping Algorithm

The algorithm we used maintains a circularly linked list. Each node in the list has an integer label, which is occasionally revised. We also impose

[1] Note that ϵ represents a null type.

[2] 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 '$\bmod M$' to be prematurely applied) if M is chosen, as suggested, to exactly fit the machine word size.

a fixed but arbitrary upper limit, N, on the number of elements that will be stored in the list. Typically, N might be chosen to fit the size of a machine word, or 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 have 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 set M to be 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.

Initially, the list holds a single element, which is never deleted. This node is special in that its label (which can be assigned an arbitrary value) is used when performing order comparisons. We shall call this element the base.

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 (corresponding to succ function we specified in the timestamping interface). 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:

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

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

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. We will suppose that we wish to place a new element, n, 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:

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

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 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.

4.4.2 Refining the Algorithm

The algorithm, as presented so far, takes $O(\log n)$ amortized time to perform an insertion[7], but there is a simple extension of the algorithm which allows it to take $O(1)$ amortized time per insertion[26, 7]. 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 list, $c(e)$, and a node in the upper 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 $n/\log n$ 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 n, 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 n_c after $c(e)$, and perform the assignments $c(n) \leftarrow n_c$ and $p(n) \leftarrow p(e)$. If 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 second sublist is given a new parent, n_p that is inserted in the upper order list immediately after e_p.

These techniques are used for insertions until the number of nodes in the overall order list, n, 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, the arena size, M, can be slightly lower. Recall that previously we imposed the condition $M > N^2$. Now 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 wish 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)[3].

4.4.3 Conclusions

Following this scheme then, we can implement efficient ordered lists and by simple derivation, a quick and effective scheme for totally ordered timestamps. Their $O(1)$ performance makes them a very attractive method, and for this reason we shall adopt them over partially ordered timestamps. The rather artificial flattened timeline they provide does itself raise some issues that we need to deal with — we address those issues in the next section.

4.5 Correctly Implementing Element Histories

To properly understand what we need from our timestamping scheme we need to briefly recap some of the things we learned in Chapter 3. We implement arrays through the use of timestamps and multiple *Element Histories*. An interface for element histories is shown below:

$$\begin{aligned}
\textsf{create} &: \epsilon \to (\alpha \to \mathit{ordered}(\alpha), \mathit{timestamp}) \\
\textsf{record} &: (\mathit{ordered}(\alpha), \mathit{timestamp}, \alpha) \to (\mathit{ordered}(\alpha), \mathit{timestamp}) \\
\textsf{locate} &: (\mathit{ordered}(\alpha), \mathit{timestamp}) \to \alpha
\end{aligned}$$

[3]The exact number of items allowed in this case is 1376234. Our ML implementation sets M to 2^{29}, and allows more than 2^{18} items in the list (440239 to be precise). If this limit is exceeded, the algorithms will continue to work, but insert is no longer guaranteed to take $O(1)$ amortized time. For functional arrays, we believe this limit to be entirely reasonable, since this only places a largish limit on the number of array versions that may exist concurrently and not the size of the arrays used.

(a) An element history with gaps. The entry for t_0 is used to provide values for t_1, t_2 and t_3 which are not present in the history.

(b) Naively inserting a value for $t_{1.5}$ inside a gap, upsets the values of t_2 and t_3. Note that t_1 is unaffected.

(c) To ensure correct behaviour, we need to also insert an entry corresponding to t_2 (which was previously just inferred using the t_0 entry, before inserting $t_{1.5}$.

Figure 4.6: Potential difficulties in using linear timestamps.

In the totally ordered timestamping scheme, all timestamps belong to a single linear 'time-line', which serves all element histories and thus all the instances of a particular array. Since the same timestamps span all the element histories of a persistent array data structure, we need to consider how this affects things. So far we have only ever looked at one element history of the array at a time when considering updates. When we move out to look at the bigger picture, it becomes clear that there is a problem with the linear timestamping method.

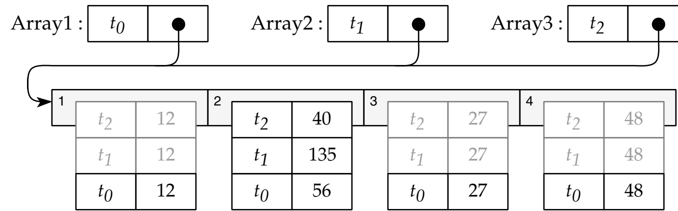
When updates take place over more than one element history there will be 'gaps' in the element histories. The gaps occur because an update adds a new entry with a new timestamp to an element history, but no entries for that timestamp are added to any of the other element histories. When reading an element history to retrieve the entry corresponding to a particular timestamp, if that timestamp is missing, we retrieve the nearest one, that being the latest timestamp that is earlier than the one

we were looking for. No problems occur with this method when all we require is partial persistence, and this is exactly the method we presented in the previous chapter.

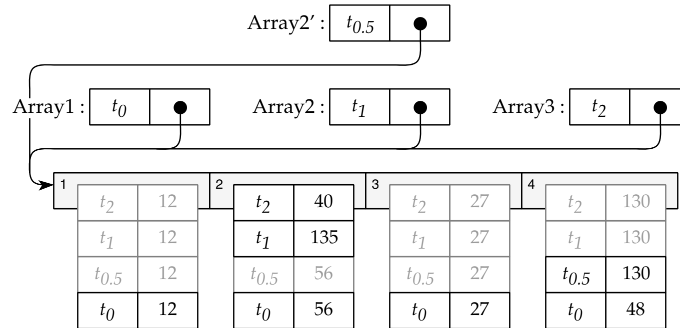
But these missing timestamps do present a problem when implementing full persistence using linear time. The problem occurs when the element history lacks a value for the timestamp we are looking up, and we return the nearest one (i.e., the one corresponding to the most recent timestamp earlier than the one we are seeking). In this case, the *wrong* value can be returned. This can happen because entries can be inserted in the element history in such a way that they reside inside a preexisting gap. Figure 4.6 shows how such an error can occur, and how it can be prevented.

We cannot prevent new element history 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 returned for timestamps that lie after the one we've added. If we will be adding an entry to an element history for timestamp t, we need to check whether the entry for the nearest timestamp after t, $\mathsf{succ}(t)$, exists and if so, whether it is represented in the element history. If $\mathsf{succ}(t)$ does exist and is not represented in the history, we need to add an entry for $\mathsf{succ}(t)$, giving it the value that would previously have been inferred by retrieving the value for the entry with the latest timestamp earlier than t. (If $\mathsf{succ}(t)$ does not exist, no timestamps lie beyond the one we're adding and so we do not need to take any additional steps.)

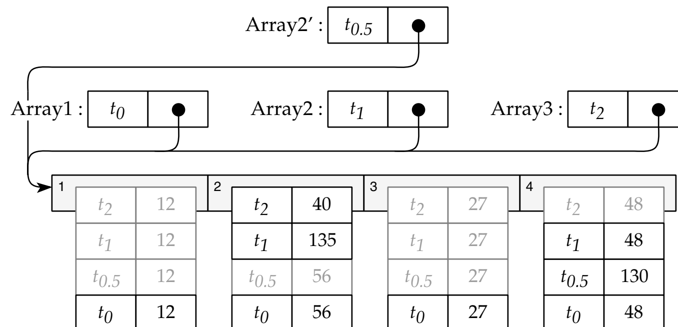
Figure 4.7 also describes this same issue, but with the added context of the complete array data structure. A useful exercise for the reader might be to examine Figure 4.7(b) and investigate what happens for the same sequence of updates when using partially ordered time stamps, to see why the same problem does not occur in that case.



(a) This figure shows two of the element histories of a persistent array data structure after two single threaded updates, both of which have taken place on the second element of the array. The first update, on Array1, created Array2 which has timestamp t_1 and the second, on Array2, created Array3 with timestamp t_2.

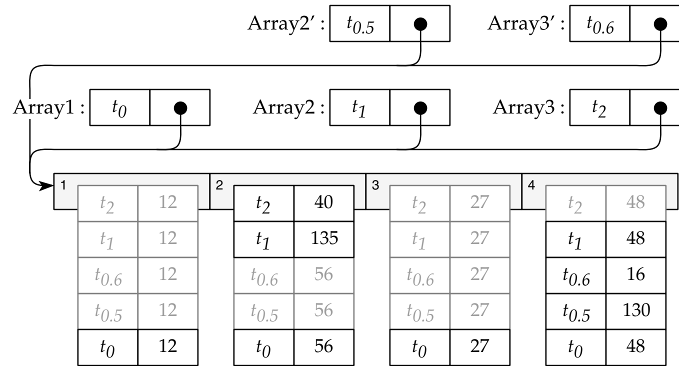


(b) A naively applied fully persistent update causes problems however. A second update on Array1, this time on the fourth element, may look innocuous, but in fact it has had an unwanted effect on the values of Array2[4] and Array3[4] which should not have been affected. Their value should be the one associated with t_0, 48, but now 130 will be erroneously returned.



(c) The problems that occurred in (b) can be resolved by also adding a history entry for t_1 when adding the entry for $t_{0.5}$.

Figure 4.7: *(continued over...)*



(d) We do not need to add any additional entries when adding an entry for $t_{0.6}$ in element four's history because there is already an entry for t_1. Also, we don't have to do any extra work if we then add an entry for t_3.

Figure 4.7: The gap problem shown in the context of the array data structure.

4.6 Conclusion

In this chapter we have seen how to extend the partially persistent data structure we developed in Chapter 3. The extensions are, in fact, reasonably straightforward. The largest changes have occurred in the way we generate timestamps, since we have gone from performing a simple increment, to a more complex scheme. We have presented two time-stamping schemes, both of which can address the issue. The first is reasonably straightforward to implement, but with poor worst case performance, while the second is a little more complex, but offers better time and space complexity, and is therefore more suitable for general use.

In the following chapter we look beyond complexity measures and examine the actually performance of functional arrays using totally ordered time.

Chapter 5

Performance Issues

In Chapter 4, we developed fully persistent arrays where the accesses to the most recently read or written value of an array element takes constant time, and where access to an array element which has been updated u_e times takes $O(\log u_e)$ time. In this chapter we show what this means in practice, by examining the performance of our techniques and comparing them to the performance of the other popular techniques that can be used to support functional arrays.

5.1 The Issue of Performance

While computational complexity measures are useful, in practice other issues come into play. Knowing that an array update or access can be preformed in constant time is useful, but it's also useful to know the approximate size of the constant. For example, one might assume that an $O(n)$ algorithm is obviously better than an $O(n \log n)$ one but in practice it may not be — if it the first algorithm was thirty times slower for $n = 1$, it would only surpass the second algorithm for n over one billion. Thus we need to measure the actual performance of our method, and compare it to the performance of other techniques, comparing not only computational complexity, but actual execution times.

Measuring performance is, however, a thorny subject and a research field in its own right. Measures of performance are vulnerable to be

attacked as either being too simplistic, too contrived, or not being representative. Often performance is measured using some traditional and recognized benchmark, but for functional arrays, no such benchmark exists. This isn't surprising, since the vast majority of array algorithms have been developed for imperative languages and ephemeral arrays, but it does mean that we have to develop our own performance measures.

We have designed our performance tests to highlight the strengths and weaknesses of each array implementation technique. One of the key issues we examine is performance for both conventional single-threaded array access and also for simple and complex non-single-threaded access. We consider performance for three implementation techniques besides our technique of *Element Histories*, namely, *Trailers*[1, 14, 15], *Binary Trees* and *Naive Copying* (in the latter technique, we just copy the whole array for every update). The actual tests used and the results obtained are described below.

5.2 A Simple Test, Reversing an Array

We use array reversal as a performance measure both because it is a readily understandable concept, and because it has a good deal in common with a variety of other operations on arrays. It is also interesting because there is more than one 'obvious' algorithm — we consider three algorithms, each with slightly different properties. The three implementations are shown in Table 5.1.

The first way of implementing reverse, shown in Table 5.1(a) is the one which probably occurs most readily to users of imperative languages. It works from the outside in, swapping leftmost and rightmost elements, stopping when we reach the middle of the array. This approach would work equally well in an imperative language, in which we would be updating the array in-place. As one would expect for an algorithm originating in the realm of imperative programming, this first reverse algorithm is single-threaded.

A programmer who has experience with functional languages and less

```
reverse array =
    let
        arraysize           = size array
        halfway             = size / 2
        rev (array, index)  =
            if index < halfway then
                let
                    left        = subscript (array, index)
                    right       = subscript (array, arraysize - index - 1)
                    tmparray    = update(current, index, right)
                    newarray    = update(tmparray, size - index - 1, left)
                in
                    rev (newarray, index + 1)
            else
                array
    in
        rev (array, 0)
```

(a) A traditional single-threaded implementation of reverse.

```
reverse array =
    let
        arraysize           = size array
        rev (array, index)  =
            if index < arraysize then
                let
                    element     = subscript (array, arraysize - index - 1)
                    newarray    = update(array, index, element)
                in
                    rev (newarray, index + 1)
            else
                array
    in
        rev (array, 0)
```

(b) A simpler non-single-threaded implementation of reverse.

Table 5.1: *(continued over...)*

```
reverse array =
    let
        arraysize                = size array
        reverse-subscript index  = subscript (array, arraysize - index - 1)
    in
        create (size, reverse-subscript)
```

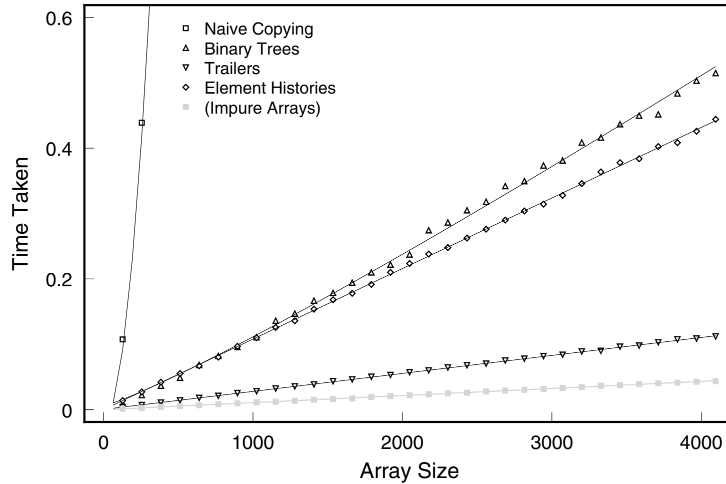
(c) A quick, monolithic, implementation of reverse.

Table 5.1: Specification of three implementations of array reversal.

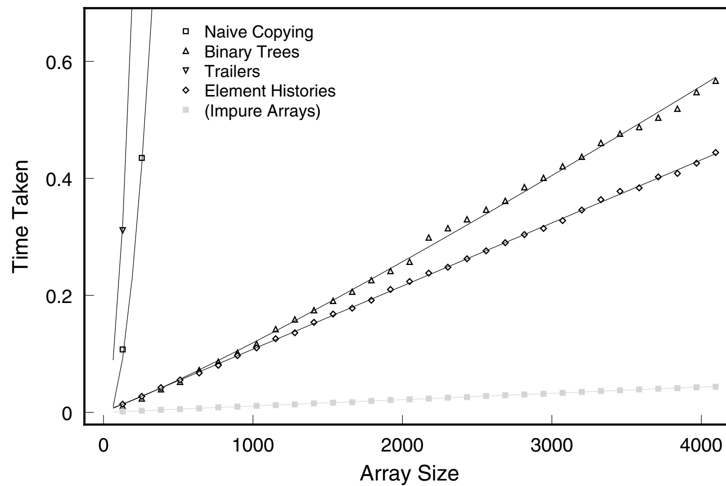
imperative language indoctrination, may realize that we can reverse the array just by working left to right (see Table 5.1(b)). This algorithm relies on the fact that we can still access the original array, and not worry that it will have been overwritten. It performs the same number of read and update accesses and eliminates a little algorithmic complexity. These differences do mean that unlike the first algorithm, this second one is not single-threaded, but this should not be considered a problem or failing.

The third and final algorithm is one that comes from looking at the problem a little more deeply. Array reversal changes every single element of the array, and so the reversed array has little in common with the original one. Since array update does not perform update-in-place but, semantically speaking, copies the array making a single change, the other techniques generate $n - 2$ intermediate arrays, each being a partially reversed array. We can skip these intermediate arrays and the creation costs associated with them creating a brand new array and filling it with the contents of the original array, reversed. This 'monolithic' approach provides the fastest and simplest implementation for reverse, and is also applicable to other algorithms which significantly reorganize an array; monolithic array use is no panacea however, and may not be suited to complicated algorithms or algorithms which may only change a few elements of the array.

Having seen the three different implementations of reverse, we now

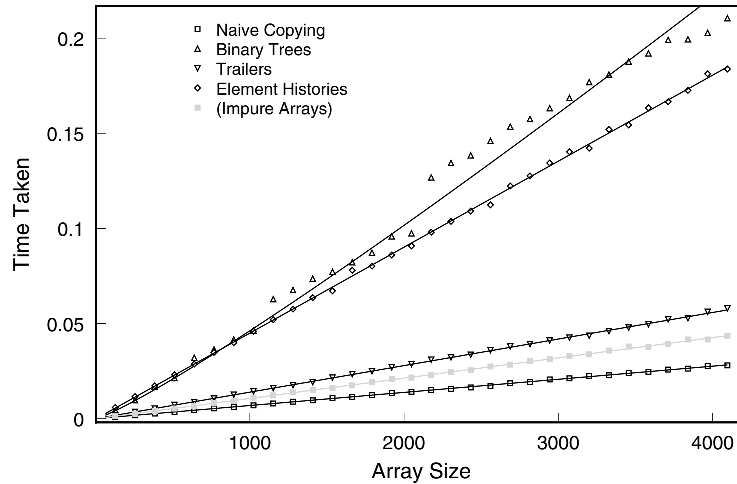


(a) For the very traditional reverse algorithm, both Element Histories and Trailers turn in $O(n)$ performance, while Binary Trees achieve $O(n \log n)$ performance. Naively copying the array at each update is $O(n^2)$.



(b) For the simpler reverse algorithm, the Element Histories again provides $O(n)$ performance, but Trailers is $O(n^2)$. Naive copying is also $O(n^2)$, but Trailers is actually three times slower. As before, Binary Trees are $O(n \log n)$.

Figure 5.1: *(continued over...)*



(c) For the monolithic approach, where instead of updating the original array, we create an entirely new one, all methods except Binary Trees turn in $O(n)$ performance. As in the previous examples, Binary Trees are $O(n \log n)$

Figure 5.1: Performance for the three implementations of array reversal.

need to examine their performance for each of the functional array methods. Since each of the implementations is, in its own way, reasonable, we would hope that they would take a similar amount of time, only differing by some constant factor. Figure 5.1 shows the actual results of timing each reverse algorithm for each of four array methods[1] (it also shows timings for mutable (non-functional) arrays).

Timings for the first version of `reverse`, Figure 5.1(a), show that for single threaded access, both Trailers and Element histories offer $O(n)$ performance. The graph also shows that Element Histories are nearly four times slower than Trailers. This isn't such a surprise, since Element Histories is a more complex technique[2]. Binary Trees offer $O(n \log n)$

[1] The timings come from executing an implementation of the algorithms described, written in Standard ML. Testing was done using *Standard ML of New Jersey*, running on a NeXTstation Turbo (Mach 2.5/BSD variant Unix, Motorola MC68040 processor at 33MHz, 48MB of RAM).

[2] In addition, it is worth noting that our implementation of Element Histories was done in a modular and layered way. An implementation oriented around execution speed could

performance[3], but despite that, when it comes to actual execution times the method comes very close to matching the times for Element Histories[4], particularly for small n. Binary Trees do, however, require $O(\log n)$ space for each array update, unlike Trailers and Element Histories, which only need $O(1)$ space. Moving on to Naive Copying, as one might expect, this technique turns in $O(n^2)$ performance.

Things become interesting when we examine the performance of the second implementation of reverse, see Figure 5.1(b). For the most part, the graphs are very similar, with the exception of the line for Trailers. Trailers has gone from a speedy $O(n)$ to a dreadful $O(n^2)$, and in terms of actual performance times, it is three times slower than Naive Copying, which is also $O(n^2)$. This highlights the fundamental weakness of Trailers (which we mention in Chapter 2): for single-threaded algorithms, Trailers is fine, turning in $O(1)$ performance for array subscript and update, but when the array is used non-single-threadedly, it is a lottery as to what kind of performance will result. In some cases, it may still manage to give $O(1)$ amortized time for these accesses, but in many others it will not.

Finally, we can move on to the last implementation of reverse, shown in Figure 5.1(c). What we are really looking at in this case is how long it takes to create a new array and traverse an old one. All techniques except Binary Trees turn in $O(n)$ performance. Binary Trees takes $O(n \log n)$.

We have seen here that in for some techniques, especially Trailers, subtly different algorithms that intuitively seem equivalent can produce vastly different execution times. Element Histories however, achieves consistent results, turning in $O(n)$ performance in all three cases. Binary Trees also give consistent results, always taking $O(n \log n)$ time. We've also seen that for small n, Binary Trees may be a viable and somewhat simpler approach to use.

---

almost certainly do better.

[3]In fact, while we have drawn an $n \log n$ curve through the points, an $n \lceil \log n \rceil$ curve would be more appropriate for our particular implementation.

[4]The Binary Tree based array implementation was fairly well optimized. Thus improvements to the implementation of Element Histories would widen the gap and also make the comparison a slightly fairer one.

5.3 The Multiple Versions Test

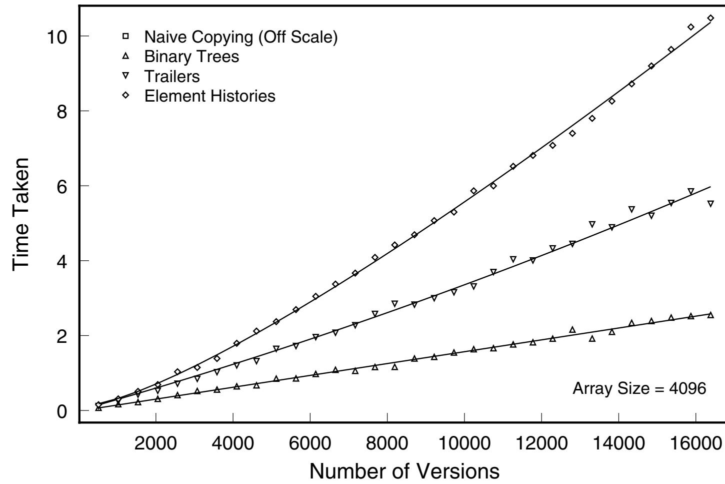
In this test, we make v versions of an array, all subtly different. We do this by repeatedly choosing an array version at random from those that exist, choosing an element, at random, and updating it, to make a new array version. We start with just one array and end when we have generated v versions.

The primary purpose of this test is to reveal the worst case behaviour of Element Histories. Recall that with Element Histories, the time taken to retrieve the most recently read or written value of an array element is $O(1)$, but otherwise it may be $O(\log v_e)$, where v_e is the number of different versions stored in that element. In our performance test, $v_e \approx v/n$. Looking at Figure 5.2(a), we can see that when as v exceeds n, the graph takes on a $\log v$ component — since n is constant we cannot be sure whether it is $\log v$ or, as we asserted earlier $\log(v/n)$. Trailers also seems to be following some sort of curve, perhaps a gentle $O(n^2)$ curve — it's hard to be sure exactly what kind of curve to draw given the unpredictable behaviour of Trailers. Certainly with sufficient analysis we could probably find out, but this itself makes a point about Trailers, that we need to go to some trouble to know exactly how it will perform.

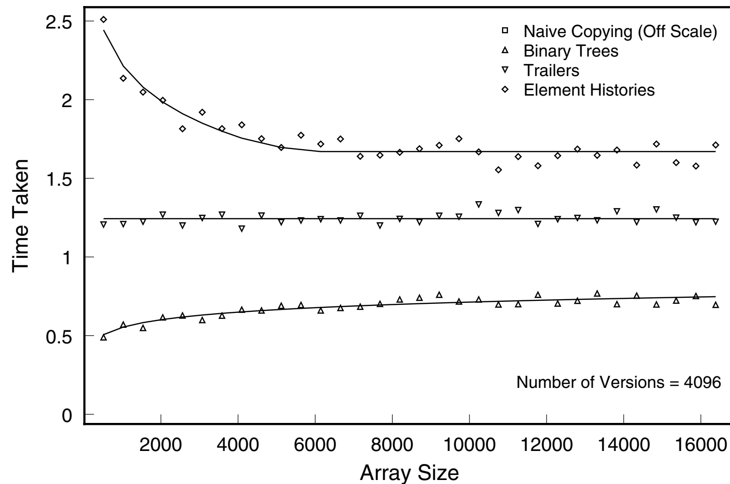
In the second graph, Figure 5.2(b), we keep v constant and vary the size of the array, n. For Trailers, this results in constant time, but for Element Histories, we see that as we increase array size, we decrease access time, until we reach the point where $n > v$. Binary Trees reveal their $O(\log n)$ component here also.

5.4 Conclusion

Although our technique is more complex than the other approaches, and is therefore slower by a small factor in simple cases, it lacks the terrible worst case performance of Trailers, and does better than Binary Trees in almost all cases (see below). In many situations our technique turns in



(a) Using a test that creates v versions of an array, performing v reads and v updates to do so, we see Element Histories' worst case performance. Instead of taking $O(v)$ time, it takes $O(v \log(v/n))$, where n is the size of the array. Trailers, on the other hand, manages to turn in $O(v)$ performance. The algorithm takes $O(v)$ time under Binary Trees because we are not varying the size of the array, only the number of versions.



(b) For Element Histories, the test takes less time as we increase the size of the array, n, (the number of concurrent array versions, v, is kept constant). The shape of the graph demonstrates that the logarithmic component is indeed $O(\log(v/n))$ and not just $O(\log v)$. The graph also reveals the $O(\log n)$ component present in Binary Trees.

Figure 5.2: Worst case performance for Element Histories.

$O(1)$ time for array subscript and update, and it's worst case performance is only $O(\log v_e)$, where v_e is the number of different versions of that element that exist, making its worst case performance better than the worst case performance of the other techniques we've looked at, for most applications.

We have also observed that binary trees may sometimes be a viable approach, especially when the size of the array, n, is small and v/n is large (v is the number of array versions), since in this case $O(\log n) < O(\log v/n)$ (since typically $v_e \approx v/n$). In this one case, Binary Trees does better than our approach.

Finally, we have seen that the techniques do best when array versions are similar; if we are making wholesale changes to an array, we probably do better to just create a new array than to use repeated updates to create an array which is a descendent of an original one. However, if we do need to do the latter, Element Histories will turn in respectable performance.

Chapter 6

Case Study, Implementing Heaps

In this chapter we take a look at one use to which functional arrays can be put, that of implementing a functional heap. This is of interest for several reasons, one being that there is an equivalence between functional arrays and functional heaps, and another being that heaps themselves raise some interesting issues about the axioms of functional programming. Finally, functional heap implementation has a particular significance to the author, since it was a need for an efficient functional heap that originally motivated her work in this area.

6.1 Introduction

The heap data structure, a structure which is very useful for supporting multi-linked data structures[4] and unification based algorithms, is an ideal candidate for implementation using functional arrays. A heap abstract data type supports the following operations:

$$
\begin{aligned}
\text{empty} &: heap(\alpha) \\
\text{allocate} &: (\alpha, heap(\alpha)) \rightarrow (location(\alpha), heap(\alpha)) \\
\text{read} &: (location(\alpha), heap(\alpha)) \rightarrow \alpha \\
\text{update} &: (location(\alpha), heap(\alpha), \alpha) \rightarrow heap(\alpha)
\end{aligned}
$$

A heap is like an extensible piece of RAM, we can place an object in the heap, and then refer to it using a pointer, referencing a *heap location*. We can not only read the location; we can also update it. However, the update operation remains fully functional, as it returns a new heap with the change made. If a read operation is made on the updated location using the old heap, the old value is returned, thus referential transparency is preserved.

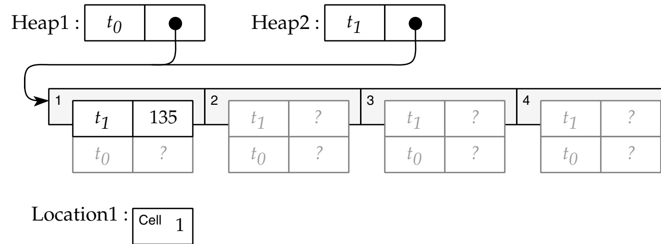
Our goal, as with the functional array construct, is to have an heap data structure whose locations can be updated or read in constant time, and where the same locations in older versions of the heap retain their old values, which are still accessible in a reasonable time.

6.2 A Straightforward Heap Implementation

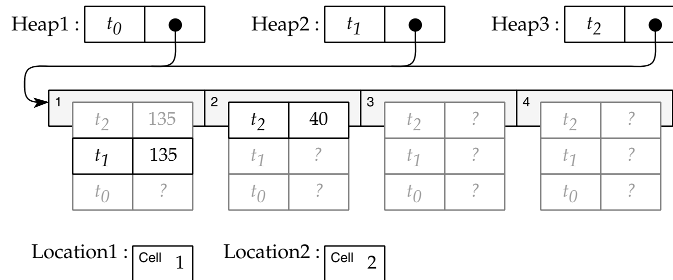
If one assumes a functional array construct already exists, we can immediately see a simple implementation method for this construct (see Figure 6.1). The heap can be a functional array[1], with heap locations just being indexes into that array. Usually heap locations tend to be very independent of each other, so we desire a functional array where updates to one element, or a group of elements, has little or no effect on other array elements. This is a feature of the functional array mechanism we have presented, but is lacking in most other functional array approaches (see Chapter 2).

There is only one potential problem with an array based heap implementation, and that is one of garbage collection for unreferenced heap locations. A heap location created by allocate may exist forever after its creation, even if all references to that location get forgotten. Whether or not this happens depends on exactly how allocation of heap pointers is managed and how garbage collection is performed on functional arrays, the later issue being one we address in our later work[20]. An obvious solution to this problem would be to add the following operation to our

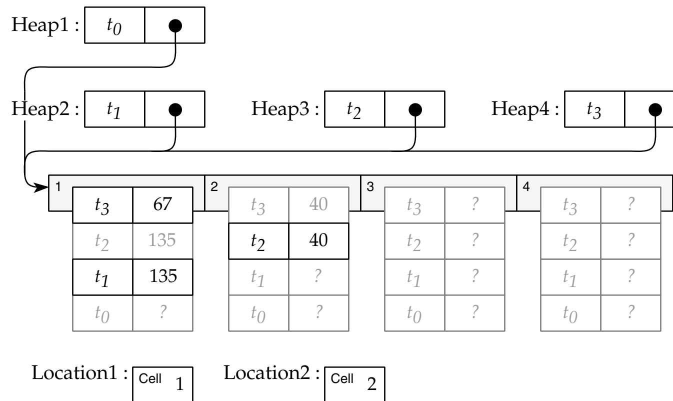
[1]We might perhaps add some housekeeping information, to make allocation of new heap locations faster.



(a) A heap data structure using an element-history based functional array. One heap location, Location1, has been allocated in the heap, and points to the integer value 135.



(b) The heap data structure after the allocation of another heap location. Note that the newly allocated location only exists in Heap3.



(c) The heap data structure after Location1 has been updated.

Figure 6.1: Implementing heaps using Functional Arrays.

heap specification:

$$\textsf{deallocate} \quad : \quad (\mathit{location}(\alpha), \mathit{heap}(\alpha)) \rightarrow \mathit{heap}(\alpha)$$

Since the location could be deallocated in one heap, but not in others, this is not necessarily such a bad approach, since at least the possibility of the 'resource freed while references to it still exist' bug is reduced. In some applications however, explicit deallocation is something the programmer would rather avoid if possible.

We can avoid having explicit deallocation if we implement heaps at a system level, and thereby allow the garbage collector to understand the structure of heaps. In fact, if heaps are primitive to the system, we can write functional array support in user level code, since an array could be seen as the combination of a heap and an immutable array (or vector) of heap locations. However, when we look into the issue of implementing heaps at a primitive level, and simplifying the data structures involved, we actually encounter some intriguing issues.

6.3 Problems over Referential Transparency

When we examine heaps, we uncover some curious and slightly counter-intuitive facts about referential transparency, which is a basic tenet of functional programming. The issues we raise here are interesting and probably worthy of further study, but given the issue is only very weakly related to functional arrays, we shall only touch on it briefly.

Referential transparency makes it easier for us to reason about the behaviour of functional programs, making it a very useful property, and one we should be reluctant to break. However, there are cases where it is actually more of a hindrance than a help because the rules a function must obey to be referentially transparent may actually prevent it from operating in the most useful and natural way. The heap abstract data type is an example of this. There is a very close relationship between heaps and heap locations; normally one would only expect to use a heap location with the heap to which it belongs (i.e the heap in which it was

first created or in a descendant of that heap). Referential Transparency, however, states that we can use a location with heaps other than the one it was actually created for. This additional flexibility provides nothing useful for most normal programs, but instead allows code sequences that are almost certainly erroneous to be executed without any error being raised, and prevents some optimizations.

It would be nice if we could both have our cake and eat it; retain referential transparency and yet somehow be able to express the fact that there is a connection between a particular heap and its locations, and thereby still be able to apply some useful optimizations, and catch programming errors. The closely coupled relationship we desire would not so much break referential transparency as approximate it. Heaps themselves need not admit equality, making the issue of whether two of them are equal academic, and locations should only be used with heaps to which they belong or their descendants. We could say that failure to do so results in $\perp$[2] (i.e. it is an untrapable error). Finally, we could require that locations may be tested for equality, but only if there exists a heap in which both are valid; testing locations which come from completely different heaps for equality could, once again, cause $\perp$ to be the result.

Ideally, we could statically enforce restrictions only allowing code that uses locations and heaps sensibly, thus preventing $\perp$ from ever being generated by the heap access functions. If this were the case, the heap interface could appear referentially transparent while still allowing useful optimizations and preventing erroneous code sequences. Unfortunately, how such restrictions could be applied statically is unknown at this time. At present detection of such erroneous use can be done at runtime (raising an error if the access is an invalid one — such checking can be done with only a very minor runtime penalty, and those who feel the 'the need for speed' could disable runtime checking if they were convinced their

[2]Or 'may result in $\perp$', it may also return a normal result depending on the optimizations performed by the compiler. If a result is returned, it could still be a *correct* answer, in that it will be the value that a fully referentially transparent implementation would return. Returning $\perp$ can be viewed as electing not to give an answer at all, rather than being an answer in itself.

code had no faults, in the same way as they might be able to disable runtime array bounds checking).

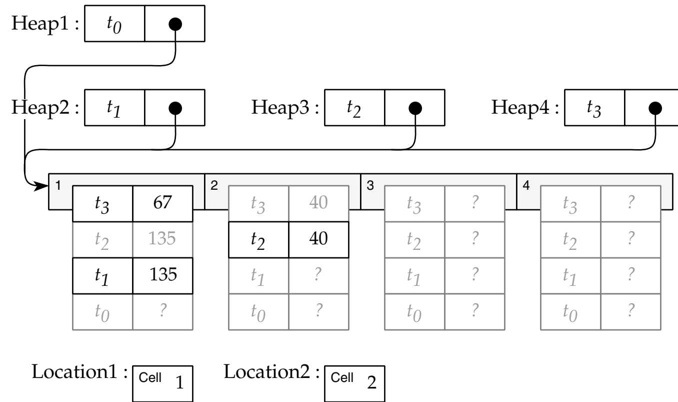
In languages such as Standard ML[19], preserving true referential transparency is often considered not quite so important, and so the optimisation we will present would probably not be considered a problem, especially since it can catch programming errors. However, purer languages are probably better off with the simple scheme, unless the requirements can be enforced at compile time (or unless they support linear/unique types, since they could be used to make all heaps distinct).

6.4 Optimising the Implementation of Heaps

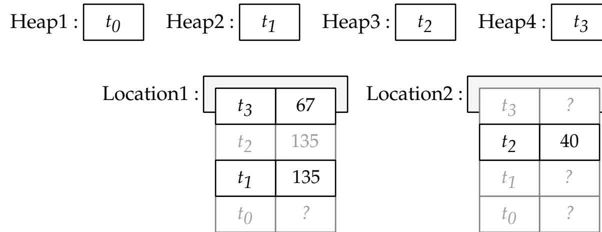
Earlier we informally described how one could implement heaps given the existence of functional arrays. In fact, if the functional arrays are implemented using element histories, we can simplify the data structure used by eliminating some parts of the functional array scheme that prove unnecessary.

Figure 6.2 shows a typical data structure arrangement for implementing heaps using history stack based functional arrays. We can simplify this scheme by removing an unnecessary indirection; instead of making the heap location hold an index into an array of element histories, we make the location itself be an element history. Eliminating this indirection adds a minor access time speedup, but more importantly, simplifies heap maintenance, since we do not have to allocate an array or worry about dynamically re-sizing the array, nor do we need to worry as much about garbage collection - if a heap location ceases to be referenced anywhere, the element history associated with that heap location will be removed.

The only downside to this approach is that it is only weakly referentially transparent. Since the data is actually stored *inside* the location, rather than the location being a pointer into the heap, the 'heap' becomes solely a timestamp. The problem with this refinement is that while in



(a) There is some redundancy in the representation using functional arrays.



(b) We can simplify the data structure for heaps, by using element histories directly. This simplification may be problematic, however (*see text*).

Figure 6.2: Optimization for heaps.

many respects it is more logical, code which relied on referentially transparent semantics could fail. As has been pointed out earlier, usually only very bizarre and probably erroneous code would be affected.

6.5 Conclusion

We've shown here how the element history data structure can be used in other situations besides the management of functional arrays. We've also seen how current type systems and the conditions for referential transparency make the efficient implementation of heap data structures awkward. This latter point is probably worthy of further study.

Chapter 7

Conclusion

We have examined the problem of providing functional arrays, looked at previous approaches to the problem, and then developed our own general solution, which offers better performance than prior methods.

We began with a scheme that offered partial persistence, and developed that to provide full persistence, presenting and comparing two timestamping models that each provide full persistence. Having developed a technique for implementing truly functional arrays, we've examined its performance when compared to prior methods. We've also examined one use of functional arrays, namely their use to implement heaps, and have explored some special issues that relate to heaps.

Our technique is both effective in theory and practice, providing constant time performance for single threaded array access, and fast access to old array versions — a significant improvement over previous approaches. As well as allowing favourite imperative array algorithms to be used with little penalty in a functional context, it also allows the practical use of arrays as a persistent data structure. It is probably the first kind of array where the programmer is free to to use arrays in whatever way they like without risking huge penalties.

Finally, our work here also reveals fertile ground on which to conduct further research. Two important issues we've not covered here are special garbage collection techniques for our functional arrays, and optimisations that can impact access times for our structures (including techniques to

reduce storage and access overheads, and ways to break up a large array when it holds many dissimilar versions). Also, our array structure is very amenable to being used in parallel access situations, and has impressive potential in this regard. These issues, among others, are covered by our later work[20].

Appendix A

Source Listings

In this section we list all some sample source code for the techniques covered by this thesis. The source code is written in Standard ML. The reader should also note that on occasion the functions are named slightly differently from the names used in the main text — however, the correlation between function names used in the source code and names used in the text should be clear enough.

A.1 Functional Arrays using totally ordered time-stamps

A.1.1 Structure Signatures

array.sig

```
signature ARRAY =
  sig
    type 'a array
    exception Size
    exception Subscript
    val tabulate : int * (int -> '1a) -> '1a array
    val sub : '1a array * int -> '1a
    val size : '1a array -> int
    val update : '1a array * int * '1a -> '1a array
    val method : string
  end
```

history.sig

```
signature HISTORY =
  sig
```

```
    type 'a history
    type timestamp
    exception Unbound
    val create   : unit -> ('a -> 'a history) * timestamp
    val locate   : 'a history * timestamp -> 'a history * 'a
    val record   : ('a history * timestamp) * 'a -> 'a history * timestamp
    val history_string   : ('a -> string) -> 'a history -> string
    val time_string : timestamp -> string
  end
```

rewriting-splaytree.sig

```
signature REWRITING_SPLAYTREE =
  sig
    datatype 'a splay =
      SplayObj of {
        value : 'a,
        right : 'a splay,
        left : 'a splay
      }
    | SplayNil
    datatype 'a position =
      Before | After | Identical | Delete | Rewrite of 'a splay -> 'a splay
    val splay : ('a -> 'a position) * 'a splay -> LibBase.relation * 'a splay
    val join : 'a splay * 'a splay -> 'a splay
  end
```

rewriting-splayaccess.sig

```
signature REWRITING_SPLAYACCESS =
  sig
    eqtype 'a splay
    datatype 'a position =
      Before | After | Identical | Delete | Rewrite of 'a splay -> 'a splay
    exception NotFound
    val empty : 'a splay
    val insert : 'a splay * ('a -> 'a position) * 'a -> 'a splay
    val find : 'a splay * ('a -> 'a position) -> 'a splay * 'a option
    val remove : 'a splay * ('a -> 'a position) -> 'a splay * 'a option

    val findafter : 'a splay * ('a -> 'a position) -> 'a splay * 'a option
    val afteronly : ('a -> 'a position) -> ('a -> 'a position)
    val findbefore : 'a splay * ('a -> 'a position) -> 'a splay * 'a option
    val beforeonly : ('a -> 'a position) -> ('a -> 'a position)

    val root : 'a splay -> 'a option

    val listItems : 'a splay -> 'a list
    val app : ('a -> 'b) -> 'a splay -> unit
    val revapp : ('a -> 'b) -> 'a splay -> unit
    val fold : ('a * 'b -> 'b) -> 'a splay * 'b -> 'b
    val revfold : ('a * 'b -> 'b) -> 'a splay * 'b -> 'b

    val string  : ('a -> string) -> 'a splay -> string
  end
```

timestamps.sig

```
signature TIMESTAMPS =
  sig
    eqtype timestamp

    val create    : unit -> timestamp
    val insertAfter  : timestamp -> timestamp
    val insertBefore : timestamp -> timestamp

    val delete  : timestamp -> unit

    val equal   : timestamp * timestamp -> bool
    val compare : timestamp * timestamp -> LibBase.relation

    (* Functions to tranverse and examine the data structure *)

    val pred    : timestamp -> timestamp option
    val succ    : timestamp -> timestamp option
    val string  : timestamp -> string
    val label   : timestamp -> string
  end
```

order-list.sig

```
signature ORDERLIST =
    sig
        eqtype 'a olist

        val create           : '1a -> '1a olist
        val insertAfter      : '1a olist * '1a -> '1a olist
        val insertBefore     : '1a olist * '1a -> '1a olist
        val insertManyAfter  : '1a olist * '1a list -> '1a olist list
        val insertManyBefore : '1a olist * '1a list -> '1a olist list

        val delete        : 'a olist -> 'a

        val contents      : 'a olist -> 'a
        val setContents   : 'a olist * 'a -> unit
        val applyContents : 'a olist * ('a -> ('a * 'b)) -> 'b

        (* Comparison functions *)

        val equal    : 'a olist * 'a olist -> bool
        val compare  : 'a olist * 'a olist -> LibBase.relation

        (* Functions to tranverse and examine the data structure *)

        val pred     : 'a olist -> 'a olist option
        val succ     : 'a olist -> 'a olist option
        val string   : ('a -> string) -> 'a olist -> string
        val label    : 'a olist -> string
    end
```

circular-list.sig

```
signature CIRCULARLIST =
    sig
        eqtype 'a clist
        val create : '1a -> '1a clist
```

```
        val insert : '1a clist * '1a -> '1a clist
        val delete : 'a clist -> 'a
        val equal  : 'a clist * 'a clist -> bool
        val splice : 'a clist * 'a clist -> unit
        val chop   : 'a clist * 'a clist -> unit
        val move   : 'a clist * 'a clist * 'a clist -> unit

        val contents      : 'a clist -> 'a
        val setContents   : 'a clist * 'a -> unit
        val applyContents : 'a clist * ('a -> ('a * 'b)) -> 'b

        val succ : 'a clist -> 'a clist
        val pred : 'a clist -> 'a clist

        val string : ('a -> string) -> 'a clist -> string
        val revstring : ('a -> string) -> 'a clist -> string
    end
```

double-list.sig

```
signature DOUBLELIST =
    sig
        eqtype 'a dlist
        val create : '1a -> '1a dlist
        val insertAfter  : '1a dlist * '1a -> '1a dlist
        val insertBefore : '1a dlist * '1a -> '1a dlist
        val delete       : 'a dlist -> 'a

        val chop         : 'a dlist * 'a dlist -> unit
        val moveAfter    : 'a dlist * 'a dlist * 'a dlist -> unit
        val moveBefore   : 'a dlist * 'a dlist * 'a dlist -> unit

        val contents      : 'a dlist -> 'a
        val setContents   : 'a dlist * 'a -> unit
        val applyContents : 'a dlist * ('a -> ('a * 'b)) -> 'b

        val equal  : 'a dlist * 'a dlist -> bool

        val pred : 'a dlist -> 'a dlist option
        val succ : 'a dlist -> 'a dlist option
        val string : ('a -> string) -> 'a dlist -> string
    end
```

A.1.2 Structure Implementations

double-list.sml

```
abstraction DoubleList : DOUBLELIST =
    struct
        datatype 'a dlist =
            Cell of { pred:'a dlist option ref,
                      succ:'a dlist option ref,
                      contents:'a ref}

        val pred    = fn Cell {pred=ref pred,...} => pred
        val succ        = fn Cell {succ=ref succ,...}     => succ
        val contents    = fn Cell {contents=ref contents,...} => contents
        val setContents =
            fn (Cell {contents,...}, newContents) => contents := newContents
        val applyContents =
```

```
    fn (Cell {contents=contents as ref value,...}, applyFn) =>
        let
           val (newContents, result) = applyFn value
        in
          (contents := newContents; result)
        end

fun create contents : '1a dlist =
    Cell {succ=ref NONE,pred=ref NONE,contents=ref contents}

val insertAfter  =
    fn (left as Cell {succ=leftsucc as ref leftsucc_val, ...},
        contents) =>
      case leftsucc_val of
          SOME (right as Cell {pred=rightpred, ...}) =>
            let
                val newCell =
                    Cell {pred=ref (SOME left),
                          succ=ref (SOME right),
                          contents=ref contents}
            in
                leftsucc  := SOME newCell ;
                rightpred := SOME newCell ;
                newCell
            end
        | NONE =>
            let
                val newCell =
                    Cell {pred=ref (SOME left),
                          succ=ref NONE,
                          contents=ref contents}
            in
                leftsucc  := SOME newCell ;
                newCell
            end

val insertBefore  =
    fn (right as Cell {pred=rightpred as ref rightpred_val, ...},
        contents) =>
      case rightpred_val of
          SOME (left as Cell {succ=leftsucc, ...}) =>
            let
                val newCell =
                    Cell {pred=ref (SOME left),
                          succ=ref (SOME right),
                          contents=ref contents}
            in
                leftsucc  := SOME newCell ;
                rightpred := SOME newCell ;
                newCell
            end
        | NONE =>
            let
                val newCell =
                    Cell {pred=ref NONE,
                          succ=ref (SOME right),
                          contents=ref contents}
            in
                rightpred  := SOME newCell ;
                newCell
            end
```

```
val delete  =
    fn it as Cell {pred=itspred as ref left_opt,
                   succ=itssucc as ref right_opt,
                   contents=ref contents} =>
      ( case left_opt of
            SOME (Cell {succ=leftsucc,  ...}) =>
              leftsucc  := right_opt
          | NONE => ()  ;
        case right_opt of
            SOME (Cell {pred=rightpred, ...}) =>
              rightpred := left_opt
          | NONE => ()  ;
        itssucc   := NONE;
        itspred   := NONE;
        contents )

val chop  =
    fn (rangeleft  as Cell {pred=rangepred as ref left_opt, ...},
        rangeright as Cell {succ=rangesucc as ref right_opt,...}) =>
      ( case left_opt of
            SOME (Cell {succ=leftsucc,  ...}) =>
              leftsucc  := right_opt
          | NONE => ()  ;
        case right_opt of
            SOME (Cell {pred=rightpred, ...}) =>
              rightpred := left_opt
          | NONE => ()  ;
        rangepred := NONE;
        rangesucc := NONE )

val moveAfter  =
    fn (rangeleft  as Cell {pred=rangepred as ref left_opt, ...},
        rangeright as Cell {succ=rangesucc as ref right_opt,...},
        left as Cell {succ=leftsucc as ref leftsucc_val, ...}) =>
      ( case left_opt of
            SOME (Cell {succ=leftsucc,  ...}) =>
              leftsucc  := right_opt
          | NONE => ()  ;
        case right_opt of
            SOME (Cell {pred=rightpred, ...}) =>
              rightpred := left_opt
          | NONE => ()  ;
        case leftsucc_val of
            SOME (right as Cell {pred=rightpred, ...}) =>
              ( rangepred := SOME left ;
                rangesucc := SOME right ;
                leftsucc  := SOME rangeleft ;
                rightpred := SOME rangeright )
          | NONE =>
              ( rangepred := SOME left ;
                rangesucc := NONE ;
                leftsucc  := SOME rangeleft ) )

val moveBefore  =
    fn (rangeleft  as Cell {pred=rangepred as ref left_opt, ...},
        rangeright as Cell {succ=rangesucc as ref right_opt,...},
        right as Cell {pred=rightpred as ref rightpred_val, ...}) =>
      ( case left_opt of
            SOME (Cell {succ=leftsucc,  ...}) =>
```

```
                leftsucc  := right_opt
              | NONE => ()  ;
            case right_opt of
                SOME (Cell {pred=rightpred, ...}) =>
                  rightpred := left_opt
              | NONE => ()  ;
            case rightpred_val of
                SOME (left as Cell {succ=leftsucc, ...}) =>
                  ( rangepred := SOME left ;
                    rangesucc := SOME right ;
                    leftsucc  := SOME rangeleft ;
                    rightpred := SOME rangeright )
              | NONE =>
                  ( rangepred := NONE ;
                    rangesucc := SOME right ;
                    rightpred  := SOME rangeright ) )


    val equal = fn (Cell {pred=left1,...},Cell {pred=left2,...})
                    => (left1 = left2)

    fun string doContents startCell =
        let
            fun doPrev NONE = "$["
              | doPrev (SOME cell) =
                ( doPrev (pred cell) ^
                  doContents (contents cell) ^ ",")
            fun doRest NONE = "]$"
              | doRest (SOME cell) =
                  ( "," ^ doContents (contents cell) ^ doRest (succ cell) )
        in
            doPrev (pred startCell)
          ^ "*" ^ doContents (contents startCell) ^ "*"
          ^ doRest (succ startCell)
        end
end
```

circular-list.sml

```
abstraction CircularList : CIRCULARLIST =
    struct
        datatype 'a clist =
            Cell of {pred:'a clist ref, succ:'a clist ref, contents:'a ref}

        fun create contents : '1a clist =
            let
                val left = ref (System.Unsafe.cast 0 : '1a clist)
                val right = ref (System.Unsafe.cast 0 : '1a clist)
                val base = Cell {succ=right,pred=left,contents=ref contents}
            in
                left  := base;
                right := base;
                base
            end

        val pred        = fn Cell {pred=ref pred,...} => pred
        val succ        = fn Cell {succ=ref succ,...} => succ
        val contents    = fn Cell {contents=ref contents,...} => contents
        val setContents =
            fn (Cell {contents,...}, newContents) => contents := newContents
```

```
val applyContents =
    fn (Cell {contents=contents as ref value,...}, applyFn) =>
        let
           val (newContents, result) = applyFn value
        in
          (contents := newContents; result)
        end

val insert  =
    fn (left as
            Cell {succ=leftsucc as
                           ref (right as
                                    Cell {pred=rightpred, ...}), ...},
        contents) =>
        let
            val newCell =
                Cell{pred=ref left,
                     succ=ref right,
                     contents=ref contents}
        in
            leftsucc  := newCell ;
            rightpred := newCell ;
            newCell
        end

val delete  =
    fn it as Cell {pred=itspred as
                            ref (left  as Cell {succ=leftsucc,  ...}),
                   succ=itssucc as
                            ref (right as Cell {pred=rightpred, ...}),
                   contents=ref contents} =>
      ( leftsucc  := right;
        rightpred := left;
        itssucc   := it;
        itspred   := it;
        contents )

val equal = fn (Cell {contents=x,...},Cell {contents=y,...})
                => (x = y)

val splice =
    fn (first  as Cell {pred=firstPredptr  as
          ref (prefirst  as Cell {succ=prefirstSuccptr,...}),  ...},
        second as Cell {pred=secondPredptr as
          ref (presecond as Cell {succ=presecondSuccptr,...}), ...}) =>
      ( prefirstSuccptr := second ;
        presecondSuccptr := first ;
        firstPredptr := presecond ;
        secondPredptr := prefirst )

val chop = fn (left, right) => splice (left, succ right)

val move = fn (left, right, after) =>
    ( splice (left, succ right) ; splice (succ after, left) )



fun string doContents startCell =
    let
        fun doRest cell =
```

```
                if equal(cell,startCell) then
                    "]%"
                else
                  ( "," ^ doContents (contents cell) ^ doRest (succ cell) )
        in
            "%[" ^ doContents (contents startCell) ^
              doRest (succ startCell)
        end

    fun revstring doContents startCell =
        let
            fun doRest cell =
                if equal(cell,startCell) then
                    "]%"
                else
                  ( "," ^ doContents (contents cell) ^
                      doRest (pred cell) )
        in
            "%[" ^ doContents (contents startCell) ^
              doRest (pred startCell)
        end

end
```

order-list.sml

```
structure OrderList : ORDERLIST =
(* structure OrderList = *)
  let
    open LibBase
  in
  struct
    structure CList = CircularList

    type 'a olist = {contents:'a, label:int, first:int ref} CList.clist
    exception EmptyList

    val firstref =  #first    o CList.contents : 'a olist -> int ref
    val label    =  #label    o CList.contents : 'a olist -> int
    val contents =  #contents o CList.contents : 'a olist -> 'a
    val isfirst  = fn (entry : 'a olist) =>
        let val {first=ref firstlabel, label, ...} = CList.contents entry
        in  firstlabel = label  end
    val equal = CList.equal : 'a olist * 'a olist -> bool
    val delete = fn (entry : 'a olist) =>
        let
          val {first=firstref as ref firstlabel, label=itslabel, contents} =
              CList.contents entry
        in
          if firstlabel = itslabel then
            let val successor = CList.succ entry in
              if equal (entry,successor) then
                raise EmptyList
              else
                firstref := label successor
            end
          else
            () ;
          CList.delete entry ;
          contents
```

```
    end


val succ = fn entry =>
    let val successor = CList.succ entry in
      if isfirst successor then NONE else SOME successor
    end

val pred = fn entry =>
    if isfirst entry then NONE else SOME (CList.pred entry)

val setLabel = fn (entry, newLabel) =>
    CList.applyContents
      (entry,
        fn {contents, label, first=first as ref firstlabel} =>
          ({contents=contents, label=newLabel, first=first},
          if label = firstlabel then first := newLabel else ()) )

val setContents = fn (entry, newContents) =>
    CList.applyContents
      (entry,
        fn {contents, label, first} =>
          ({contents=newContents, label=label, first=first}, ()) )

val applyContents = fn (entry,applyFn) =>
    let
      val revisedApplyFn =
        fn {contents, label, first} =>
          let val (newContents, value) = applyFn contents in
            ({contents=newContents, label=label, first=first}, value)
          end
    in
      CList.applyContents (entry,revisedApplyFn)
    end

val arena = 536870912

fun investigateGap (startEntry, n) =
    let
        val v0 = label startEntry
        fun walker (j, theEntry) =
            let
                val theGap = (label theEntry - v0) mod arena
            in
                case theGap of
                    0 => (j+1,arena)  (* used to be (j,arena-1) *)
                  | _ =>
                    if theGap <= j * j then
                        walker(j+1, CList.succ theEntry)
                    else
                        (j+1,theGap)
            end
    in
        walker(n, CList.succ startEntry)
    end

fun newLabels (startEntry, n) =
    let
      val firstLabel = label startEntry
      val (lastEntry,nodeGap) = investigateGap (startEntry, n)
      fun makeLabel nodeNo =
```

```
          ( nodeGap div lastEntry * nodeNo
            + (nodeGap mod lastEntry) * nodeNo div lastEntry
            + firstLabel )  mod arena
        fun fixEntry (nodeNo,thisEntry) =
            if nodeNo < lastEntry then
              ( setLabel (thisEntry, makeLabel nodeNo) ;
                fixEntry (nodeNo + 1, CList.succ thisEntry) )
            else ()
      in
        fixEntry(n+1, CList.succ startEntry) ;
        makeLabel
      end

(*   fun newLabel thisEntry = (** OBSOLETE **)
      let
          val thisLabel = label thisEntry
          and nextLabel = label (CList.succ thisEntry)
      in
          if thisLabel < nextLabel then
              (thisLabel + nextLabel) div 2
          else
              ((thisLabel + nextLabel) div 2 + arena div 2) mod arena
      end *)

  fun create contents =
      CList.create {contents=contents, label=0, first=ref 0}

  fun insertAfter (thisEntry,contents) =
        CList.insert
        ( thisEntry,
          { contents = contents,
            label    = newLabels (thisEntry,1) 1,
            first    = firstref thisEntry } )

  fun insertManyAfter (thisEntry,contentsList) =
      let
        val firstRef = firstref thisEntry
        val makeLabel = newLabels (thisEntry, length contentsList)
        fun doInsert (nil, _, _) = nil
          | doInsert (value :: rest, nodeNo, afterNode) =
            let val insertedNode =
                  CList.insert
                  ( afterNode,
                    { contents = value,
                      label    = makeLabel nodeNo,
                      first    = firstRef } )
            in
              insertedNode :: doInsert(rest, nodeNo + 1, insertedNode)
            end
      in
        doInsert (contentsList, 1, thisEntry)
      end

  fun insertBefore (thisEntry,newContents) =
      let val predEntry = CList.pred thisEntry
          val newEntry = insertAfter (predEntry,newContents)
      in if isfirst thisEntry then
            let val {label, first=firstref, ...} = CList.contents newEntry
            in  firstref := label end
          else () ;
          newEntry
```

```
        end

    fun insertManyBefore (thisEntry,newContentsList) =
        let val predEntry = CList.pred thisEntry
            val newEntries = insertManyAfter (predEntry,rev newContentsList)
        in
          if isfirst thisEntry then
            case newEntries of
                nil => ()
              | leftEntry :: _ =>
                let val {label, first=firstref, ...} = CList.contents leftEntry
                in  firstref := label end
          else () ;
          rev newEntries
        end


    fun compare (leftEntry,rightEntry) =
        if equal (leftEntry,rightEntry) then
            Equal
        else
            let
                val startLabel = !(firstref leftEntry)
                and  leftLabel = label  leftEntry
                and rightLabel = label rightEntry
            in
                if ((leftLabel - startLabel) mod arena)
                      < ((rightLabel - startLabel) mod arena) then
                    Less
                else
                    Greater
            end


    fun string doContents =
        let
            fun doEntry ({contents,label,first=ref firstlabel}) =
              ( "(" ^ (if firstlabel = label then "*" else "") ^
                makestring (label:int) ^ "," ^ doContents contents ^ ")" )
        in
            CList.string doEntry
        end

    val label = fn entry : 'a olist =>
        let val {label,first=ref firstlabel, ...} = CList.contents entry
        in  (if firstlabel = label then "*" else "") ^ makestring label end

end
end
```

order-list-2.sml

```
structure OrderList2 : ORDERLIST =
  let
    open LibBase
  in
  struct
    structure DList = DoubleList
    structure OList = OrderList
```

```
    type majorlist = unit OList.olist
    type minorlist = {label : int, major : majorlist} DList.dlist

    type 'a olist  = {contents:'a, minor : minorlist} DList.dlist

    val contents =  #contents o DList.contents : 'a olist -> 'a
    val setContents : 'a olist * 'a -> unit = fn (entry, newContents) =>
        DList.applyContents
          (entry,
            fn {contents, minor} => ({contents=newContents, minor=minor}, ()) )
    val applyContents : 'a olist * ('a -> ('a * 'b)) -> 'b = fn (entry,applyFn)
=>
        let
          val revisedApplyFn =
            fn {contents, minor} =>
              let val (newContents, value) = applyFn contents in
                ({contents=newContents, minor=minor}, value)
              end
        in
          DList.applyContents (entry,revisedApplyFn)
        end

    val minor = #minor o DList.contents : 'a olist -> minorlist
    val setMinor : 'a olist * minorlist -> unit = fn (entry, newMinor) =>
        DList.applyContents
          (entry,
            fn {contents, minor} => ({contents=contents, minor=newMinor}, ()) )

    val major : 'a olist -> majorlist =
        #major o DList.contents o #minor o DList.contents
    val setMajor = fn (entry, newMajor) =>
        DList.applyContents
          (minor entry,
           fn {label, major} => ({label=label, major=newMajor}, ()) )

    val majorAndLabel = fn (entry : 'a olist) =>
        let val contents = DList.contents (#minor (DList.contents entry))
        in  (#major contents, #label contents)  end

    val equal = DList.equal : 'a olist * 'a olist -> bool
    val succ  = DList.succ : 'a olist -> 'a olist option
    val pred  = DList.pred : 'a olist -> 'a olist option

    fun create contents : '1a olist =
        let
          val major = OList.create ()
          val minor = DList.create {label=0, major=major}
        in
          DList.create {contents=contents, minor=minor}
        end

    exception NotImplemented

 (*   fun newLabel thisEntry = (** OBSOLETE **)
        let
            val thisLabel = label thisEntry
            and nextLabel = label (CList.succ thisEntry)
        in
            if thisLabel < nextLabel then
                (thisLabel + nextLabel) div 2
```

```
        else
            ((thisLabel + nextLabel) div 2 + arena div 2) mod arena
    end *)

val standardGap  = 262144
val standardSize = 18

fun take (_,nil) = (nil,nil)
  | take (0,x)   = (nil,x)
  | take (n,h :: t) = let val (taken, rest) = take (n-1,t)
                      in  (h :: taken, rest)  end

fun group list =
    case take (standardSize,list) of
      (nil,nil) => nil
    | (taken, nil) => [taken]
    | (taken, rest) => taken :: group rest

fun deleteToStart startEntry =
    let
      fun deleter (n,thisEntry) =
          case DList.pred thisEntry of
            SOME predEntry => deleter (n+1,predEntry)
          | NONE => (DList.chop (thisEntry, startEntry) ; n)
    in
      deleter (1,startEntry)
    end

fun deleteToEnd startEntry =
    let
      fun deleter (n,thisEntry) =
          case DList.succ thisEntry of
            SOME succEntry => deleter (n+1,succEntry)
          | NONE => (DList.chop (startEntry, thisEntry) ; n)
    in
      deleter (1,startEntry)
    end

fun tabulateList (n,f) =
    let
       fun tabulate (0,acc) = acc
         | tabulate (n,acc) = tabulate (n-1,f (n-1) :: acc)
    in
       tabulate (n,nil)
    end

fun regroupAfter (startEntry : '1a olist) =
    let
      val minorEntry = minor startEntry
      val majorEntry = #major (DList.contents minorEntry)
      val minorCount = deleteToEnd minorEntry
      val majorCount = (minorCount + standardSize - 1) div standardSize
      val majorEntries = majorEntry ::
          OList.insertManyAfter
            (majorEntry, tabulateList (majorCount, fn n => ()))
      fun engroup (0,_,_,_,_) = ()
        | engroup (toDo, 0, _, SOME thisEntry,
                   _ :: (majorEntries as newMajor :: _)) =
          let
            val newMinor = DList.create {label=0, major=newMajor}
            val succEntry = DList.succ thisEntry
```

```
              in
                setMinor (thisEntry, newMinor);
                engroup (toDo-1, 1, newMinor, succEntry, majorEntries)
              end
            | engroup (toDo, inGroup, prevMinor, SOME thisEntry,
                       majorEntries as thisMajor :: _) =
              let
                val newMinor =
                   DList.insertAfter (prevMinor,
                                     {label=standardGap * inGroup,
                                      major=thisMajor})
                val succEntry = DList.succ thisEntry
              in
                setMinor (thisEntry, newMinor);
                engroup (toDo-1, (inGroup+1) mod standardSize, newMinor, succEnt
ry, majorEntries)
              end
        in
          engroup (minorCount, 0, minorEntry, SOME startEntry, majorEntries)
        end

    fun insertAfter (thisEntry : '1a olist,newContents) =
        let
          val thisMinor = #minor (DList.contents thisEntry)
          val {label=thisMinorLabel,major=thisMajor} = DList.contents thisMinor
        in let
          val newMinorLabel =
              case DList.succ thisMinor of
                  SOME succMinor =>
                    let
                      val {label=succMinorLabel,...} = DList.contents succMinor
                      val newMinorLabel =
                          thisMinorLabel
                          + (succMinorLabel - thisMinorLabel) div 2
                    in
                      if newMinorLabel > thisMinorLabel then
                        newMinorLabel
                      else
                        let
                          val SOME succEntry = DList.succ thisEntry
                        in
                          regroupAfter succEntry;
                          thisMinorLabel + standardGap
                        end
                    end
                | NONE => thisMinorLabel + standardGap
        in
          DList.insertAfter
             (thisEntry,
              {contents=newContents,
               minor=
                 DList.insertAfter
                    (thisMinor, {label=newMinorLabel, major=thisMajor})})
        end
        handle Overflow =>
            DList.insertAfter
             (thisEntry,
              {contents=newContents,
               minor=
                 DList.create{label=0, major=OList.insertAfter (thisMajor,())}})
        end
```

```
    fun create contents : '1a olist =
        let
          val major = OList.create ()
          val minor = DList.create {label=0, major=major}
        in
          DList.create {contents=contents, minor=minor}
        end



    fun regroupBefore (startEntry : '1a olist) =
        let
          val minorEntry = minor startEntry
          val majorEntry = #major (DList.contents minorEntry)
          val minorCount = deleteToStart minorEntry
          val majorCount = (minorCount + standardSize - 1) div standardSize
          val majorEntries = majorEntry ::
              OList.insertManyBefore
                (majorEntry, tabulateList (majorCount, fn n => ()))
          fun engroup (0,_,_,_,_) = ()
            | engroup (toDo, 0, _, SOME thisEntry,
                       _ :: (majorEntries as newMajor :: _)) =
              let
                val newMinor =
                    DList.create {label=0, major=newMajor}
                val predEntry = DList.pred thisEntry
              in
                setMinor (thisEntry, newMinor);
                engroup (toDo-1, 1, newMinor, predEntry, majorEntries)
              end
            | engroup (toDo, inGroup, prevMinor, SOME thisEntry,
                       majorEntries as thisMajor :: _) =
              let
                val newMinor =
                   DList.insertBefore (prevMinor,
                                       {label=standardGap * ~inGroup,
                                        major=thisMajor})
                val predEntry = DList.pred thisEntry
              in
                setMinor (thisEntry, newMinor);
                engroup (toDo-1, (inGroup+1) mod standardSize, newMinor, predEnt
ry, majorEntries)
              end
        in
          engroup (minorCount, 0, minorEntry, SOME startEntry, majorEntries)
        end

    fun insertBefore (thisEntry : '1a olist,newContents) =
        let
          val thisMinor = #minor (DList.contents thisEntry)
          val {label=thisMinorLabel,major=thisMajor} = DList.contents thisMinor
          val newMinorLabel =
              case DList.pred thisMinor of
                  SOME predMinor =>
                    let
                      val {label=predMinorLabel,...} = DList.contents predMinor
                      val newMinorLabel =
                          predMinorLabel
                          + (thisMinorLabel - predMinorLabel) div 2
                    in
```

```
                  if newMinorLabel > predMinorLabel then
                    newMinorLabel
                  else
                    let
                      val SOME predEntry = DList.pred thisEntry
                    in
                      regroupBefore predEntry;
                      thisMinorLabel - standardGap
                    end
                end
            | NONE => thisMinorLabel - standardGap
    in
      DList.insertBefore
        (thisEntry,
         {contents=newContents,
          minor=
            DList.insertBefore
              (thisMinor, {label=newMinorLabel, major=thisMajor})})
    end

fun insertManyAfter (entry, nil) = nil
  | insertManyAfter (entry, h :: t) =
    let val newEntry = insertAfter (entry, h)
    in  newEntry :: insertManyAfter (newEntry, t) end

fun insertManyBefore (entry, nil) = nil
  | insertManyBefore (entry, h :: t) =
    let val newEntry = insertBefore (entry, h)
    in  newEntry :: insertManyBefore (newEntry, t) end

fun delete entry =
    let
      val minorEntry = minor entry
    in
      case (DList.pred minorEntry, DList.succ minorEntry) of
        (NONE,NONE) =>
          OList.delete (#major (DList.contents minorEntry))
      | _ =>
          (DList.delete minorEntry ; ()) ;
      #contents (DList.delete entry)
    end


fun compare (leftEntry,rightEntry) =
      let
        val (leftMajor,leftLabel) = majorAndLabel leftEntry
        val (rightMajor,rightLabel) = majorAndLabel rightEntry
      in
        case OList.compare (leftMajor, rightMajor) of
            Equal =>
              if equal (leftEntry,rightEntry) then
                Equal
              else (if leftLabel < rightLabel then Less else Greater)
          | notequal => notequal
      end

fun label entry =
    let val (major, label) = majorAndLabel entry
    in  "<" ^ OList.label major ^ "," ^ makestring label ^ ">" end

fun string doContents =
```

```
        let
            fun doEntry {contents,minor} =
              let
                val {label,major} = DList.contents minor
              in
                "(<" ^ OList.label major ^ "," ^ makestring (label:int)
                 ^ ">," ^ doContents contents ^ ")"
              end
        in
            DList.string doEntry
        end
end
end
```

timestamps.sml

```
structure TimeStamps =
  struct
    structure OList = OrderList2

    type timestamp = unit OList.olist

    val create  : unit -> timestamp = OList.create
    val insertAfter  : timestamp -> timestamp =
        fn x => OList.insertAfter  (x,())
    val insertBefore : timestamp -> timestamp =
        fn x => OList.insertBefore (x,())

    val delete  : timestamp -> unit = OList.delete

    val equal   : timestamp * timestamp -> bool = OList.equal
    val compare : timestamp * timestamp -> LibBase.relation = OList.compare

    (* Functions to tranverse and examine the data structure *)

    val pred    : timestamp -> timestamp option = OList.pred
    val succ    : timestamp -> timestamp option = OList.succ
    val string  : timestamp -> string = OList.string (fn () => "")
    val label   : timestamp -> string = OList.label
  end
```

rewriting-splaytree.sml

```
(* splaytree.sml
 *
 * COPYRIGHT (c) 1993 by AT&T Bell Laboratories.  See COPYRIGHT file for details
.
 *
 * Splay tree structure.
 *
 *)

structure RewritingSplayTree : REWRITING_SPLAYTREE =
  struct
    local open LibBase in
      datatype 'a splay =
        SplayObj of {
          value : 'a,
          right : 'a splay,
          left : 'a splay
```

```
        }
      | SplayNil

      datatype 'a position =
        Before | After | Identical | Delete | Rewrite of 'a splay -> 'a splay

      datatype 'a ans_t =
        No | Eq of 'a | Lt of 'a | Gt of 'a | Rw of ('a splay -> 'a splay) * 'a

      fun lrotate SplayNil = SplayNil
        | lrotate (arg as SplayObj{value,left,right=SplayNil}) = arg
        | lrotate (SplayObj{value,left,right=SplayObj{value=v,left=l,right=r}})
=
            lrotate (SplayObj{value=v,left=SplayObj{value=value,left=left,right=
l},right=r})

      fun join (SplayNil,SplayNil) = SplayNil
        | join (SplayNil,t) = t
        | join (t,SplayNil) = t
        | join (l,r) =
            case lrotate l of
              SplayNil => r      (* impossible as l is not SplayNil *)
            | SplayObj{value,left,right} => SplayObj{value=value,left=left,right
=r}

      fun splay (compf, root) = let
          fun adj SplayNil = (No,SplayNil,SplayNil)
            | adj (arg as SplayObj{value,left,right}) =
                (case compf value of
                  Delete => adj (join (left,right))
                | Identical => (Eq value, left, right)
                | Rewrite rewrite => (Rw (rewrite,value), left, right)
                | After =>
                    let val rec adjleft = fn
                      SplayNil => (Gt value,SplayNil,right)
                    | SplayObj{value=value',left=left',right=right'} =>
                        (case compf value' of
                          Delete => adjleft (join (left',right'))
                        | Identical => (Eq value',left',
                                      SplayObj{value=value,left=right',right=rig
ht})
                        | Rewrite rewrite => (Rw (rewrite,value'), left',
                                      SplayObj{value=value,left=right',right=rig
ht})
                        | After =>
                            (case left' of
                              SplayNil => (Gt value',left',SplayObj{value=value,
left=right',right=right})
                            | _ =>
                              let val (V,L,R) = adj left'
                                  val rchild = SplayObj{value=value,left=right',
right=right}
                              in
                                (V,L,SplayObj{value=value',left=R,right=rchild})
                              end
                            ) (* end case *)
                        | Before =>
                            (case right' of
                              SplayNil => (Lt value',left',SplayObj{value=value,
left=right',right=right})
                            | _ =>
```

```
                            let val (V,L,R) = adj right'
                                  val rchild = SplayObj{value=value,left=R,rig
ht=right}

                                  val lchild = SplayObj{value=value',left=left
',right=L}
                            in
                              (V,lchild,rchild)
                            end
                          ) (* end case *)
                      ) (* end case *)
                    in
                      adjleft left
                    end
                | Before =>
                    let val rec adjright = fn
                      SplayNil => (Lt value,left,SplayNil)
                    | SplayObj{value=value',left=left',right=right'} =>
                        (case compf value' of
                          Delete => adjright (join (left',right'))
                        | Identical =>
                            (Eq value',SplayObj{value=value,left=left,right=left
'},right')
                        | Rewrite rewrite =>
                            (Rw (rewrite,value'),SplayObj{value=value,left=left,
right=left'},right')
                        | Before =>
                            (case right' of
                              SplayNil => (Lt value',SplayObj{value=value,left=l
eft,right=left'},right')
                            | _ =>
                              let val (V,L,R) = adj right'
                                  val lchild = SplayObj{value=value,left=left,ri
ght=left'}

                              in
                                (V,SplayObj{value=value',left=lchild,right=L},R)
                              end
                            ) (* end case *)
                        | After =>
                            (case left' of
                              SplayNil => (Gt value',SplayObj{value=value,left=l
eft,right=left'},right')
                            | _ =>
                              let val (V,L,R) = adj left'
                                  val rchild = SplayObj{value=value',left=R,righ
t=right'}

                                  val lchild = SplayObj{value=value,left=left,ri
ght=L}

                              in
                                (V,lchild,rchild)
                              end
                            ) (* end case *)
                        ) (* end case *)
                    in
                        adjright right
                    end
                ) (* end case *)
          fun rewrite root =
              case adj root of
                  (No,_,_)   => (Greater,SplayNil)
                | (Eq v,l,r) => (Equal,SplayObj{value=v,left=l,right=r})
                | (Lt v,l,r) => (Less,SplayObj{value=v,left=l,right=r})
```

```
                | (Gt v,l,r) => (Greater,SplayObj{value=v,left=l,right=r})
                | (Rw (f,v),l,r) => rewrite (f (SplayObj{value=v,left=l,right=r}
))
        in
          rewrite root
        end

    end
  end (* SplayTree *)
```

rewriting-splayaccess.sml

```
(* splaydict.sml
 *
 * Based on splay-dict.sml from the SML/NJ Library.
 *
 * COPYRIGHT (c) 1993 by AT&T Bell Laboratories.
 * See COPYRIGHT file for details.
 *
 * Functor implementing a lookup table using splay trees.
 *
 *)

structure RewritingSplayAccess : REWRITING_SPLAYACCESS =
  struct
    structure RSTree = RewritingSplayTree
    open RSTree LibBase

    exception NotFound

    val empty = SplayNil

  (* Insert an item.
    *)
    fun insert (root,cmp,v) =
        case splay (cmp, root) of
            (_,SplayNil) =>
              SplayObj {value=v, left=SplayNil, right=SplayNil}
          | (Equal, SplayObj {value,left,right}) =>
              SplayObj {value=v,left=left,right=right}
          | (Less,  SplayObj {value,left,right}) =>
              SplayObj {value=v,
                        left=SplayObj {value=value,
                                       left=left,
                                       right=SplayNil},
                        right=right}
          | (Greater,SplayObj{value,left,right}) =>
              SplayObj {value=v,
                        left=left,
                        right=SplayObj {value=value,
                                        left=SplayNil,
                                        right=right}}

  (* Find an item, raising NotFound if not found
    *)

    fun find (root, cmp) =
        case splay (cmp, root) of
            (_,r as SplayNil) => (r, NONE)
```

```
      | (Equal, r as SplayObj{value,...}) => (r, SOME value)
      | (_, r) => (r, NONE)

fun findbefore (root, cmp) =
    case splay (cmp, root) of
        (_, r as SplayNil) => (r,NONE)
      | (Greater, root') =>
          (case splay (cmp, root') of
              (_, r as SplayNil) => (r,NONE)
            | (Greater, r) => (r,NONE)
            | (_ ,r as SplayObj{value,...}) => (r, SOME value))
      | (_, r as SplayObj{value,...}) => (r, SOME value)

fun findafter (root, cmp) =
    case splay (cmp, root) of
        (_, r as SplayNil) => (r,NONE)
      | (Less, root') =>
          (case splay (cmp, root') of
              (_, r as SplayNil) => (r,NONE)
            | (Less, r) => (r,NONE)
            | (_ ,r as SplayObj{value,...}) => (r, SOME value))
      | (_ ,r as SplayObj{value,...}) => (r, SOME value)

fun afteronly cmp arg =
    case (cmp arg) of
        Identical => Before
      | it => it

fun beforeonly cmp arg =
    case (cmp arg) of
        Identical => After
      | it => it

(* Remove an item.
* Raise NotFound if not found
*)
fun remove (root, cmp) =
    case (splay (cmp, root)) of
        (Equal,SplayObj{value,left,right}) =>
            (join (left,right), SOME value)
      | (_,r) => (r, NONE)


fun root SplayNil = NONE
  | root (SplayObj {value, ...}) = SOME value


    (* Return a list of the items (and their keys) in the dictionary *)
fun listItems root =
    let fun apply (SplayNil,l) = l
          | apply (SplayObj{value,left,right},l) =
              apply(left, value::(apply (right,l)))
    in
      apply (root,[])
    end

    (* Apply a function to the entries of the dictionary *)
fun app af root =
      let fun apply SplayNil = ()
            | apply (SplayObj{value,left,right}) =
                (apply left; af value; apply right)
```

```
        in
          apply (root)
        end

    fun revapp af root =
          let fun apply SplayNil = ()
                | apply (SplayObj{value,left,right}) =
                    (apply right; af value; apply left)
        in
          apply (root)
        end

        (* Fold function *)
    fun fold abf (root,b) =
          let fun apply (SplayNil, b) = b
                | apply (SplayObj{value,left,right},b) =
                    apply(left,abf(value,apply(right,b)))
        in
          apply (root,b)
        end

    fun revfold abf (root,b) =
          let fun apply (SplayNil, b) = b
                | apply (SplayObj{value,left,right},b) =
                    apply(right,abf(value,apply(left,b)))
        in
          apply (root,b)
        end

    fun string doContents root =
        let
          fun whiz SplayNil = ""
            | whiz (SplayObj{value,left,right}) =
              ( "[" ^
                  whiz left ^
                    "(" ^ doContents value ^ ")" ^
                  whiz right ^
                "]" )
        in
            "!" ^ whiz root ^ "!"
        end

  end
```

history.sml

```
(*
functor History ( structure TimeStamps : TimeStamps
                  structure RewritingSplayAccess : REWRITING_SPLAYACCESS
                      )  : HISTORY  =
*)
structure ElementHistory : HISTORY =
  struct
    structure Time = TimeStamps
    structure Splay = RewritingSplayAccess
    local
      open Splay LibBase
    in
      type timestamp = Time.timestamp
      type 'a entry  = {tag:timestamp, value:'a}
```

```
type 'a history   = 'a entry Splay.splay

exception Unbound

fun entry_string doValue ({tag,value} : 'a entry) =
    Time.label tag ^ ": " ^ doValue value

fun history_string doValue =
    Splay.string (entry_string doValue)

val time_string = Time.string

fun cmp this ({tag,...} : 'a entry) : 'a entry position =
    case Time.compare (tag, this) of
        Less => Before
      | Greater => After
      | Equal => Identical

fun create () =
    let
      val time = Time.create ()
      val position = cmp time
    in
      (fn value =>
        Splay.insert(empty,position,{tag=time,value=value} : 'a entry),
       time)
    end

fun locate (instances,time) =
    case Splay.findbefore (instances, cmp time) of
        (instances', SOME it) => (instances', #value it)
      | (_,NONE) => raise Impossible "Nodes missing in history (read)"

fun record ((entries,thisTime),value) =
    let
      val succTimeOpt = Time.succ thisTime  (* Order matters here *)
      val newTime     = Time.insertAfter thisTime
      val instances'' =
        case succTimeOpt
        of  NONE => entries
          | SOME succTime =>
            let
              val findSucc = cmp succTime
            (* val _ = print (time_string time ^ "\n"); *)
            in
              case Splay.findbefore (entries, findSucc)
              of  (_,NONE) =>
                    raise Impossible "Nodes missing in history (write)"
                | (instances', SOME {value=priorValue,tag=priorTime}) =>
                  ( case Time.compare (priorTime, succTime)
                    of  Equal => instances'
                      | Less =>
                        Splay.insert
                          ( instances',
                            findSucc,
                            {tag=succTime,value=priorValue} )
                      | Greater =>
                          raise Impossible "Earlier time is later!" )
            end
    in
      (insert (instances'', cmp newTime, {tag=newTime,value=value}),
```

```
            newTime)
          end

    end
  end
```

impure-array.sml

```
structure ImpureArray : ARRAY =
    struct
        structure Array = Array

        open Array
        fun update(a,i,v) = (Array.update(a,i,v) ; a)
        val size = length
        val method = "Mutable Arrays"
    end
```

func-array.sml

```
(* functor FunctionalArray ( ElementHistory : HISTORY ) : ARRAY = *)
structure FunctionalArray : ARRAY =
  struct
    structure History = ElementHistory
    structure IArray  = ImpureArray

    local
      open History
    in
      exception Size = IArray.Size;
      exception Subscript = IArray.Subscript;

      type 'a array = {iarray:'a history IArray.array, ctime:timestamp}

      fun tabulate (size,init) =
          let
            val (make_history, ctime) = create ()
          in
            { iarray =
                IArray.tabulate (size, fn index => make_history (init index)),
              ctime = ctime }
          end

      fun sub ({iarray,ctime},index) =
          let
             val history = IArray.sub (iarray,index)
             val (history', value) = locate (history,ctime);
          in
             IArray.update (iarray,index,history');
             value
          end

      fun update ({iarray,ctime},index,value) =
          let
             val history = IArray.sub (iarray,index)
             val (history', ctime') = record ((history,ctime),value);
          in
             IArray.update (iarray,index,history');
             {iarray=iarray, ctime=ctime'}
          end
```

```
      fun size ({iarray, ...} : '1a array) = IArray.size iarray

      val method = "Element Histories"
    end
  end
```

A.1.3 Implementations of Other Array Techniques

tree-array.sml

```
structure TreeArray (* : ARRAY *) =
    struct
        exception Size = ImpureArray.Size;
        exception Subscript = ImpureArray.Subscript;

        open Bits
        fun odd x = System.Unsafe.cast (andb (x,1)) : bool;

        datatype 'a tree =
            Branch of 'a tree * 'a tree
          | Leaf of 'a
          | Bud
        and 'a array = Array of 'a tree * int

        fun update (Array (tree, size), index, value) =
            let
                fun update' (Leaf _, 0) = Leaf value
                  | update' (Branch (left,right), index) =
                    if odd index then
                        Branch (left,update'(right,rshift(index,1)))
                    else
                        Branch (update'(left,rshift(index,1)),right)
                  | update' _ = raise Subscript
            in
                Array(update'(tree,index),size)
            end

        fun sub (Array(tree, size), index) =
            let
                fun sub' (Leaf value, 0) = value
                  | sub' (Branch (left,right), index) =
                        if odd index then
                            sub'(right,rshift(index,1))
                        else
                            sub'(left,rshift(index,1))
                  | sub' _ = raise Subscript
            in
                sub' (tree, index)
            end

        fun tabulate (size,init) =
            let
                fun tabulate' (n, level) =
                    if n >= size then Bud
                    else
                        let
                            val level' = lshift(level,1)
                        in
                            if level < size then
```

```
                          Branch (tabulate'(n,level'),
                                  tabulate'(n+level,level'))
                      else
                          Leaf (init n)
                  end
            in
                Array(tabulate' (0,1),size)
            end

        fun size (Array(tree,size)) = size

        val method = "Trees"
    end
```

trailer-array.sml

```
(*
 * Trailer (and rerooting) Based Arrays
 *)

structure TrailerArray (* : ARRAY *) =
    struct
        structure ImpureArray = ImpureArray

        exception Size = ImpureArray.Size;
        exception Subscript = ImpureArray.Subscript;

        datatype 'a array =
            Array of 'a array_node ref
        and          'a array_node =
            Root of 'a ImpureArray.array
          | Modification of int * 'a * 'a array

        fun tabulate (size, init) =
            Array (ref (Root (ImpureArray.tabulate (size,init))))

        fun reroot (root  as ref (Root array)) = array
          | reroot (child as ref (Modification (index,value,Array parent))) =
            let
               (* val _ = print "." *)
                val array  = reroot parent
                val value' = ImpureArray.sub(array,index)
            in
                child  := Root (ImpureArray.update(array,index,value));
                parent := Modification (index, value', Array child);
                array
            end

        fun update (Array parent,index,value) =
            let
                val array  = reroot parent;
                val value' = ImpureArray.sub(array,index)
                val result =
                    Array (ref (Root (ImpureArray.update(array,index,value))))
            in
                parent := Modification (index, value', result);
                result
            end

        fun sub (Array array,index) =
```

```
            ImpureArray.sub(reroot array,index)

        fun size (Array array) =
            ImpureArray.size(reroot array)

        val method = "Trailers"
    end
```

copying-array.sml

```
(*
 * Naive copying implementation.
 *)

structure NaiveArray (* : ARRAY *) =
    struct
        structure Vector = Vector

        type 'a array = 'a Vector.vector
        open Vector
        fun update(array,index,value) =
            let
                fun copy i =
                    if i = index then value
                    else sub(array,i)
                val size = length array
            in
                if index < size then
                    tabulate (size,copy)
                else
                    raise Vector.Subscript
            end
        val size = length
        val method = "Naive Copying"
    end
```

A.1.4 Functions used for Performance Timing

timer.sml

```
(*
 * Comparison of array implementation methods.
 *)

(*
 * SML/NJ specific tweaks.
 *)

val gc = System.Unsafe.CInterface.gc
val _  = System.Control.Runtime.softmax := 8 * 1024 * 1024
val _  = System.Control.Runtime.ratio := 32
val _  = System.Control.Print.printDepth := 64
local
    open System.Timer

in
    val ticks_per_second = 1.0 / 50.0
    fun round accuracy x = real (floor (x / accuracy + 0.5)) * accuracy
    fun sci_round acc 0.0 = 0.0
      | sci_round acc x = let val pow = exp (real (floor (ln (abs x) / ln 10.0 +
```

```
 1.0)) * ln 10.0)
                         in  (round acc (x / pow)) * pow end
    fun seconds (TIME {sec:int,usec:int}) =
        round ticks_per_second (real sec + (real usec / 1000000.0))

  fun time1 action =
      let
          val timer  = start_timer ()
          val result = action ()
          val utime  = check_timer timer
          val stime  = check_timer_sys timer
          val gctime = check_timer_gc timer
      in
        (result, seconds utime, seconds stime, seconds gctime)
      end
  fun timeN (n,action) =
      let
          fun worker 1 = action ()
            | worker n = (action () ; worker (n-1))
          fun newAction () = worker n
          val (result,usecs, ssecs, gcsecs) = time1 newAction
          val rn = real n
          fun scale x = round 2.5E~7 (x / rn)
      in
          (result, scale usecs, scale ssecs, scale gcsecs)
      end
  val overhead = #2 (timeN (100000, fn () => 1))

  fun time action =
      let
        val _ = outputc std_out "% Timing... ("
        fun timing n =
          let
            val _ = outputc std_out (Integer.makestring n);
            val _ = flush_out std_out;
            val results as (result,usecs, ssecs, gcsecs) = timeN (n,action)
            val rn = real n
            val _ = (print "["; print usecs; print "]"; flush_out std_out)
            val tsecs = (usecs * rn) - (ticks_per_second / 2.0)
          in
            if tsecs <= 0.0 then (outputc std_out ","; timing (n * 500))
            else (
            if tsecs < 4.0 then
              (outputc std_out ",";
               timing (ceiling (rn * 5.0 / tsecs)))
            else
              let val clip =
                    sci_round (sci_round 0.1 (ticks_per_second / tsecs))
              in (result, clip (usecs - (round ticks_per_second (rn * overhead)
/ rn)), clip ssecs, clip gcsecs)
              end )
          end
        val (result,usecs, ssecs, gcsecs) = timing 1
        val _ = outputc std_out ")\n"
      in
        outputc std_out
              ("%   User Time:       "
           ^ (Real.makestring usecs)
           ^ "\n%   System time      "
           ^ (Real.makestring ssecs)
           ^ "\n%   Time Collecting: "
```

```
          ^ (Real.makestring gcsecs)
          ^ "\n") ;
        (result, usecs)
      end
end
```

A.1.5 Array Test Code

Note that test1, test2 *and* test3 *correspond to the three implementations of reverse. The function* test5 *corresponds to the multi-version test.*

tests.sml

```
functor ArrayTestsFun ( Array : ARRAY ) =
  struct
    structure Array = Array

    local
      open Array
    in
      fun tolist array =
          let
              val size = size array
              fun worker x =
                  if x = size then nil
                  else
                      sub(array,x) :: worker (x+1)
          in
              worker 0
          end

      fun sum array =
          let
              val size = size array
              fun worker x =
                  if x = size then 0
                  else
                      sub(array,x) + worker (x+1)
          in
              worker 0
          end

      fun zero array =
          let
              val size = size array
              fun worker (x,array) =
                  if x = size then ()
                  else
                      worker (x+1,update(array,x,0))
          in
              worker (0,array)
          end

      fun max (array1, array2) =
          let
              val size = size array1
              fun worker (array,index) =
                  if index = size then array
                  else
                      let
```

```
                    val x = sub(array1, index)
                    and y = sub(array2, index)
                in
                    worker
                        ( if Integer.< (x,y) then
                            update(array, index, y)
                        else
                            array
                        , index + 1 )
                end
    in
        worker (array1,0)
    end

fun print_array printer array =
    let
        val size = size array
        fun worker x =
          ( print "{";
            if x = size then print "}"
            else
              ( printer (sub(array,x));
                worker' (x+1) ) )
        and worker' x =
          ( if x = size then print "}"
            else
              ( print ", ";
                printer (sub(array,x));
                worker' (x+1) ) )
    in
        worker 0
    end


fun create n = Array.tabulate (n, fn x => x)

(*
 * A functional implementation of an imperative reverse algorithm,
 * array techniques optimized for single threaded access should
 * have no problem providing O(n) operation.
 *)

fun reverse1 original =
    let
        val size = size original
        val half = size div 2
        fun rev (current,count) =
            if count < half then
                let
                    val count'    = count + 1
                    val left      = sub(current, count)
                    val right     = sub(current, size - count')
                    val current'  = update(current, count, right)
                    val current'' = update(current', size - count', left)
                in
                    rev (current'', count')
                end
            else current
    in
        rev (original,0)
    end
```

```
(*
 * A more functional reverse algorithm.
 *)

 fun reverse2 original =
     let
       val size = size original
       fun rev (current,count) =
           if count < size then
             let
               val count'  = count + 1
               val current' =
                   update(current, count, sub(original, size - count'))
             in
               rev (current', count')
             end
           else current
     in
       rev (original,0)
     end

(*
 * The monolithic approach.
 *)

 fun reverse3 original =
     let
       val size = size original
       fun reverse_read index =
           sub(original,size - (index+1))
     in
       tabulate (size, reverse_read)
     end



 fun test4 n =
     let
         val _ = gc 1
         val _ = app print ["\nTesting ",method,", n=",makestring n,"\ncrea
te(n) :\n"]
         val (x,xt) = time (fn _ => create n);
         val _ = print "reverse1 (create(n)) :\n"
         val (y,yt) = time (fn _ => reverse1 (create(n)));
         val _ = print "let val x = create n in max (x, reverse1(x)) end :\
n"
         val (z,zt) = time (fn _ => let val x = create n in max (x, reverse
1(x)) end)
     in
         (n,(xt,yt - xt,zt - yt))
     end

 fun test_rev reverse n =
     let
         val _ = gc 1
         val _ = app print ["\nTesting ",method,", n=",makestring n,"\nReve
rse:\n"]
         val (_,t0) = time (fn _ => create n)
         val (_,t) = time (fn _ => reverse (create n));
     in
```

```
            (n,t - t0)
          end

      val test1 = test_rev reverse1

      val test2 = test_rev reverse2

      val test3 = test_rev reverse3

      fun random seed =
          let
             val randomGenerator = Random.mkRandom seed
          in
             fn (x,y) => Random.range (x,y) (randomGenerator ())
          end

      fun versions (n,m) =
          let
            val initial  = create n
            val versions = ImpureArray.tabulate (m,fn x => initial)
            val rnd = random 1.0
            fun versionize x =
                if x >= (m-1) then () else
                  let
                    val v = rnd (0,x)
                    val e = rnd (0,n-1)
(*                  val _ = outputc std_out ("v" ^ Integer.makestring (x+1) ^ "
<- increment v" ^ Integer.makestring v ^ "[" ^ Integer.makestring e ^ "]\n")
*)                  val version = ImpureArray.sub (versions,v)
                    val newVersion = Array.update (version, e, Array.sub (versio
n, e) + 1)
                  in
                    ImpureArray.update(versions, x+1, newVersion);
                    versionize (x+1)
                  end
          in
            versionize 0 ; initial
          end

      fun bredth_prep (n,mv) =
          let
            val initial  = create n
            val versions = ImpureArray.tabulate (mv,fn x => initial)
            val rnd = random 1.0
            val mv_last = mv - 1
            fun breed x =
                if x < mv_last then
                  let
                    val v = rnd (0,x)
                    val e = rnd (0,n-1)
(*                  val _ = outputc std_out ("\^Mv" ^ Integer.makestring (x+1) ^
 " <- increment v" ^ Integer.makestring v ^ "[" ^ Integer.makestring e ^ "]")
*)                  val _ = flush_out std_out;
                    val version = ImpureArray.sub (versions,v)
                    val newVersion = version
                  in
                    ImpureArray.update(versions, x+1, newVersion);
                    breed (x+1)
                  end
                else ()
          in
```

```
          breed 0 ; versions
        end

    fun bredth (n,mv) =
        let
          val initial  = create n
          val versions = ImpureArray.tabulate (mv,fn x => initial)
          val rnd = random 1.0
          val mv_last = mv - 1
          fun breed x =
              if x < mv_last then
                let
                  val v = rnd (0,x)
                  val e = rnd (0,n-1)
(*                val _ = outputc std_out ("\^Mv" ^ Integer.makestring (x+1) ^
 " <- increment v" ^ Integer.makestring v ^ "[" ^ Integer.makestring e ^ "]")
*)                val _ = flush_out std_out;
                  val version = ImpureArray.sub (versions,v)
                  val newVersion = Array.update (version, e, Array.sub (versio
n, e) + 1)
                in
                  ImpureArray.update(versions, x+1, newVersion);
                  breed (x+1)
                end
              else ()
        in
          breed 0 ; versions
        end

   fun test5 n mv =
       let
            val _ = gc 1
            val _ = app print ["\nTesting Bredth, under ",method,", n=",makest
ring n, ", v=",makestring mv,"... \n"]
            val (_,secs0) = time (fn _ => bredth_prep (n,mv));
            val (_,secs) = time (fn _ => bredth (n,mv));
        in
            (mv,secs - secs0)
        end

    end
  end


structure TAT = ArrayTestsFun ( TrailerArray )
structure FAT = ArrayTestsFun ( FunctionalArray )
structure BAT = ArrayTestsFun ( TreeArray )
structure CAT = ArrayTestsFun ( NaiveArray )
structure IAT = ArrayTestsFun ( ImpureArray )
```

Bibliography

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional languages. *BIT*, 28(3):490–503, 1988.

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

[3] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Functional Programming Languages and Computer Architecture, London*. ACM, Septemeber 1989.

[4] F. Warren Burton and Hsi-Kai Yang. Manipulating multilinked data structures in a pure functional language. *Software – Practice and Experience*, 20(11):1167–1185, November 1990.

[5] L. Damas and R. Milner. Principle type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, N.M., January 1982.

[6] O. Danvy and A. Filinski. Abstracting control. In *Conference on Lisp and Functional Programming*, pages 717–740, Nice, France, 1972.

[7] Paul F. Dietz and Daniel D. Sleator. 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.

[8] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[9] R. Hindley. Principle type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[10] Paul Hudack, Simon L. Peyton Jones, Philip L. Wadler, Arvind, B. Bontel, J. Fairbairn, J. Fasel, M. Guzman, Kevin Hammond, John Hughs, T. Johnson, R Kieburtz, W. Partain, and J. Peterson. Report on the functional programming language Haskell. version 1.2. *SIGPLAN Notices*, 27, July 1992.

[11] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, U.S.A., January 1985. ACM.

[12] H. S. Huitema and M. J. Plasmeijer. The Concurrent Clean system users manual, version 0.8. Technical Report Technical Report 92-19, University of Nijmegen, August 1992.

[13] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University, December 1993.

[14] Henry G. Baker Jr. Shallow binding in lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

[15] Henry G. Baker Jr. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, August 1991.

[16] David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing. Springer Verlag, 1993.

[17] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[18] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mess., 1990.

[20] Melissa E. O'Neill. PhD Thesis. *In preparation*. Chapters covering functional array optimisations, garbage collection and parallel update for functional arrays, are available on request.

[21] J. Reynolds. Definitional interperters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, June 1990.

[22] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7:299–310, 1985.

[23] Robert Segewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[24] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[25] W. F. Tichy. Design, implementation and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, September 1982. IEEE.

[26] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[27] David A Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[28] P. L. Wadler. Comprehending monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Nice*. ACM, June 1990.

[29] P. L. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

[30] P. L. Wadler. The essence of functional programming. In *Conference record of the 19th annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, U.S.A.* ACM, January 1992.