Version Stamps for Functional Arrays and Determinacy Checking: Two Applications of Ordered Lists for Advanced Programming Languages

by

Melissa Elizabeth O'Neill

B.Sc., University of East Anglia, 1990

M.Sc., Simon Fraser University, 1994

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in the
School of Computing Science

Simon Fraser University
30 November 2000

Approval

Name:	Melissa Elizabeth O'Neill	
Degree:	Doctor of Philosophy	
Title of Dissertation:	Version Stamps for Functional Arrays and Determinacy Checking: Two Applications of Ordered Lists for Advanced Programming Languages	
Examining Committee:	Dr. Robert D. Cameron	Chair
	Dr. F. Warren Burton	Senior Supervisor
	Dr. M. Stella Atkins	Supervisor
	Dr. Arvind Gupta	Supervisor
	Dr. Binay Bhattacharya	SFU Examiner
	Dr. Christopher Okasaki Assistant Professor, Dept of Computer Science, Columbia University	External Examiner
Date Approved:	30 NOVEMBER 2000	

Abstract

This dissertation describes the fat-elements method for providing functional arrays and the LR-tags method for determinacy checking. Although these topics may seem very different, they are actually closely linked: Both methods provide reproducibility in advanced programming languages and share many implementation details, such as tagging data using version stamps taken from an ordered list.

The fat-elements method provides arrays as a true functional analogue of imperative arrays with the properties that functional programmers expect from data structures. It avoids many of the drawbacks of previous approaches to the problem, which typically sacrifice usability for performance or vice versa.

The fat-elements method efficiently supports array algorithms from the imperative world by providing constant-time operations for single-threaded array use. Fully persistent array accesses may also be performed in constant amortized time if the algorithm satisfies a simple requirement for uniformity of access. For algorithms that do not access the array uniformly or single-threadedly, array reads or updates take at most $O(\log n)$ amortized time for an array of size n. The method is also space efficient—creating a new array version by updating a single array element requires constant amortized space.

The LR-tags method is a technique for detecting indeterminacy in asynchronous parallel programs—such as those using nested parallelism on shared-memory MIMD machines—by checking Bernstein's conditions, which prevent determinacy races by avoiding write/write and read/write contention between parallel tasks. Enforcing such conditions at compile time is difficult for general parallel programs. Previous attempts to enforce the conditions at runtime have had non–constant-factor overheads, sometimes coupled with serial-execution requirements.

The LR-tags method can check Bernstein's (or Steele's generalized) conditions at runtime while avoiding some of the drawbacks present in previous approaches to this problem. The method has constant-factor space and time overheads when run on a uniprocessor machine and runs in parallel on multiprocessor machines, whereas the best previous solution, Cilk's Nondeterminator (a constant-space and nearly constant-time approach), requires serial execution.

Both parts of the dissertation include theoretical and practical material. Tests from implementations of both techniques indicate that the methods should be quite usable in practice.

In memory of Wallace Clawpaws

Acknowledgments

Researching and writing a doctoral dissertation is a daunting venture. My endeavor has been made much easier by those around me who have been willing to offer advice, assistance, and practical help. Warren Burton's contribution to my research has been invaluable. Under Warren's tutelage, I have felt able to explore, to take the time to examine research avenues that looked interesting, and, above all, to enjoy the research process. Had I had a different senior supervisor, I am sure I would have produced a solid thesis, but I am equally sure that I would not have discovered the parallels between functional arrays and determinacy checking.

The other members of my supervisory committee, Arvind Gupta and Stella Atkins, have also been helpful. In particular, Arvind's recommendation of *Graph Drawing: Algorithms for the Visualization of Graphs* (Di Battista et al., 1999) saved me from needlessly proving well-known results.

Interactions with other members of SFU's academic community have also proved fruitful. For example, a chance conversation with John Borwein revealed the existence of Jenson's Inequality, saving me a good deal of effort searching the literature for a proof of a result that I considered "obvious" but that Warren rightly believed needed a formal underpinning for more wary readers.

I have also received help in my endeavours from beyond the university. Emil Sarpa and Bob McCartney of Sun Microsystems arranged access to a highly parallel Sun Enterprise 10000 "Starfire" valued at more than a million dollars. My special thanks go to Emerich R. Winkler, Jr, Network Systems Engineer at the Burnaby Field Office of Sun Microsystems of Canada, who acted as my local contact, allowing me to visit his office

to access the machine and acting as "job jockey" via email, running tests and sending results back to me.

I am also indebted to numerous free-software projects that have provided me with tools and inspiration. Most notably, the CIL project furnished the benchmark code used in Chapter 11, but my work would have been much more arduous without other free software projects, including Standard ML, Haskell, GCC, and Perl, among many others.

Finally, moving to those closest to me, Claire Connelly has performed sterling service as unpaid copy editor (any typographical errors or awkward constructions remaining are my fault alone). She has also patiently listened to me ramble on about one research topic or another on countless occasions, without ever demanding that I keep my nonsense to myself. And I could hardly conclude without mentioning my parents' abiding encouragement for all my endeavours—especially their interest in exactly when I was going to finish my dissertation, which has been a constant reminder of the importance of timeliness in publication.

Contents

Appendices 185

Figures

Tables

Code Listings

Preface

A doctoral dissertation is many things: an academic discourse, an exhibition of research for an examining body, and, of course, a personal milestone. My dissertation is both an academic work, describing my contributions to the fields of functional arrays and determinacy checking, and a narrative, recounting how research I began in one field lead me to another.

When I began my doctoral work, I was fairly sure that my dissertation would be about functional arrays—I had scarcely given any thought to the idea of parallel determinacy checking. But as my research progressed, I began to look first at the parallel use of functional arrays and then at the parallel use of memory in general. As I did, I discovered that some of the core techniques I had developed to implement functional arrays could be employed in determinacy checking.

In retrospect, the connections between the topics seem obvious. Both address non–single-threaded access to data, both deal with preserving reproducibility, and my solutions to both problems involve using ordered lists to provide version stamps that chronicle data accesses.

Nevertheless, in an attempt to make this dissertation as accessible as possible to readers from both the functional-programming and parallel-programming communities, I have broken it into two distinct parts, the first discussing functional arrays; the second, determinacy checking. A discussion of the relationship between the two fields is available in Appendix A.

However you choose to read this dissertation (and there are many ways to do so!), I hope you will learn as much from reading it as I did during the journey.

Overview

In this overview, I will outline the content of my dissertation, providing some brief comments about the content of each chapter and touching on the important new contributions made.

The dissertation consists of two main parts and a set of appendices. The first part discusses functional arrays. The second discusses determinacy checking for programs that use nested parallelism (and some other, related, kinds of parallelism).

Functional Arrays

Chapter 1 reviews previous research on functional arrays. This chapter draws heavily on the review present in my master's thesis, and also on the review paper presented for the depth-examination component of my doctoral study. It does not present any new research, and can probably be skipped over or skimmed through by someone familiar with prior work in the area of functional arrays.

Chapter 2 describes the fat-elements method for representing functional arrays. Since this method is an improvement on the array representation I presented in my master's thesis, the information contained in the first two sections will be familiar to those who have read that earlier work. The concept of *array splitting* introduced in the remainder of this chapter is a new idea that forms part of my doctoral research. A preliminary version of this chapter was published as "A New Method for Functional Arrays" in the *Journal of Functional Programming* (O'Neill & Burton, 1997) so that my research results could be disseminated in a timely manner.

Chapter 3 discusses both the theoretical and actual performance of the fat-elements

method. Like Chapter 2, an early version of this chapter was included in "A New Method for Functional Arrays" to provide others in the research community with an opportunity to evaluate my work.

Chapter 4 explores how persistent arrays interact with the garbage-collection mechanisms typically present in functional languages (a topic that is, in my opinion, underexplored in the literature). It shows how fat-element arrays fail to fit the assumptions made by typical garbage collectors (a property they share with several other functional-array and persistent–data-structure mechanisms), and how this issue can be addressed so that fat-element arrays may be garbage collected efficiently.

Chapter 5 discusses a few of the possible optimizations for improving the real-world performance of the fat-elements method without affecting its theoretical properties.

Determinacy Checking

Chapter 6 reviews previous research on the topic of determinacy checking for parallel programs that use nested parallelism. This review draws heavily on the review paper I presented for the depth-examination component of my doctoral study

Chapter 7 introduces the LR-tags method for determinacy checking, which involves tagging data so that we may rapidly determine whether access to that data satisfies or violates Bernstein's (or Steele's) conditions for determinacy.

Chapter 8 proves the correctness of the algorithm used to allocate tags to new tasks given in Chapter 7.

Chapter 9 shows that the LR-tags method has good time and space complexity for both serial and parallel execution of the algorithm.

Chapter 10 discusses some of the optimizations that can be made to improve the real-world performance of the LR-tags method without affecting its asymptotic performance.

Chapter 11 examines the performance of a C++ implementation of the LR-tags method running standard benchmarks. The results from running the code on a highly parallel Sun Enterprise 10000 system show that the algorithm scales well.

Chapter 12 provides a conclusion to both this part and the body of the dissertation as a whole.

Appendices

Much of the body of the dissertation relies on techniques that may not be well known or obvious to the average reader familiar with either functional arrays or determinacy checking. The appendices discuss some of these techniques.

Some of the techniques discussed in these appendices do not appear to have been published elsewhere. The Cotton/Pthieves multithreaded programming system discussed in Appendices B and C, for example, was written simply to allow me to benchmark my determinacy-checking code using off-the-shelf benchmarks from the Cilk system, and mostly uses well-known ideas for performing work-stealing scheduling. Some of the ideas underlying this system may be novel, however.

In addition, the appendices contain some work is more of historical than practical interest. Appendix G outlines techniques that may allow several fat-element array versions to be merged into a single array, a feature that could have some use in parallel programs using arrays. Although, ultimately, I was not satisfied with the usefulness of the merge operation, this work is interesting at the very least because it is what led me into my work on determinacy checking. Similarly, Appendix F discusses my first attempt at a determinacy-checking scheme. While the results presented in Appendix F have been superseded by the chapters presented in the body of the dissertation, this work is interesting because it admits Feng and Leiserson's (1997) determinacy-checking technique as a specialization.

Notation

Before I begin, let me take a moment to explain some of the notation used throughout this dissertation.

I have been slightly sloppy in my use of asymptotic notation. Usually, we use the notation $g(n) \in \mathrm{O}\big(f(n)\big)$ to indicate that

$$\forall n > L : g(n) \leq Cf(n)$$

where L and C are positive constants. In this dissertation, I will also write $x \in \mathrm{O}\big(f(n)\big)$ as a shorthand for saying that x is a variable that depends on n, such that

$$\exists g(n) : \forall n > L : \big(x = g(n)\big) \wedge \big(g(n) \leq Cf(n)\big).$$

Also, to make stating upper bounds easier, I define the function $\log_+$ as follows:

$$\log_+ n \equiv \log_2 2n.$$

This function has the advantage of being positive for positive integers (thus $\log_+ 1 = 1$ whereas a conventional $\log_2 1 = 0$).

In situations where there are a number of constants to define, I have adopted the practice of writing $\mathrm{k_m}$, where the letter k indicates a constant and the subscript is a letter that, hopefully, has some worth mnemonically.

Part I

Functional Arrays

Chapter 1

Background for Functional Arrays

Programmers writing in traditional programming languages, such as C, have a significant advantage over those writing in less traditional programming languages. As the longest-standing programming paradigm, serial imperative programming has a legacy of algorithms, data structures, and programming techniques, many of which do not transpose well into the domains of other programming paradigms. In my master's thesis (O'Neill, 1994), I investigated the problem of arrays in functional languages; in this doctoral dissertation, I continue that work, further developing my earlier techniques and investigating techniques for parallel array use.

This chapter sets the context for the research that forms the first part of my dissertation. It is based on the review written for my master's thesis, with revisions and elaborations covering advances in the field. A preliminary and abridged version of this chapter has already been made available to the functional-programming community (O'Neill & Burton, 1997) to provide timely access to my results.

1.1 Functional versus Imperative

Imperative and functional languages are fundamentally different—imperative style relies on *assignment*, whereas functional style avoids it. Underlying assignment and the imperative style is the idea that values reside in storage locations. When a value is assigned to a storage location, any previous value at that location is overwritten by the

new value: thus, assignment is also commonly called *destructive update*. Depending on context, a variable in an imperative language either refers to a storage location (the variable's *l-value*) or to the value currently residing in that storage location (the variable's *r-value*). Since assignment can be performed more than once on each storage location, with different values each time, a variable can have a different r-value at different points in a program's execution.

Functional languages, on the other hand, do not view programming in terms of storage locations, but in terms of values. Values are manipulated directly and how their storage is handled is implicit and incidental. Variables in a functional language differ from their counterparts in imperative languages in that their role is purely denotational: The lambda-bound or let-bound variables of functional languages simply denote a particular value, albeit one that may be unknown until runtime.

Traditional imperative programming languages often blur the distinction between storage locations and values, especially when it comes to aggregate structures such as arrays, records and objects, so we will clarify the difference here: Values are immutable, whereas storage locations tend to be seen as a chunks 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.[1] Similarly, since variables in functional languages represent values, not storage locations, 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 equivalent to evaluating "$(3 := 2)$". Likewise, because functional languages do not have aggregates of storage locations, only aggregates of values, we cannot redefine or modify the components of a compound value—we can only create new values, possibly derived from existing ones.

The differences between imperative and functional languages not only encourage programmers to adopt different coding styles (see Figure 1.1), but also impose differences in the data structures they use. To describe the ramifications of these differences, we will use the terminologies of Schmidt (1985) and Driscoll et al. (1989).

1. If assignments such as "$1 := 2$" *were* allowed, they would presumably have a deleterious effect on arithmetic.

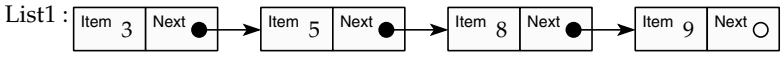
```
procedure insert(item, headptr) =
        var itemptr : list-pointer;
        while not( isnull(headptrˆ.next) or headptrˆ.nextˆ.item > item) do
                headptr := headptrˆ.next;
        allocate(itemptr);
        itemptrˆ.item := item;
        itemptrˆ.next := headptrˆ.next;
        headptrˆ.next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

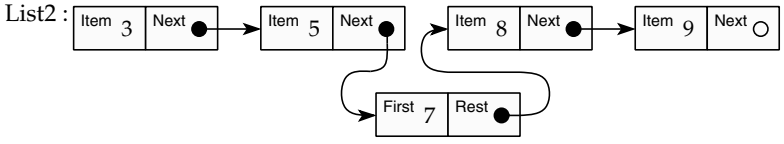
(b) A functional implementation of insert.

Figure 1.1: Comparing functional and imperative code.



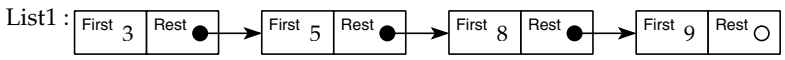
(a) The list is made up of storage locations whose values can be destructively modified.

List1 : *?????*

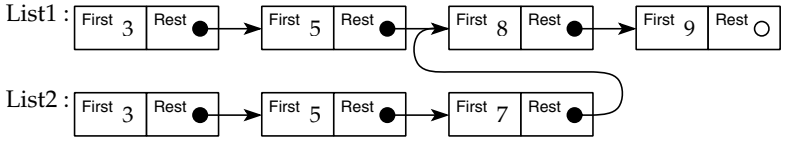


(b) Typically, insertions into a list will be done destructively. Having performed the update, we can no longer reference the old version of the list. Because updates destroy prior versions of the list, we call this kind of data structure *ephemeral*.

Figure 1.2: A linked list in an imperative language.



(a) 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; this property leads us to call the data structure *persistent*.

Figure 1.3: A linked list in a functional language.

In an imperative language, we typically perform destructive updates on data structures, modifying them as necessary. Figure 1.2 shows the implications of modifying data structures destructively: updates to the data structure destroy the old version of the data structure in creating the new. Driscoll et al. describe such mutable data structures as *ephemeral*, expressing the transient nature of their contents. In contrast, data in functional languages can only be revised by creating new values, not by destructively modifying old ones. Figure 1.3 shows how a functional linked list uses *path copying* (Sarnak & Tarjan, 1986) to create a new version of the linked list. Driscoll et al. describe such immutable data structures as *persistent*.[2]

Schmidt terms algorithms that only refer to the most recent version of a data structure as *single-threaded*; those that do not are *non-single-threaded*. From these definitions, it follows that single-threaded algorithms only require data structures that are ephemeral, and that ephemeral data structures can only support single-threaded algorithms. Functional programs can be non-single-threaded, but can also (and often do) access their data single-threadedly.

Given that functional and imperative programming styles and data structures are different, the obvious next question is: How well can we adopt one style while working

2. Driscoll also introduces the concept of *partial persistence*, in which the original version remains readable after an update but cannot itself be updated. We will return to this concept in Section 2.1.

in the other? A typical algorithms textbook covers a plethora of imperative algorithms; if these algorithms can easily be ported to a functional setting—and run efficiently!—functional programmers may benefit from this arsenal of techniques.

The linked-list example above shows us that for some data structures, even though the functional version may do more copying than its imperative counterpart, the asymptotic complexity can remain the same; additional costs can be small enough to be outweighed by the elegance, safety, and potential for non-single-threaded use present in the functional solution. Most straightforward linked data structures, such as trees, heaps, and so forth, are analogous to linked lists in this respect. However, not all imperative data structures have obvious and pleasing functional analogues (e.g., arrays, double-linked lists, and file-systems), which leads us to consider how to handle these more awkward cases.

1.2 Integrating Imperative Structures into Functional Languages

There are two ways to support imperative structures in a functional setting. The first is to find a way to allow imperative features into the language, ideally without disrupting the overall functional feel. The second is to find a way to turn the ephemeral structures of imperative languages into persistent data structures suitable for use in functional languages. These two methods are not necessarily mutually exclusive: A functional language with imperative extensions can be very useful in implementing functional versions of imperative data structures.

1.2.1 Providing Access to Imperative Features

One approach for integrating imperative features—such as destructive update—into functional languages is to provide mechanisms that allow imperative constructs to be used without violating the functional properties of the language. The three approaches we will consider are monads, linear types, and direct access to imperative features.

A criticism that applies to all three of these approaches is that support for imperative programming techniques dilutes or eliminates the very properties that differentiate functional programming from imperative programming. For example, functional-programming style makes certain coding errors (such as the accidental overwriting of data) much less likely; allowing imperative data structures or algorithms to be incorporated into functional code negates this advantage. If a language attempts to partition imperative code from functional code, in an attempt to preserve the formal properties of the functional component of the language, it is likely to create two universes for programmers to work in—a functional universe and an imperative universe—with poor integration between the two. For example, if a data structure is only available ephemerally, it cannot be incorporated into ordinary functional data structures (because they are persistent), and therefore cannot be used in non-single-threaded algorithms.

Monads

Monads (Wadler, 1990a, 1992) are a very general encapsulation mechanism, with strong mathematical foundations in category theory (Lambek & Scott, 1986; MacLane, 1971).[3] From a software engineering perspective, monads are simply an abstract data type with an interface that satisfies a particular set of algebraic laws. For our discussion, we are mostly concerned with monads that "hide" state by cloaking it with a monadic veil that completely controls all access to that state. A monad can ensure single-threaded access to ephemeral state by making it impossible to specify any other access pattern. (Monads are not the only means to accomplish this end—Paul Hudak (1992a; 1992b) has suggested continuation-based mutable abstract datatypes, which have a similar flavour to the monadic datatypes we are discussing here.)

Haskell (Hudak et al., 1992; Peyton Jones et al., 1999), for example, uses monads extensively to model the file-system. Extensions to Haskell have been proposed to support *state threads* (Launchbury & Peyton Jones, 1995), which provide single-threaded access to mutable storage locations and mutable arrays.

3. In fact, the generality of monads may also be a weakness. Although monads are expected to satisfy certain laws as to their behaviour, that behaviour can vary significantly from monad to monad, meaning that programmers' intuitions about how monads behave gleaned from working with one kind of monad can mislead them when they deal with another kind.

Monads do allow pure functional languages to provide features that are usually found only in "impure" languages, but they are not without their problems. Creating a composite monadic operation to perform a particular algorithm can result in complicated-looking and unintuitive expressions. Also, using more than one monad at a time can be difficult (it remains an open research area). If separate monads are used to support distinct language features (such as arrays, continuations, and IO) we may end up with a situation where programmers cannot use two or more of these features at the same time. This problem can be exacerbated by the fact that programmers can write their own monads, (e.g., for error reporting or program profiling), increasing the likelihood of needing to use multiple monads.

With careful design, many of these problems are not insurmountable. Language syntax can make the construction of monadic expressions more palatable (as exemplified by the do-notation added to Haskell in the 1.3 revision of the language), and if the various kinds of monads available are well designed and properly integrated, we can eliminate some of the occasions when we might have had to ponder how to combine two monads (Launchbury and Peyton Jones's state-threads monad (1995), for example, makes Haskell's IO monad an instantiation of their state monad, thereby removing any difficulties in combining the two).

Linear Types

Linear types (Wadler, 1990b) (and also *unique types* (Achten et al., 1993)) seem to be a very promising solution to the problem of enforcing single-threadedness. If a data item has a linear type, the type system statically ensures that at most one reference to it exists at any time. If such a data item is passed into a function, the function can safely use destructive update on that data because the function can be certain that no other references to that data exist. Issues such as garbage collection also become greatly simplified, due to the impossibility of there being multiple references to objects with linear type.

By using the type system to enforce the single-use property, this approach allows single-threaded algorithms that use ephemeral data to look very much like ordinary functional code, and represents a viable alternative to monads for allowing imperative structures into a language safely. *Concurrent Clean* (Huitema & Plasmeijer, 1992) uses

this approach to provide access to the file system and mutable arrays.

However, linear types are not a panacea. The single-use restriction can be quite limiting when it comes to expressing algorithms, making porting imperative algorithms (which inherently have no single-use restrictions) more complex than it might first appear.[4]

Giving In

A final possibility is for the language to abandon the requirement that it be purely functional, allowing imperative storage locations and abandoning referential transparency (Quine, 1960; Søndergaard & Sestoft, 1990). Such a language would usually retain a purely functional sublanguage, in which programmers could choose to remain. *Standard ML* (Milner et al., 1990) takes this approach.

This approach can be criticized on the grounds that imperative code can violate referential transparency. Without the assurance of referential transparency, programs become much harder to reason about (both formally and informally) and lose some opportunities for aggressive optimization. This argument may lead us to prefer the strong separation of functional and imperative data that occurs in both monads and linear types.

But barriers between functional and imperative code can be a problem, preventing functional programmers from using some valuable data structures. For example, externally functional self-adjusting data structures (e.g., a data structure that rearranges itself on each read access to improve the performance of successive reads) cannot be written using monads or linear types. Recognizing the value of carefully managed ventures into the imperative world, compiler authors for purely functional languages sometimes spurn their language's definition to provide unsafe imperative features when absolutely necessary (e.g., both *GHC* and *Hugs* provide `unsafePerformIO` as a means to escape the usual constraints of Haskell's IO monad (Peyton Jones et al., 1993)). Programmers are responsible for ensuring that their code cannot lead to visible violations of referential transparency if they choose to use these language extensions.

4. In general, we might have to resort to passing an additional argument representing the memory of the imperative program into and out of every function.

Moreover, allowing easy access to imperative features allows experienced programmers to write code as they see fit, without being cast out of Eden for occasionally nibbling on the imperative apple. Whether this property is really an advantage or not is likely to remain a topic of lively debate.

1.2.2 Wrapping Imperative Data So That It Appears Functional

We have seen that one approach to incorporating imperative data into functional languages is to provide direct or indirect access to imperative features. The alternative is to find a way to make an ephemeral imperative data structure appear to be a persistent functional data structure. Many of the schemes that follow this second strategy permit persistent (non-single-threaded) use of the formerly-ephemeral data structure, but do not make such usage patterns efficient—they expect the data structure to actually be used single-threadedly.

Update Analysis

Update analysis (Bloss, 1989; Hudak & Bloss, 1985)—sometimes called *sharing analysis* (Jones & Le Métayer, 1989)—provides a compiler with knowledge of when it may avoid allocating space for a new value and instead destructively modify an old value that has just become unreferenced; in essence performing garbage collection at compile time (Mohnen, 1995). These techniques use *abstract interpretation* (Cousot & Cousot, 1977, 1979) to work out the point at which a data item cannot possibly be referenced and can safely be overwritten with new data. When update analysis discovers a single-threaded access sequence, data structures can be updated in place, rather than needing to be copied.

A problem with this approach is that, for an arbitrary functional program, the task of statically determining whether a data structure is used single-threadedly is undecidable. So the determination of single-threadedness is conservative, erring on the side of caution and flagging some programs that will actually execute single-threadedly as non-single-threaded. Unfortunately, it may not always be obvious to the programmer whether or

not update analysis will flag their code as suitable for destructive update or not. A small local change can have far-reaching and unexpected consequences on performance if it upsets the analysis.

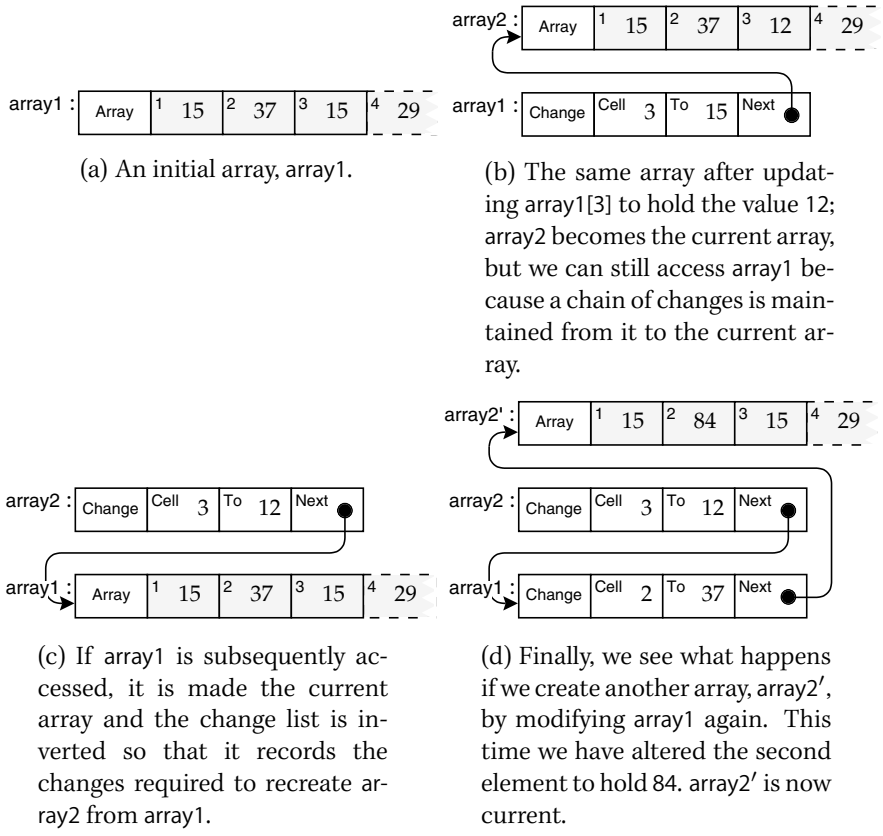
There are also some difficulties with abstract interpretation in general. For example, abstract interpretation may not operate well under separate compilation (it may not be clear, for instance, whether an external function will keep a reference to one of its arguments).

Some of the variability of update analysis can be eliminated by merging its static checks with the runtime checks of *single-bit reference counting* (Wise & Friedman, 1977; Stoye et al., 1984). This technique can catch a few more occasions when destructive update can used safely, but adding this technique may not catch all opportunities for destructive update.

Although update analysis can and should be used to improve the performance of functional languages generally—allowing destructive update where it can be used invisibly—it does not address the issue of what to do if an ephemeral data structure is used persistently. The implicit answer is to make a copy, but this approach far from ideal, because copying the whole data structure could be extraordinarily expensive. Thus, update analysis is best used in the background, supporting other techniques, rather than as the sole solution to the problem.

Change Histories

An oft re-invented technique to make an ephemeral data structure persistent is to use *change histories* (Overmars, 1983; Baker Jr., 1978). The idea is simple: Change the imperative data structure, keeping a log of the information necessary to undo that change. We thereby have one "in use" copy of the ephemeral data structure, and "rollback logs" that can be used to transform it into any previous version. In this way, we use destructive update on the most recent version of the data structure, but if the program references an older version, we undo the changes (while creating a new log that can be used to redo those changes), consider the rolled-back data structure to be the "in use" version, and then access it.



(a) An initial array, array1.

(b) The same array after updating array1[3] to hold the value 12; array2 becomes the current array, but we can still access array1 because 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 it records the changes required to recreate array2 from array1.

(d) Finally, we see what happens if we create another array, array2′, by modifying array1 again. This time we have altered the second element to hold 84. array2′ is now current.

Figure 1.4: Understanding trailers.

This technique has frequently been applied to arrays (Aasa et al., 1988; Baker Jr., 1991), where it has been called *trailers* or *version arrays*, sometimes with some slight variations (Chuang, 1992, 1994). We will discuss the application of change histories to arrays in Section 1.3.4.

Figure 1.4 shows the change-histories technique applied to arrays. Multiple functional-array versions link to a single ephemeral array. The "current" array version links directly to the ephemeral array; other versions are linked to the ephemeral array through difference lists that indicate how those versions differ from the current array version. Whenever a particular array version is accessed, the usual first step is to rearrange the data structure so that the desired version becomes the one that uses the internal array,

adjusting and creating trailers for other versions as necessary.[5] Figure 1.4(c) shows how a subsequent read access to array1 makes it the current array.

The advantage of this approach is that it is simple and can be very widely applied, suiting any data structure that can allow changes to be undone. Change histories only add modest overhead (constant time and space) to imperative structures they encapsulate, provided that the "functional" data structure they provide is used single-threadedly. Non-single-threaded access can be costly, however, because a long change list may have to be traversed to construct the desired version of the data structure. In fact, for all techniques based on this method that are discussed in this dissertation (i.e., all the techniques I am aware of), there are pathological non-single-threaded access sequences that exact huge time or space penalties (see Section 1.3.4 for some examples).

Linked Data Structures

Driscoll et al. (1989) describe general techniques that can transform any ephemeral linked data structure into a persistent one. Unfortunately, their most impressive results were reserved for linked data structures of bounded in-degree, and the in-degree of a data structure cannot, in general, be guaranteed to be bounded in a functional context. Their techniques and terminology set the stage for research into persistence, however, and form the inspiration for some of my work.

The *fat-node* approach suggested by Driscoll et al. avoids storing changes for the data structure as a whole, and instead stores changes at the level of the data structure's link nodes. A fat node stores different values for different versions of the data structure.

Getting Specific

It is no surprise that there is no general technique that can simply and elegantly turn an arbitrary ephemeral structure into an efficient persistent one. As a result, researchers and practitioners have attempted to create efficient functional equivalents of particular imperative data structures.

5. Not all implementations of trailers always rearrange the data structure so that 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 array's future access patterns, information that is rarely available or determinable.

Examples of this approach can be seen in functional implementations of *queues* (Gries, 1981; Hood & Melville, 1981; Burton, 1982), *priority queues* (Brodal & Okasaki, 1996), *double-ended queues* (Hood, 1982; Hoogerwoord, 1992b; Chuang & Goldberg, 1993; Okasaki, 1995, 1997), *binomial queues* (King, 1994), *sorted lists* (Kaplan & Tarjan, 1996), and so on. Excellent reviews of progress in this area can be found in Graeme E. Moss's PhD qualifying dissertation (1996) and Chris Okasaki's PhD dissertation (1996). The latter contributes many new functional data structures, as well as some important techniques for analysing and implementing such data structures.

In this dissertation, we are particularly interested in how functional arrays may be implemented. Existing approaches to this problem are the focus of the next section.

1.3 Specific Techniques for Functional Arrays

Arrays are the quintessential imperative data structure. An imperative 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.

Arrays provide a very low-level abstraction that corresponds very closely to the behaviour of a random-access machine—in fact, because of an array's ability to simulate computer memory, virtually any data structure may be represented inside an array. This property makes arrays possibly the most difficult imperative data structure to re-engineer to be functional.

A functional array must be persistent: The update of a single element must generate a new array, and leave the old version accessible, holding the same values it held before the update. A naïve implementation of these semantics would make a copy of the whole array at every update. Copying an entire array of arbitrary size at every update seems extravagant and inefficient.

There are two techniques for providing efficient update for functional arrays that can be used in combination. The first is to restrict access so that array operations are done en masse—if we have enough updates to do at once (i.e., $\Theta(n)$, where n is the size of the

array), the cost of copying the array can be amortized over the update sequence. The second is to have a data structure that provides the same operations as arrays but is not a conventional imperative array internally, instead having some more complex data structure that can avoid performing an expensive copy operation for each array update.

1.3.1 Monolithic Approaches

The monolithic approach to array operations does not try to find ways of performing small operations on arrays efficiently; instead it provides operations that act at the level of of the array as a whole. A very simple monolithic array operation would be

$$\mathsf{arraymap} \ : \ (\alpha \to \beta, \mathit{array}(\alpha)) \to \mathit{array}(\beta).$$

This operation takes a transformation function and an array as its arguments and returns a new array created by applying that function to every element of the array. The arraymap function may freely copy the array because the $\Theta(n)$ cost of copying the array is matched by the $\Omega(n)$ cost of applying the transformation function to every element. Usually, we want to perform more complex operations on the entire array, and thus most monolithic approaches use a special syntax to denote monolithic array operations.

Having a separate syntax obviously adds complexity, not only for programmers, but also for the language, which may have to perform some checks on the correctness of the operation specified. Monolithic specification does have some useful properties, however, being better suited to supporting parallelism in array access.

Haskell (Hudak et al., 1992) does not provide a special syntax for monolithic array access, but its array-access functions are designed to process groups of updates at once. In current implementations these mass-update operations have $O(n+u)$ cost, where n is the size of the array and u is the number of updates performed together. Although some algorithms can use this technique efficiently—by collecting groups of $O(n)$ updates and performing them en masse—others cannot.

A general problem with the monolithic approach is that unless *update analysis* (Bloss, 1989) is employed by the compiler, the technique cannot even offer good performance

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

(a) The functions above provide monadic operations for arrays. An *array-op* (α, β) 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.

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

(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.

Figure 1.5: An example monadic interface for arrays.

for common single-threaded algorithms from the imperative world, because not all algorithms can be cast into a monolithic-access framework.

1.3.2 Monadic Arrays

In the case of arrays, a monad can be used to bring traditional imperative arrays into a functional language, allowing us to enforce single-threaded access to arrays and thereby not violate the properties of functional programming languages. A monad can refer to an imperative array implicitly (i.e., the array itself cannot be captured as a value), and can restrict operations that can be performed on the array and the ways in which those operations can be combined. An array monad is an abstract data type that encapsulates an operation on an array, not the array itself. An *array operation*[6] has a parametric type;

6. The term *array operation* is used here as a synonym for *array monad*. I believe the term "array operation" makes the text accessible to a wider audience. To the uninitiated, monads may seem to be an esoteric concept.

the type parameters are the type of the array elements and the type of the result from the operation (e.g., the type of size is *array-op* (α, int) because it operates on arrays of arbitrary type, α, and the result of the operation—the size of the array—is an integer).

Figure 1.5 defines a possible interface for a monadic implementation of an array. Figure 1.5(a) shows the core operations for arrays. These operations are of no use by themselves; they only become useful when combined and executed with the functions in Figure 1.5(b). The compose function allows array algorithms to be constructed (and embodied in the form of a compound array operation) from our primitive array operations. The execute function executes an array operation (its other arguments are the size of the array to create and a function to provide initial values for the array elements). The execute function does not take an array as an argument, but instead 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 and returned by the operation being executed because none of the core array operations provide such a feature.

The compose function is essential for specifying array algorithms as array operations. The compose function takes two arguments: an array operation, o, and a function f that returns an array operation. compose returns a new array operation c that, when executed, will perform the array operation o, producing a result, r, and then execute the operation returned by $f(r)$. The resulting array operation c, can then be executed or passed back into compose to create a more complex composite array operation. This mechanism allows us to pass information along a chain of operations and thus allows us to specify useful algorithms. Figure 1.6 compares a simple monad-based algorithm implementation with a more conventional implementation of the same algorithm.[7]

Algorithms that involve more than one array (e.g., an algorithm that merges 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, it may not be embedded within any other data structures; this limitation also prevents

7. In recent versions of Haskell, the ugliness of specifying monadic array algorithms has been significantly mitigated by the introduction of do-notation. However, in our discussion we are concerned with understanding how monadic arrays actually operate, and thus this syntactic sugar would only add obfuscation.

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

(a) A monad-based implementation of max.

```
max array = loop last (read array last)
            where
               last                = (size array) - 1
               loop pos best       = best,                                  if pos = 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.

Figure 1.6: Two implementations of max, written in a Haskell-like language. The first implementation uses monadic array operations, the latter uses a more traditional array interface. Both implementations follow the same algorithm: first finding the size of the array, then working from back to front, keeping track of the largest value found so far. The implementations are a little more wordy than necessary to facilitate the comparison of the two approaches.

multidimensional arrays, since arrays of arrays are not possible unless they are explicitly supported in the monad.

1.3.3 Trees and Tries

One of the most commonly used techniques to provide functional arrays is to use a *balanced binary tree* with integer keys (Myers, 1984), or a tree where the bit pattern of the array index is used to determine the position of the element in the tree (Hoogerwoord, 1992a). Trees are conceptually simple, and can be implemented in almost any functional language, making them a popular choice when programmers need arrays and have to implement them themselves. In languages such as Miranda (Turner, 1986), which lack both arrays and the facility to extend the runtime environment to provide new functions that use imperative features internally, tree-based arrays become the only practical choice.[8]

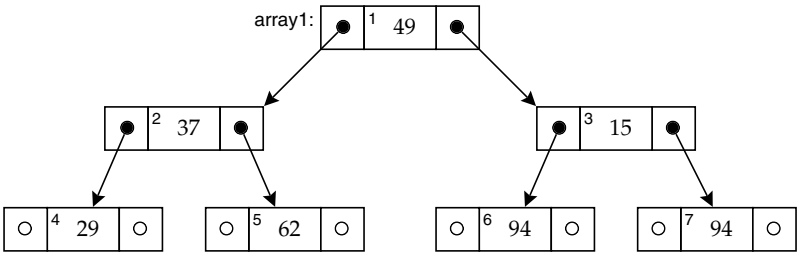
Figure 1.7(a) shows an example of Hoogerwoord's approach; Figure 1.7(b) shows how the data structure is affected by an array-update operation. In general, it is possible to use k-ary trees instead of binary trees; for a k-ary tree representing an array of size n, the tree will have branch nodes of size k and a height of $\log_k n$. As k increases, reads become cheaper (because the height of the tree diminishes) and updates become more expensive (because $\log_k n$ branch nodes of size k must be replaced for every update operation).[9] In the limit, when $k = n$, we have the naïve array-copying approach.

Okasaki (2000) observes that we can choose k based on n (rather than making it a constant), and advocates a choice of $k = 2^{\sqrt{\log n}}$. As we would expect, this technique offers improved performance for reads, but imposes a greater time and space penalty for writes over traditional binary trees.

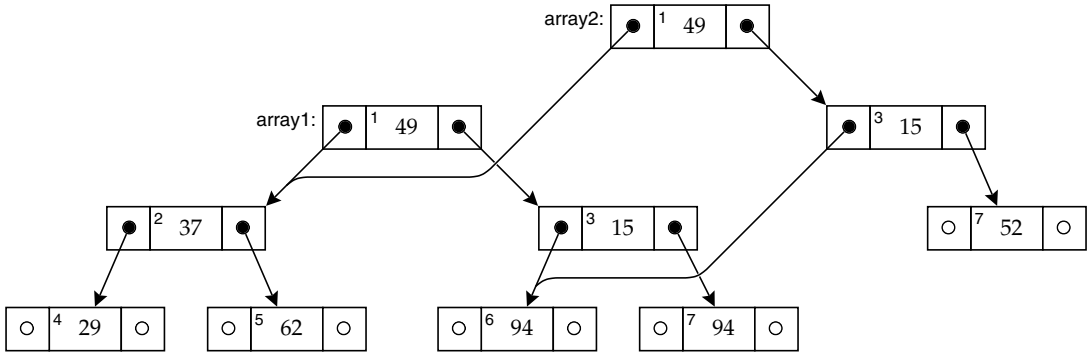
The problem with using trees as an array representation is that read operations require $\mathrm{O}(\log_k n)$ time and update operations require $\mathrm{O}(k \log_k n)$ time and space (where n and k are as defined above). However, if the arrays are small or sparse, the overheads of

8. Another common choice is to use an association list (or just a plain list) to simulate arrays. Representing arrays as lists is computationally inefficient, requiring $\mathrm{O}(n)$ time for element read and $\mathrm{O}(n)$ time and space for element update, where n is the size of the array. The only redeeming feature of list-based arrays is that they can typically be implemented in only a couple of lines of code.

9. For very small k, increasing k may *reduce* the time and space costs of updates. For example, $k = 3$ and $k = 4$ may offer better performance than $k = 2$ (Tarjan, 1983).



(a) An initial array, array1.



(b) The same array after updating array1[7] to hold a value of 52; notice how array2 shares much of the tree structure of array1.

Figure 1.7: Understanding the tree-based array representation.

tree-based arrays make them a practical and simple data structure, especially if they are used single-threadedly.

1.3.4 Trailer Arrays

Sören Holmström (1983), Lars-Henrik Eriksson & Manny Rayner (1984), John Hughes (1985), and Annika Aasa et al. (1988) have proposed a technique called *version-tree arrays* or *trailer arrays*. The technique, which is, in fact, a rebirth of Henry Baker's *shallow-binding* method (1978; 1991), provides excellent performance for single-threaded algorithms. The performance of trailer arrays for non-single-threaded access is less satisfactory, however. When used as a persistent data structure, its performance can be arbitrarily bad, taking time proportional to the number of updates between the old version being accessed and the most recent version.

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

Chuang later developed an alternative method for providing cuts to the version-tree array based on randomization (1994): Instead of making cuts to ensure that reads of any array element take at most $O(n)$ steps, cuts are performed with a probability of $1/n$ (per step) during read operations. This method has an expected–worst-case performance of $O(r + nu)$ for u updates and r reads of an initial array of size n, which we can restate as an expected amortized performance of $O(nu/(r + u))$ per access.[10] Chuang proves that in many cases the expected performance of his technique is within a factor of two from optimal for any strategy involving cuts to a version-tree array. Chuang does not consider the theoretical space overheads of his algorithm, but it appears that the expected amortized space requirements are $O(1)$ space per element read, until the array versions take $O(nu)$ space in total; in other words, the upper bound on space is the same as the upper bound for naïvely copying the array at every update. This result contrasts sharply with other array techniques, which almost universally require no space consumption for element reads.

1.3.5 Fat Elements

Prior to Tyng-Ruey Chuang's work, Paul Dietz (1989) presented, in extended abstract form, a technique that supports fully persistent arrays in $O(\log \log n)$ expected amortized time for read and update, and constant space per update. Dietz's technique is, how-

10. One way to produce this worst-case behaviour is to choose r and u such that $r > u/p$, where p is the probability of array copies (e.g., $p = 1/n$), and then proceed as follows: After each update, read the newly created array $\lceil 1/p \rceil$ times, resulting in an (expected) array copy, requiring $\Theta(n)$ work.

ever, particularly complex, and appears to have never been successfully implemented. It seems unlikely that Dietz's technique could be implemented without a large constant-factor overhead, making it more interesting in theory than useful in practice. Dietz's work in this area seems to have been largely overlooked by those working in this field (Baker Jr., 1991; Chuang, 1992), perhaps because the method was never published as a complete paper. Dietz's technique is interesting, however, because in its early stages it is broadly similar to my own prior and current work on functional arrays (O'Neill, 1994)—both my work and his are inspired by the work of James Driscoll and colleagues on persistent data structures (1989). However, beyond the common inspiration of Driscoll's work, my work and Dietz's differ significantly.

Even more overlooked than Dietz's work is that of Shimon Cohen (1984), which was ignored in all papers I have discussed except for Okasaki (Okasaki & Kc, 2000). Cohen's method is similar to both my fat-elements method and Dietz's method, being based on the idea of storing changes at the element level rather than the array level, and using a version counter. Cohen's method is less sophisticated than either Dietz's or my own, however, and claims[11] a worst-case performance for reads of $\mathrm{O}(u_e)$, where u_e is the number of updates performed on element e (although the method does achieve updates in $\mathrm{O}(1)$ time).

Functional arrays using the fat-elements method were also the subject of my master's thesis (O'Neill, 1994). Like the techniques of Dietz and Cohen, my earlier technique required only constant space per element update, and supported single-threaded array access in constant amortized time. Non-single-threaded reads and updates required $\mathrm{O}(\log u_e)$ amortized time, where u_e is the number of updates performed on element e. Chapters 2 and 3 of this dissertation, show how enhancements to my earlier algorithm can improve this bound.

11. From my understanding, of Cohen's work his result only holds for partially persistent updates—a pathological fully persistent update sequence can actually cause accesses to take $\mathrm{O}(u)$ time, where u is the total number of updates made to the array.

Chapter 2

The Fat-Elements Method for Functional Arrays

In the previous chapter, we saw that prior attempts at functional arrays have had problems with poor worst-case performance, poor space-per-update, or poor performance for the most common uses of arrays. In this chapter, we will discuss the *fat-elements method*, a new method for functional arrays that avoids these weaknesses.

I first discussed the concept of arrays based on a fat-elements technique in my master's thesis, so the ideas presented in the first part of this chapter are not new. In my master's thesis, I developed array and heap data structures and explored different options for version stamping. Sections 2.1 and 2.2 restate the relevant portions of that earlier work.[1] The new material in this chapter begins at Section 2.3, introducing array splitting and its consequences.

The fat-elements method has broad similarities to the *fat-node method* for persistent linked data structures of bounded in-degree, developed by Driscoll et al. (1989). Both methods use a system of *version stamps* and record how data changes over time in a localized way (in *fat nodes* for linked data structures and in *fat elements* for arrays). However, the fat-elements method is more than a naïve application of Driscoll's techniques to the functional-array problem. It uses innovative techniques to ensure good performance for common algorithms and provide good worst-case behaviour.

1. Although the first two sections mostly describe my prior research, they were written as part of this dissertation and allude to the work in later sections in order to provide a cohesive chapter.

In the fat-elements method, each array version receives a unique version stamp, which is used, along with fat elements, to store multiple array versions in a single master array. A fat element maps version stamps to values, allowing it to return that element's value for any version of the array. The master array is simply an array of fat elements. To ensure good performance, the master array never holds more than $\Theta(n)$ versions, where n is the size of the array; the master array is broken into two independent master arrays as necessary to preserve this condition.

2.1 Version Stamps

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

If the most recent version of an array was the only version that could be updated (a restriction known as *partial persistence*), issuing version stamps would be simple: a simple integer counter would suffice. Although partial persistence would be sufficient for some algorithms, it would not properly satisfy the needs of functional programmers, because the array interface would not be referentially transparent and programmers would need to ensure that their programs always "followed the rules", only updating the most recent version of the array.

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

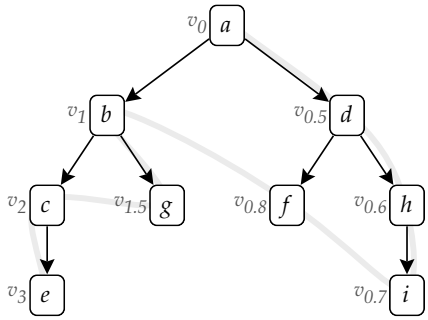
A partially ordered version-stamping scheme would be problematic, however, because it would preclude the use of efficient data structures for storing data keyed by version stamp.

Imposing additional structure on version stamps allows them to follow a total order rather than a partial order. The total ordering for version stamps is defined as follows: If $x_v < y_v$ and $x_v < z_v$, and y_v was created before z_v, then $y_v > z_v$. More formally, we can state the rule as follows: Let V be the set of all currently existing version stamps and x be an array version with version stamp x_v. If we update x to form a new version, y, y will be given a version stamp y_v such that $(x_v < y_v) \ \wedge \ \forall v \in V : (x_v < v) \Rightarrow (y_v < v)$. In other words, y_v is the least version greater than x_v.

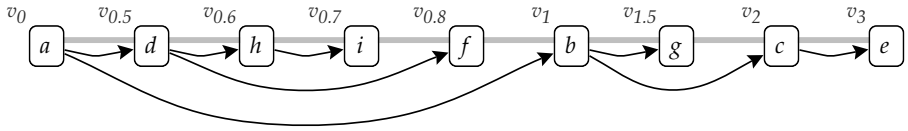
This version scheme corresponds to an existing and well-studied problem: the *list-order problem*. The technique described above is equivalent to inserting an item into an ordered list: y_v is inserted into the ordered list of versions immediately after x_v. A naïve solution to the list-order problem is to maintain a linked list of ordered-list items, where each item is tagged with a real number indicating its position in the list. In this scheme, order comparisons between list items are "efficiently" achieved by comparing the tag values; insertions are performed by finding a (possibly fractional) value for the tag of the newly inserted list item that satisfies the ordering rule (a version inserted between two other versions is given a tag that lies between the tags of its neighbours). A problem with this simple ordered-list algorithm is that it requires arbitrary-precision real arithmetic, which cannot be done in constant time. Practical solutions that take constant time for insertion, deletion, successor and predecessor queries, and comparisons do exist, however (Dietz & Sleator, 1987; Tsakalidis, 1984). A description of Dietz and Sleator's constant–amortized-time ordered-list algorithm is included in Appendix D.

Although the naïve ordered-list scheme has efficiency problems that make it unsuitable for a fast and robust implementation of version stamps, it does have some value in its simplicity: It is useful for describing the concepts of the fat-elements method without being burdened with the details of particular efficient ordered-list algorithms. In the discussion that follows, I will write specific version stamps as v_{tag}, where *tag* is a real number following the naïve scheme. Figure 2.1 provides an example of version stamping using the naïve ordered-list scheme, showing how our version-stamping rules coerce the natural partial order of array versions into a total order.

Figure 2.1(b) also illustrates the concept of *adjacent versions*, to which we will refer later. Put simply, two versions are adjacent if there is no other version between them. It should be clear from the discussion above that in the case of partially persistent (or



(a) A version tree and the totally ordered version stamps that are applied to it. In the diagram, arrows represent updates, and, further, the array versions were created in alphabetical order. Thus, array version c was created before version d was, and so forth.



(b) The same version tree flattened to better show the imposed total ordering.

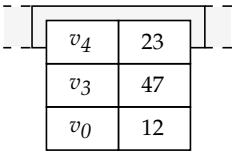
Figure 2.1: Providing a total order for a version tree.

single-threaded) updates, versions that are adjacent at one time will always remain adjacent, because any new versions will be added after all previous versions. With fully persistent updates, versions that are adjacent at one time need not remain adjacent because new versions may be inserted between them.

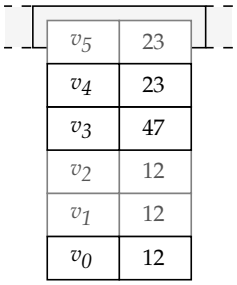
2.2 The Fat-Element Data Structure

As outlined earlier, a fat element is a data structure that provides a mapping from version stamps to values. For our discussion, we will consider this mapping as a set of version-stamp/value pairs. As we will see shortly, the fat-elements method actually uses a tree to represent this mapping efficiently.

Array elements often hold the same value for several array versions, a property that the fat-elements method uses to its advantage. The nature of array updates is that each



(a) The value of this fat element changes in versions v_0, v_3, and v_4. Entries for versions v_1, v_2, and v_5 are omitted because their values can be inferred from the versions preceding them.



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

Figure 2.2: Fat elements need not contain entries for every version stamp.

update operates on a single element of the array—all other elements remain unchanged.[2] Fat elements exploit this property by storing only the entries for versions in which the element's value changes.

The values associated with versions not explicitly represented in a fat element is inferred according to a simple rule: The value corresponding to some desired version stamp, v_d, is determined by finding the closest version stamp, v_c, in the fat element F; where

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

and retrieving the value corresponding to that version stamp. In other words, v_c is the greatest version stamp less than or equal to v_d. This technique is shown graphically in Figure 2.2, which introduces the abstract diagrammatic notation for fat elements.

Since version stamps have a total order, we can use a relatively efficient data structure to provide the necessary insert and lookup operations, such as a height-balanced mutable tree. Further, by using a *splay tree* (Sleator & Tarjan, 1985) instead of an ordinary

2. Although ordinary array update operations change only a single element, the fat-elements method allows for greater difference between array versions—any of the array elements may be changed between one version and the next (although if two versions have little in common, there is little reason to keep them together, an issue we will address in the next section). Thus, the fat elements method is compatible with batched updates that modify several elements of the array at once and with garbage collection of unused array versions; we will discuss these topics in Chapters 5 and 4 respectively.

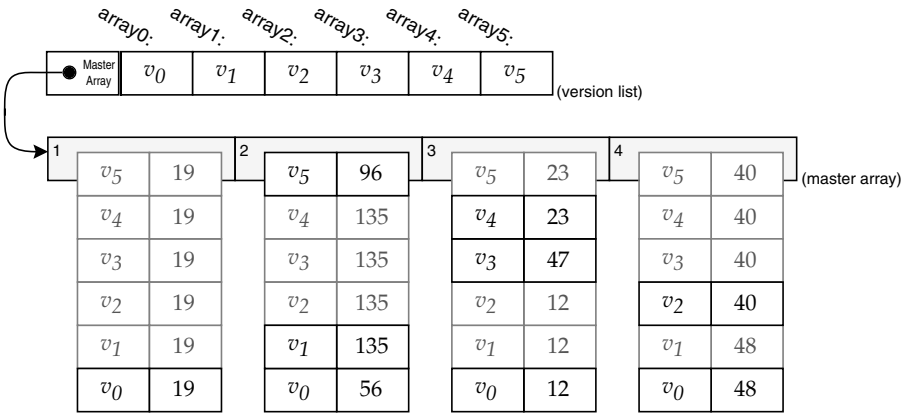


Figure 2.3: Putting the pieces together.

balanced binary tree, we can gain the property that the most recently read or updated version in a fat element is at the top of the tree, and can be accessed in $O(1)$ time. This property guarantees that accesses performed by single-threaded array algorithms execute in $O(1)$ time (because every read will be of the root of the tree, and every update will make the new fat-element entry the root (and the previous root becomes its left child; thus creating a tree with no right children)). Splay trees can also provide useful locality for some non-single-threaded algorithms, while guaranteeing $O(\log e)$ amortized worst-case performance, where e is the number of entries in the fat element.[3] Later, we will ensure that $e \in O(n)$.

Putting fat elements together with the version-stamping system developed in Section 2.1 gives us the basic data structure used for the fat-elements method. Figure 2.3 shows a fat-element array containing six array versions—the initial array version, array0, was initialized with element values [19, 56, 12, 48], and then additional array versions were created by updating the array as follows:

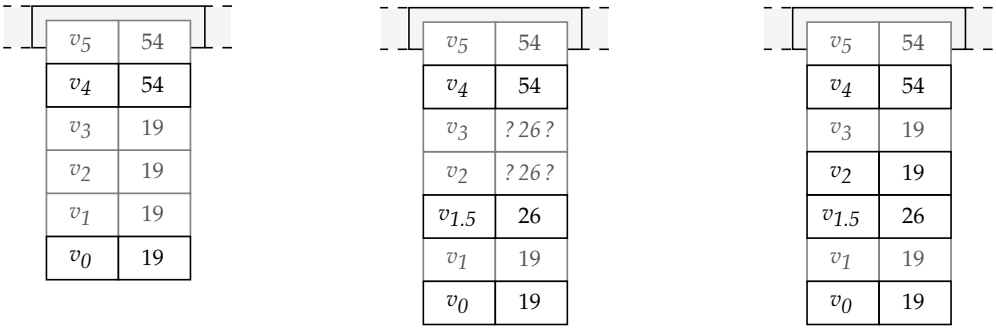
3. Variations of the splay-tree idea that minimize the amount of tree reorganization may also be used to provide single-threaded algorithms with constant-time performance. All that is necessary is to "stack" single-threaded tree updates so that they do not require $O(\log n)$ time to perform, and have the most recent write at the top of the stack. If a non-single-threaded write is performed, the stack is processed and the new additions properly integrated into the tree. Thus, much of the complexity that accompanies splay trees can be eliminated, if desired.

```
array1 = update(array0, 2, 135)
array2 = update(array1, 4, 40)
array3 = update(array2, 3, 27)
array4 = update(array3, 3, 23)
array5 = update(array4, 2, 96)
```

Note that the data structure holding the version list also provides a pointer to the master array such that, given any entry in the version list, the master array can be found in constant time and updated in, at most, $O(v)$ time, where v is the number of versions.

There is, however, one flaw in the method that we must rectify before we move on. It relates to *fully persistent* updates (updates done to array versions that have already been updated at least once) and to an interaction between totally ordered version stamps and the mechanism that fat-elements use to avoid storing redundant information. The problem arises because fat elements need not contain entries for every version stamp—in other words, they may contain "gaps". If an update to an element causes a version-stamp/value pair to be added where there is a gap, the values inferred for versions in that gap may change when they should have remained the same. An example of this problem is shown in the first two parts of Figure 2.4, where we see that adding elements into a gap can change the value of any inferred entries that fall immediately after the added fat-element entry.

Figure 2.4(c) shows how an element value can be inserted into a gap where element values were previously inferred without upsetting any existing inferred element values. In general, if we need to add an entry with version stamp v_x to a fat element, and v_x is not the last version stamp in the version list, we need to check whether the value for v_x's immediate successor in the version list, v_y, is explicitly represented in the fat element, or whether that value is merely inferred. If the value for v_y is inferred, we must convert the inferred entry into an explicit entry before inserting the entry for v_x. Thus, representing a new array version generated by an array update requires constant space, because at most two fat element entries are added for each array update. An array update never requires more than two fat-element entries (one for v_y and one for v_x), and much of the time (when v_x is the last array version, or when v_y has an explicit entry in the fat element) we will only need to add one entry (for v_x).



(a) A fat element with gaps. The entry for v_0 is used to provide values for v_1, v_2, and v_3; similarly, the value for v_5 is inferred from the value for v_4.

(b) Naïvely inserting a value for $v_{1.5}$ inside a gap, upsets the values of v_2 and v_3. Note that the inferred values for v_1, v_4, and v_5 are unaffected.

(c) To ensure correct behaviour, we need to also insert an entry corresponding to v_2 (which was previously just inferred from the v_0 entry), before inserting $v_{1.5}$.

Figure 2.4: Potential difficulties in using linear version stamps.

We now have a working method for functional arrays. The fat-elements method, as presented so far, allows constant time access to the most recently read or updated value of an array element. In general, reading or updating an array element which has been updated u_e times takes $O(\log u_e)$ amortized time.[4] The next section explains how splitting the array can prevent elements from becoming "too fat" and thereby improve performance.

2.3 Breaking Up Is Easy to Do

The goal is now to improve the performance of the fat-elements method, especially its worst-case performance. Section 2.2 described how each fat element is an ordered tree of version-stamp/value pairs. If the size of these trees is unconstrained, performance will degrade as the fat elements increase in size.

My strategy for achieving a better worst-case bound involves splitting the master array into two independent master arrays whenever the fat elements grow too fat. In this

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

refinement, master arrays may hold no more than $O(n)$ versions, where n is the size of the array. More specifically, if c_* is the number of fat-element entries in a master array, every master array must satisfy the constraint $c_* \leq (1 + k_l)n$, where k_l is a positive constant.[5] To preserve this constraint, we split each master array into two independent master arrays when updates cause it to reach its "fatness limit". Since it requires $\Theta(n)$ updates to take a master array from half-full to full, we can amortize the cost of splitting an array of size n over those updates (a full amortized-time analysis is presented in Section 3.1).

In order to know when to split the array, we need to maintain some additional housekeeping information. Specifically, each version stamp now has an associated counter, c_i, holding the number of fat-element entries that are associated with that version. Each master array has a counter, c_*, representing the total number of entries stored in the master array. When it comes time to split the array, we perform the split such that entries corresponding to all versions up to the mth are placed in one master array, and all entries from versions $m + 1$ onwards are put in another. m is an integer such that

$$\sum_{i=1}^{m} c_i \leq \frac{c_* + n}{2} \geq n + \sum_{i=m+2}^{v} c_i$$

holds; where n is the size of the array, v is the number of versions held by the master array, $c_1, \ldots, c_v$ are the counters for each entry in the version list, and c_* is their sum (also note that c_1 will always be equal to n). The left-hand and right-hand sums will be the sizes of the two master arrays after the split—the right-hand sum is $n + \sum_{i=m+2}^{v} c_i$ and not $\sum_{i=m+1}^{v} c_i$ because the first version in the second array will need entries for all n elements, regardless of the number of entries it had before. If there is more than one possible m, we choose the m such that $c_{(m+1)}$ is maximized, thus reducing the number of fat-element entries that have to be added during the split.

5. The exact value of k_l is a matter of choice—my own tests have shown that when reads and writes are equally balanced, a value of approximately five is appropriate, with lower values improving performance when reads outnumber updates.

Another way to state the previous equation is to say that

$$L = \left\{ l \;\middle|\; \sum_{i=1}^{l} c_i \leq \frac{c_* + n}{2} \right\}$$
$$R = \left\{ r \;\middle|\; n + \sum_{i=r+2}^{v} c_i \leq \frac{c_* + n}{2} \right\}$$
$$m \in (L \cap R)$$

Clearly, $1 \in L$ and $v \in R$; L forms a prefix of the sequence $1, \ldots, v$; R forms a suffix of the same sequence. We can show that there is always at least one split point by showing that $L \cap R \neq \varnothing$, which we will do by contradiction. Suppose that $L \cap R = \varnothing$: Because L is a prefix and R is suffix of $1, \ldots, v$, it must be the case that they do not "meet in the middle"; that is,

$$\exists m \in 2, \ldots, v-1 : (m \notin L) \wedge (m \notin R).$$

Thus,

$$\left(\sum_{i=1}^{m} c_i > \frac{c_* + n}{2} \right) \wedge \left(n + \sum_{i=m+2}^{v} c_i > \frac{c_* + n}{2} \right)$$
$$\Rightarrow \sum_{i=1}^{m} c_i + n + \sum_{i=m+2}^{v} c_i > c_* + n$$
$$\Rightarrow c_* - c_{m+1} + n > c_* + n$$

which is a clear contradiction, because c_{m+1} is always positive.

This splitting technique sets an upper bound on the number of fat-element entries in each of the two new master arrays resulting from the split of $(c_* + n)/2$ (where c_* refers to the pre-split master array). An obvious lower bound on the number of entries is n.[6] Since splitting only takes place when $c_* = (1 + \mathrm{k_l})n$, we can see that an upper bound on the number of fat-element entries in each of the new master arrays is $(2 + \mathrm{k_l})n/2$.

6. We could also manipulate the inequality above to yield a tighter lower bound of $\mathrm{k_l}n/2$ when $\mathrm{k_l} \geq 2$. This lower bound does not appear to be useful, however: It is not required to show any of the useful properties of the fat-elements method, and, furthermore, suffers from the drawback that it adds restrictions to $\mathrm{k_l}$.

We will conclude this chapter by showing that the time required for such a splitting operation is $O(n)$.

Lemma 2.1 *Splitting the master array into two independent master arrays takes* $O(n)$ *worst-case time if splitting is done when* $c_* = (1 + k_1)n$.

Proof

The steps involved in splitting the master array are:

1. Allocate space for the second array, which requires $O(n)$ time.

2. Find the split point and break the version list into two separate lists at that point, which, from the description above, takes $O(v)$ time. Since $v < c_*$ and $c_* \in O(n)$, a loose bound on the time taken to find the split point is $O(n)$.

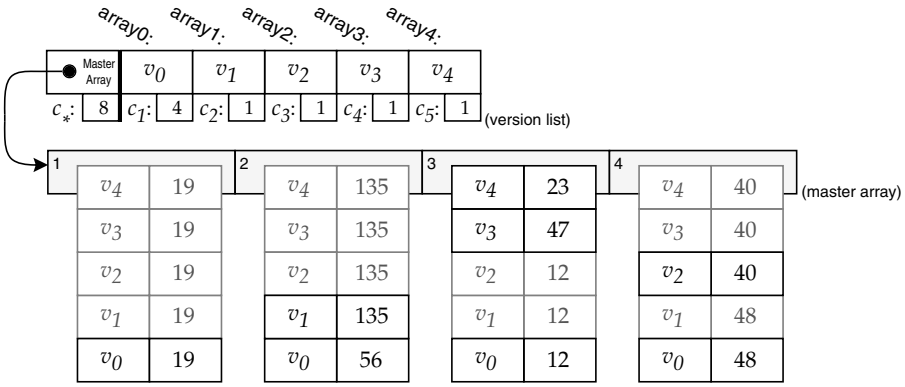
3. Change one of the two version lists so that it points to the new master array, which takes at most $O(v)$ time (assuming that every version stamp must be modified to effect the change), or, more loosely, $O(n)$ time.

4. Split all the fat elements, which takes $O(n)$ time because splitting the splay tree for an element i takes $O(\log e_i)$ amortized time and at most $O(e_i)$ worst-case time;[7] where e_i is the number of entries stored in that fat element. For simplicity, we will use the looser bound of $O(e_i)$ time to split. Thus, the time to split all the elements is $O(\sum_{i=1}^{n} e_i) = O(c_*) = O(n)$. (Note that when we split a fat element we need to ensure that it has an explicit entry for the version that is the split point, requiring us to make an implicit entry into an explicit one if necessary.)

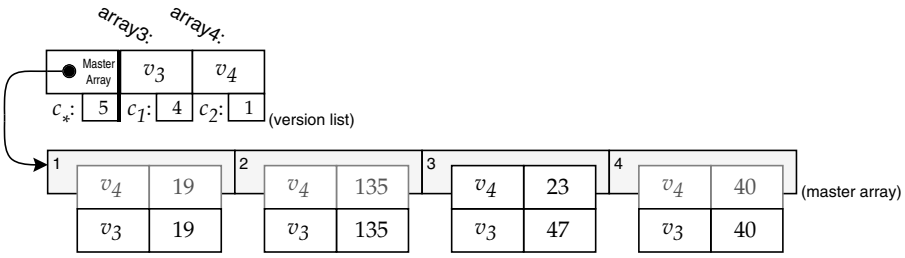
Since each of the steps is bounded by $O(n)$, the whole splitting procedure is bounded by $O(n)$. □

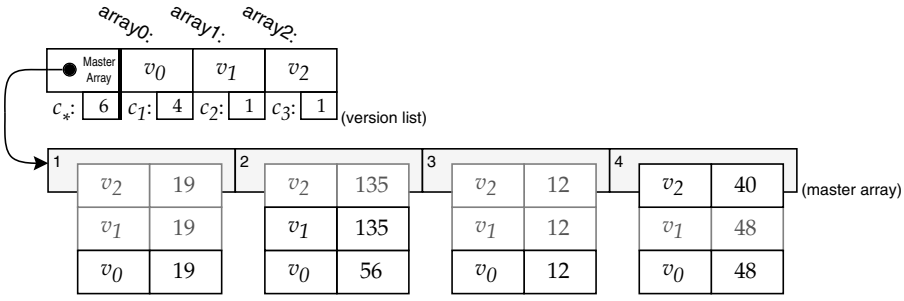
An example of splitting a master array, with $c_* = 2n$, is shown in Figure 2.5.

7. Some search-tree structures might require $O(e_i)$ time if they were used instead of splay trees, although AVL trees (Myers, 1984) can be split in $O(\log e_i)$ worst-case time.



(a) This master array holds a total of eight fat-element entries, the maximum number allowed when $k_l = 1$.





(b) If the master array receives another update, we need to split the array before applying the update so that our invariants are preserved (i.e., $c_* \leq (1 + k_l)n$). The midpoint m is 3; thus the first three versions form the first array, and the remaining two form the second array.

Figure 2.5: Splitting a master array.

Chapter 3

Theoretical and Actual Performance of the Fat-Elements Method

In this chapter, we will discuss the theoretical and actual performance of the fat-elements method. Both topics are of interest because good asymptotic performance is necessary for programs that process large arrays, but good constant-factor overheads are also required for the technique to be generally applicable.

3.1 Amortized Time Analysis

We shall examine the amortized time complexity of the fat-elements method using *potential functions* (Tarjan, 1985). The idea is that each configuration of the data structure is given a real number value called its *potential* (we may thus conceive of a function that takes a data-structure configuration and returns this value). We may think of potential as representing stored energy that can be used to compensate for expensive operations, and thus define amortized work as

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

or, using symbols to represent the words above,

$$T_A \equiv T_R + \Delta_\Phi.$$

We define the potential of a master array in terms of *primitive potential* ϕ defined as

$$\phi \;=\; \mathrm{k}_\Phi \left(c_* - \frac{(\mathrm{k}_\mathrm{l} + 2)n}{2} \right)$$

where n is the size of the array, c_* is the total number of fat-element entries in the master array, k_Φ is a suitably chosen constant (see Lemma 3.2), and k_l is the constant governing the size of the master array (i.e., $c_* \leq (1 + \mathrm{k}_\mathrm{l})n$, as defined in Chapter 2). The potential Φ of a master array is defined as

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

Since we will often be discussing changes in potential, it is worth noting that when Δ_ϕ is positive, $\Delta_\Phi \leq \Delta_\phi$; and that, since n is invariant, $\Delta_\phi = \mathrm{k}_\Phi \Delta_{c_*}$. Both of these relationships are easily derived from the equations above.

Lemma 3.1 *Splitting master array A at or before its maximum fullness (i.e., when $c_* \leq (1 + \mathrm{k}_\mathrm{l})n$), using the algorithm from Section 2.3, produces two independent arrays A' and A'', each with a primitive potential of at most zero, and hence potential of exactly zero.*

Proof

The algorithm from Section 2.3 splits the master array A into two arrays, A' and A''. Without loss of generality, consider the first of these. The splitting algorithm ensures that $c'_* \leq (c_* + n)/2$, and, because $c_* \leq (1 + \mathrm{k}_\mathrm{l})n$, we know that $c'_* \leq (\mathrm{k}_\mathrm{l} + 2)n/2$. Thus $\phi' \leq 0$ and $\Phi' = 0$. □

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

Proof

The actual time T_R taken to perform the split has the bound $T_R \leq \mathrm{k}_\mathrm{s} n$, for some k_s (from Lemma 2.1).

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

Φ'') is 0. Thus the net change in potential Δ_Φ can be defined as

$$\begin{aligned}\Delta_\Phi &= (\Phi' + \Phi'') - \Phi \\ &= 0 - \Phi \\ &= -\left(\frac{\mathrm{k}_\Phi \mathrm{k}_\mathrm{l} n}{2}\right)\end{aligned}$$

since $\Phi = \mathrm{k}_\Phi \mathrm{k}_\mathrm{l} n/2$ when $c_* = (1 + \mathrm{k}_\mathrm{l})n$. Since the amortized time T_A is defined as $T_A = T_R + \Delta_\Phi$, it has the bound

$$T_A \leq \mathrm{k}_\mathrm{s} n - \left(\frac{\mathrm{k}_\Phi \mathrm{k}_\mathrm{l} n}{2}\right)$$

If we define k_Φ such that $\mathrm{k}_\Phi \geq 2\mathrm{k}_\mathrm{s}/\mathrm{k}_\mathrm{l}$, we find that $T_A \leq 0$. □

Lemma 3.3 *Adding an entry to fat element x takes at most $\mathrm{k}_\mathrm{i} \log_+ e_x + \mathrm{k}_\Phi$ amortized time; where e_x is the number of entries stored in fat element x, and k_i is a constant of the tree-insertion algorithm.*

Proof

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

Case 1: The master array is not full.

In this case the actual time taken to insert the entry will be the time taken to perform an insertion in the tree used to represent fat elements. We will assume that the tree insertion takes time bounded by $\mathrm{k}_\mathrm{i} \log_+ e_x$, where k_i is a constant of the tree-insertion algorithm. Both splay trees (Sleator & Tarjan, 1985) and balanced binary trees satisfy this assumption.[1]

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

1. In the case of splay trees, the time bound is an amortized bound, but our amortization analysis of the fat-elements method is independent of the amortization analysis for splay trees. The potential of each splay tree has no effect on the potential of the master array that contains it.

Therefore, the amortized time in this case is $T_A \leq \mathrm{k_i} \log_+ e_x + \mathrm{k_\Phi}$.

Case 2: The master array is full.

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

After the split, we have two arrays, A' and A'', each with at most $(c_* + n)/2$ fat-element entries. We will insert the new fat-element entry in just one of them. Without loss of generality, we will call the array that receives the new fat-element entry A'.

Inserting a new fat-element entry takes at most $\mathrm{k_i} \log_+ e'_x$ time, where e'_x is the size of the fat element in A'. Since $e'_x \leq e_x$, we can also say that the time taken to perform the insertion is bounded by $\mathrm{k_i} \log_+ e_x$, and, as in the previous case, causes a change in potential for that array of at most $\mathrm{k_\Phi}$. The potential of A'' remains the same.

Thus, the amortized time $T_A \leq \mathrm{k_i} \log_+ e_x + \mathrm{k_\Phi}$. □

Lemma 3.4 *Updating element x of an array version, takes at most $2(\mathrm{k_i} \log_+ e_x + \mathrm{k_\Phi}) + \mathrm{k_f}$ amortized time; where e_x is the number of entries stored in fat element e of the array version's master array, $\mathrm{k_i}$ is a constant of the tree-insertion algorithm, and $\mathrm{k_f}$ is the time required to find the particular fat element x the master array.*

Proof

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

Lemma 3.5 *Updating any element of an array version takes $\mathrm{O}(\log n)$ amortized time. (Specifically, the update takes at most $2\left(\mathrm{k_i} \log_+ (\mathrm{k_l} n + 1) + \mathrm{k_f} + \mathrm{k_\Phi}\right)$ amortized time.)*

Proof

The maximum size of fat element e_x in the master array is $e_x \leq c_* - n + 1$ and c_* has the bound $c_* \leq (1 + \mathrm{k_l})n$. Thus, $e_x \leq \mathrm{k_l} n + 1$. Therefore this lemma follows from Lemma 3.4 □

Lemma 3.6 *Reading element i of an array version takes at most $\mathrm{k_r} \log_+ e_i$ amortized time; where e_i is the number of entries stored in fat element i of the array version's master array, $\mathrm{k_r}$ is a constant of the tree-lookup algorithm, and $\mathrm{k_f}$ is the time required to find the particular fat element i the master array.*

Proof

Finding the value for an element in a particular version of the array requires that we find that fat-element entry associated with that version in the corresponding fat element of the master array, which requires $\mathrm{k_f}$ time. The fat element holds e_i entries, and we assume that our tree lookup algorithm takes at most $\mathrm{k_r} \log_+ e_i$ amortized time. This bound is the total bound on the amortized time taken to perform the operation, because reading the array does not change its potential □

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

Proof

Analogous to Lemma 3.5. □

At this point, we have determined the asymptotic performance of the fat-elments method for arbitrary access sequences, but we can show a tighter bound on performance for an important common case. Usually, array algorithms do not single out one element and access that element alone—it is quite common for array algorithms to access $\Theta(n)$ elements of the array, and not access any element significantly more than other elements. In this case—which I call accessing the array *evenly* (defined precisely below)—we can obtain a tighter performance bound over the general case.

Lemma 3.8 *When a sequence of read accesses occurs* evenly *across a master array, those accesses require constant amortized time per access. The evenness condition is defined as follows: If element i of the master array is accessed a_i times, for a total of a_* accesses across the entire array (thus, $a_* = \sum_{i=1}^{n} a_i$), the array accesses occur such that*

$$\forall i \in \{1, \ldots, n\} : \quad a_i \leq \frac{\mathrm{k_a} a_*}{n}$$

where n is the size of the array and $\mathrm{k_a}$ is a constant. (Specifically, the amortized time required per access is at most $\mathrm{k_r} \log_+(\mathrm{k_a}(1+\mathrm{k_l}))$, where $\mathrm{k_r}$ is a constant of the tree-lookup algorithm—see Lemma 3.6.)

Proof

We will consider a master array where each fat element corresponding to array element i contains e_i entries (note that $\sum_{i=1}^{n} e_i = \sum_{i=1}^{v} c_i = c_*$).

For this analysis, we do not need to use potential functions. The amortized time per access can be defined as

$$\begin{aligned} \text{amortized time} &= \frac{\text{time for } a_* \text{ accesses}}{a_*} \\ &\leq \frac{1}{a_*} \sum_{i=1}^{n} \left(a_i \left(\mathrm{k_r} \log_+ e_i + \mathrm{k_f} \right) \right) \\ &= \mathrm{k_r} \sum_{i=1}^{n} \left(\frac{a_i}{a_*} \log_+ e_i \right) + \mathrm{k_f} \end{aligned}$$

Since $\log_+$ is a convex function, we can use Jensen's inequality (described in most texts on convex functions (e.g., Pecaric et al., 1993)):

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

Given the condition that the array is accessed evenly, we can now say

$$\begin{aligned} \text{amortized time} &\leq \mathrm{k_r} \log_+ \left(\sum_{i=1}^{n} \left(\frac{\left(\frac{\mathrm{k_a} a_*}{n} \right)}{a_*} e_i \right) \right) + \mathrm{k_f} \\ &= \mathrm{k_r} \log_+ \left(\frac{\mathrm{k_a}}{n} \sum_{i=1}^{n} e_i \right) + \mathrm{k_f} \\ &= \mathrm{k_r} \log_+ \left(\frac{\mathrm{k_a}}{n} c_* \right) + \mathrm{k_f} \end{aligned}$$

But $c_* \leq (1 + \mathrm{k_l})n$; therefore,

$$\text{amortized time} \quad \leq \quad \mathrm{k_r} \log_+ \left(\mathrm{k_a}(1 + \mathrm{k_l})\right) + \mathrm{k_f} \qquad \square$$

Notice that the conditions of the above lemma do not require all the elements to be accessed; some fraction of the elements may be ignored completely.

Lemma 3.9 *The number of updates u required to take an initial array to the point where it needs to be split is $\Theta(n)$. (Specifically, $(\mathrm{k_l}n + 1)/2 \leq u \leq \mathrm{k_l}n$.)*

Proof

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

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

Proof

An array that has just been split will contain $n \leq c_* \leq (2+\mathrm{k_l})n/2$ entries, but at moment the array is split, $c_* = (1+\mathrm{k_l})n$; thus an array produced by splitting could have as few as n fat-element entries or as many as $(2 + \mathrm{k_l})n/2$ entries (see Section 2.3 for a more detailed discussion of these bounds). We have already considered the first case in Lemma 3.9; therefore, $u \leq \mathrm{k_l}n$. For the second case (the worst case for splitting), the array resulting from the split contains $(2 + \mathrm{k_l})n/2$ fat-element entries and thus only $\mathrm{k_l}n/2$ fat-element entries may be added to such an array before it has $(1 + \mathrm{k_l})n$ fat-element entries and needs to be split again. In the worst case for updates, every update adds two fat-element entries. Thus, $2u \geq \mathrm{k_l}n/2$ or $u \geq \mathrm{k_l}n/4$. $\square$

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

Proof

Trivially from Lemma 3.9 and Lemma 3.10. □

Lemma 3.12 *When a sequence of read accesses occurs* evenly *across m master arrays, those accesses require $\mathrm{O}(\log m)$ amortized time per access. The evenness condition is defined as follows: If element i of master array j is accessed $a_{i,j}$ times, for a total of a_{**} accesses across all m master arrays (thus, $a_{**} = \sum_{j=1}^{m}\sum_{i=1}^{n} a_{i,j}$), the array accesses occur such that*

$$\forall i \in \{1,\ldots,n\}, j \in \{1,\ldots,m\} : \quad a_{i,j} \leq \frac{\mathrm{k_a} a_{**}}{n}$$

where n is the size of the array and $\mathrm{k_a}$ is a constant. (Specifically, the amortized time required per access is bounded by $\mathrm{k_r}\log_+(\mathrm{k_a}(1+\mathrm{k_l})m) + \mathrm{k_f}$, where $\mathrm{k_r}$ is a constant of the tree-lookup algorithm—see Lemma 3.6.)

Proof

This proof is analogous to that of Lemma 3.8, except that there are multiple master arrays. We will consider a collection of master arrays where each fat element corresponding to array element i of master array j contains $e_{i,j}$ entries, with e_{**} defined as $e_{**} = \sum_{j=1}^{m}\sum_{i=1}^{n} e_{i,j}$.

The amortized time per access can be defined as

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

Given the condition that the master arrays are accessed evenly, we can now say

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

But $e_{**} \leq m\big((1 + \mathrm{k_l})n\big)$; therefore,

$$\text{amortized time} \leq \mathrm{k_r} \log_+ \big(\mathrm{k_a}(1 + \mathrm{k_l})m\big) + \mathrm{k_f} \qquad \square$$

Corollary 3.13 *A collection of read accesses across a set of versions created from an initial array version by* $\mathrm{O}(n)$ *updates, in which the set of versions is accessed evenly, requires* $\mathrm{O}(1)$ *amortized time per access.*

Proof

From Lemma 3.11, the $\mathrm{O}(n)$ versions created by the updates will reside in a constant number of master arrays. Hence this corollary follows from Lemma 3.12. $\square$

Corollary 3.14 *A collection of arbitrary read or update accesses, across a set of versions created from an initial array version by* $\mathrm{O}(n)$ *updates, in which the set of versions is accessed evenly, and no more than* $\mathrm{O}(n)$ *updates are made, takes* $\mathrm{O}(1)$ *amortized time per access.*

Proof

This corollary is analogous to Lemma 3.12 and Corollary 3.13, except that splits may occur because we allow updates. The $\mathrm{O}(n)$ updates can only cause a constant number of splits (from Lemma 3.11), meaning that there is a constant upper bound on the number of master arrays. This constant bound on the number of master arrays and, thus, $\mathrm{O}(n)$ bound on the total number of fat-element entries allows us to generalize Corollary 3.13 to cover updates as well as reads in this case. $\square$

Thus, the fat-elements method offers constant amortized-time performance when array versions are accessed evenly or single-threadedly—even when arrays are not used in these patterns, the fat-elements method provides a quite reasonable $O(\log n)$ worst-case amortized performance.

3.2 Real-World Performance

In the previous section we showed that the fat-element method has solid theoretical properties. Whether a data structure is useful, however, depends not only on its theoretical complexity but on the constant-factor overheads involved in using that data structure in the real world and how they compare to those of other data structures that can be used for the same purposes. In this section we will briefly examine how a Standard ML implementation of the fat-elements method fared against ML implementations of competing techniques.

Our goal is not to try to deduce the asymptotic complexity of the fat-elements method from experimental results—we have already found these results in the preceding section. In fact, to try to make such inferences would be difficult, because experimental results often have small anomalies resulting from the complex interactions found on a modern computer system with processor caches, virtual memory, and overheads from running a large language such as ML. These anomalies, however, affect neither the deductions we can make about usability, nor the fundamental properties of the algorithms involved.

The discussion that follows compares the performance of the fat-elements method against that of an implementation of functional arrays using binary trees and an implementation using the trailers technique. The tree-based array implementation was based on the code given in *ML for the Working Programmer* (Paulson, 1991), which was inspired by the work of Hoogerwoord (1992a). The trailer-array implementation was based on that of Annika Aasa, et al. (1988).

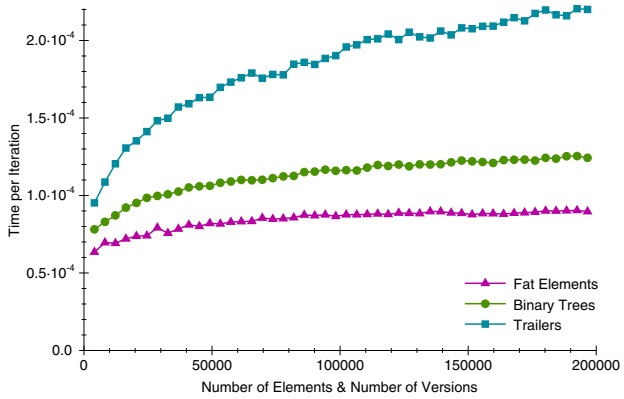
There appear to be no popular benchmarks for functional arrays (unlike imperative arrays, which are the basis of many benchmarks for imperative languages). The benchmarks we will discuss below were originally developed to test the functional-array techniques I presented in my Masters thesis (1994). The benchmark programs are simple, designed to highlight the important properties of functional arrays. The first benchmark

uses an array as a fully persistent structure. The second benchmark uses an array as an ephemeral data structure, accessing data single-threadedly. The third uses an array as a partially persistent structure. (See Appendix H for details on obtaining source code for these benchmarks). The benchmarks were run using *Standard ML of New Jersey* (version 110.0.7) on a Sun Enterprise 3000, with 832 MB of memory running *Solaris 8*. Benchmarks involving the fat-elements method had the array-splitting parameter $k_1 = 5$.

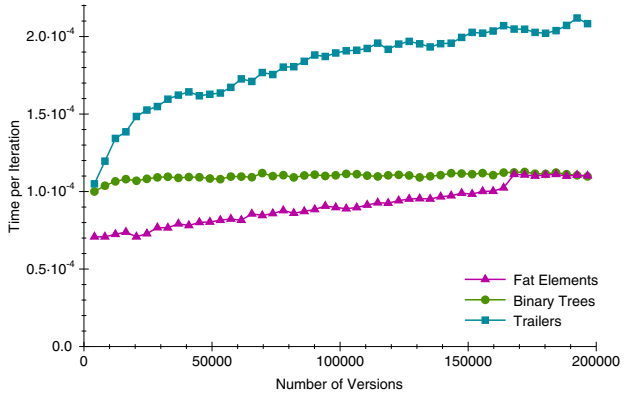
The first benchmark creates v different versions of an array of size n, with each version depending on three randomly selected prior versions of the array. The benchmark repeatedly updates a random element of a randomly selected array version with the sum of random elements of two other randomly selected array versions. Figure 3.1 shows the results of this test.

Figure 3.1(a) shows how the performance of each of the functional-array techniques varies as we increase both the number of different versions and the size of the array (with $n = v$). Notice that although the fat-element method requires $O(\log n)$ amortized time per access in the worst case, the random element accesses of this benchmark can be expected to cover the array evenly, resulting in $O(1)$ amortized time per access. The graph mostly echoes these expectations, but in any case, the benchmark using fat-elements method executes faster than the benchmarks using competing techniques.

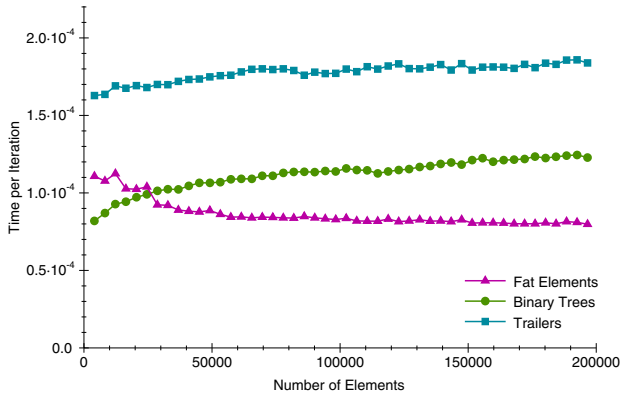
Figure 3.1(b) shows how the performance of each of the techniques varies as we increase the number of array versions while holding the size of the array constant (at 2^{16} elements). From theory, we would expect the performance of binary trees to depend only on the size of the array, and we would expect the time per access to stay constant. The graph mostly fits these expectations, but shows some anomalous behaviour for small arrays. From theory we might also expect the performance of fat elements to take a constant time per iteration. However, although there is a constant upper bound, we see that actual times increase as we increase versions until splitting begins, which is the source of the discontinuity on the graph. After this discontinuity, the difference in speed between the fat-elements method and trees appears to be small, and, for larger arrays, trees will both take more time and require $O(\log n)$ space per update compared to the constant space per update required for fat elements and trailers. These results suggest that binary trees may be the most appropriate technique for the particular case of small arrays with many versions.



(a) Varying n and v.



(b) Varying only v ($n = 2^{16}$).



(c) Varying only n ($v = 2^{16}$).

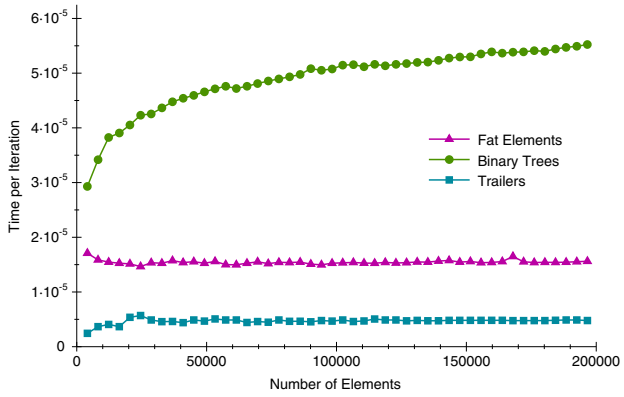
Figure 3.1: Results from the multiversion test.

Figure 3.1(c) shows how the performance of each of the techniques varies as we increase the size of the array while holding the number of versions constant (at 2^{16}). Although theory would lead us to expect the performance of the fat-elements method to be bounded by a constant, the results show varying times, and the overheads of splitting cause it to yield lower performance than binary trees for small n (with seven splits for $n = 4096$, three splits for 8192 and 12,288, and one split for 16,384, 20,480 and 24,576—for large n no splitting is required). As array size increases, the speed of the fat-elements method improves because the average number of fat-element entries per element diminishes as n increases. For $n > 25{,}000$, the fat-elments benchmarks runs faster than the benchmarks for other techniques.

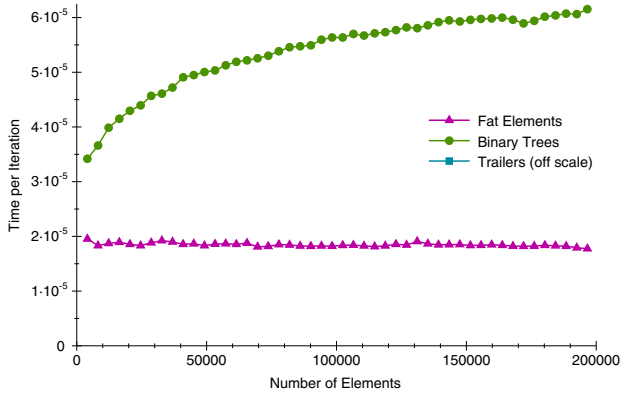
The remaining two benchmarks test the array data structures in simple ephemeral and partially persistent usage. These benchmarks use the simple test of reversing an array. The first benchmark (shown in Figure 3.2(a)) uses the standard imperative array-reversal algorithm, swapping leftmost and rightmost elements and working inwards until the entire array is reversed. The second (shown in Figure 3.2(b)) uses a partially persistent method, performing updates from left to right, always reading from the original array, but updating the most recently changed array.

Figure 3.2(a) shows that trailers significantly outperform both the fat-elements method and binary trees for imperative algorithms. This result is not surprising, because trailers are primarily designed to support imperative algorithms. These results also show that, although the fat-elements method is not the fastest technique for this application, it outperforms binary trees and fulfills our theoretical expectations of $O(1)$ performance.

Figure 3.2(b) reveals the weaknesses of the trailers method. Binary trees and fat elements show virtually identical performance to their results for the imperative algorithm. The results for trailers, on the other hand, do not even make it onto the graph, taking 0.04 seconds per iteration when $n = 4096$. The poor performance of trailers is predicted by theory, which stipulates that a trailer-based implementation of this array reversal algorithm will require $\Theta(n)$ time per access ($\Theta(n^2)$ time overall). Even if we had used the first of Chuang's techniques for speeding up trailers (1992), the results would show the same behaviour. (Chuang's second, probabilistic, method (1994) would have $O(1)$ expected amortized performance for this test, but, as we noted in Section 1.3.4, this method has its own problems when it comes to space usage.)



(a) Ephemeral array reversal.



(b) Partially persistent array reversal.

Figure 3.2: Results from the two tests based on array reversal.

3.3 Conclusion

We have seen that as well as having good asymptotic performance, the fat-elements method is competitive with other techniques for providing functional arrays and offers better real-time performance than those techniques in many cases. In particular, the fat-elements method provides reasonable performance for every case, making it a good general-purpose functional-array implementation.

Chapter 4

Garbage Collection for the Fat-Elements Method

In this chapter, we will examine the issues that arise if we wish to have array versions that are no longer referenced by the program removed from the fat-elements data structure. In the preceding chapters, we have shown how the fat-elements method can efficiently support array operations such as read and update, but like other functional data structures, fat-element arrays provide no delete operation to remove array versions that are no longer required. Functional programming languages typically eschew explicit storage management operations like delete and instead rely on implicit storage management in the form of *garbage collection*. If the fat-elements method is to provide arrays for functional languages, we need to show that it is amenable to garbage collection.

This topic appears to be underexplored in the literature. Discussions of garbage collection rarely examine "complex" data structures that are non-trivial to garbage collect, and discussions of persistent data structures usually assume that all data will need to persist indefinitely, rather than just a subset of that data. Garbage collection for persistent data structures is not mentioned in any of the works on the topic known to me (Overmars, 1981, 1983; Cohen, 1984; Driscoll et al., 1989; Dietz, 1989; Chuang, 1992, 1994).

The following sections require a passing familiarity with the most common algorithms for garbage collection. Reviews of these topics can be found in Richard Jones's excellent book on the topic (1996) and Paul Wilson's review paper (1994).

4.1 Logical Size versus Actual Size

In the preceding chapters, we have seen how our functional-array technique can store $O(v)$ versions of an n-element array in $O(v+n)$ space (assuming that the array elements themselves are constant-sized), where each array version is made by modifying a single element of the array. Since each array version is seen externally as being self-sufficient, the logical view of these array versions is of $O(v)$ arrays of size n, or $O(vn)$ space.

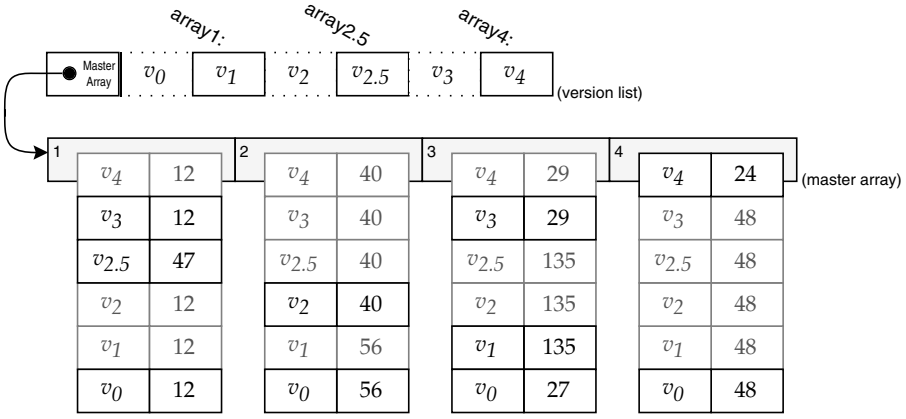
The fat-elements method, like other persistent data structures, achieves this "miracle" of packing what appears to be $O(vn)$ array elements into $O(v+n)$ space by relying on sharing: Each array version is created by altering one array element of another array version and leaving the other $n-1$ array elements unchanged.

When we introduce garbage collection, however, the picture changes. If array versions are removed, the differences between array versions may become greater. In the worst case, each remaining array version may have no elements in common with other array versions. Thus, it is quite possible (and reasonable) to have $O(v)$ versions of an n-element array occupying $O(v+n)$ space before garbage collection (i.e., averaging to constant space per array version), and $O(w)$ array versions (where $w \ll v$) that require $O(wn)$ space left after garbage collection(i.e., $O(n)$ space per array version).

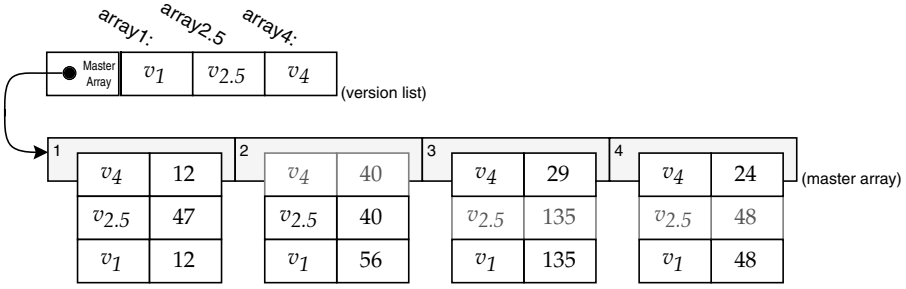
These issues are illustrated by Figure 4.1. In this example we see six array versions, three of which are garbage. But removing these three versions from the master array only allows one fat-element entry to be reclaimed—all the other fat-element entries are needed. I will leave determining *which* one as an exercise for you, the reader, as a prelude to the next section.

4.2 Persistent Structures Are Hard to Garbage Collect

In the previous section, we saw that reclaiming fat-element entries that are not required may result in only small space savings. The exercise at the end of the previous section should have also demonstrated that determining which fat elements can be reclaimed



(a) An array data structure containing some arrays that are no longer referenced externally. The array structure contains entries for version stamps v_0, v_2, and v_3, which may need to be either updated or removed, since their associated array versions are no longer referenced.



(b) The array data structure after removing the superfluous array versions. Notice that in this example it was only possible to delete one fat-element entry. The other six entries associated with v_0, v_2, and v_3 could only be adjusted to use one of the remaining version stamps.

Figure 4.1: Garbage collecting a functional-array data structure.

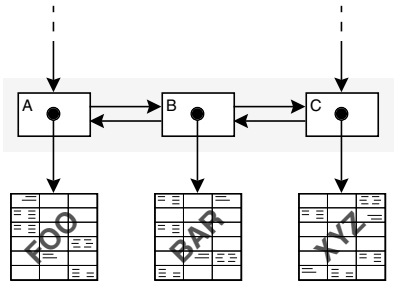


Figure 4.2: Topological reachability $\neq$ liveness.

and which need to be updated is not necessarily easy.[1] This problem is not specific to fat-element arrays; it also applies to the garbage collection of many non-trivial data structures.

The core idea behind garbage collection is to determine which objects are *live* and which are not—live objects will be accessed again at some future point during the program's execution, whereas objects that are not live will not be accessed again and can be reclaimed. Typically, garbage collectors used in functional languages use *tracing* to determine liveness. The fundamentals of tracing garbage collection are simple: Follow the pointers from objects that are known to be live and everything you can reach is also live. This assumption, that *topological reachability* closely approximates liveness, is fundamental to tracing garbage collectors. Unfortunately, there is no guarantee that this liveness heuristic works well for every data structure.

Figure 4.2 illustrates the problem of whether topological reachability equates to liveness. Should the data marked "BAR" be garbage collected? According to the rules for topological reachability, it is live because it can be reached (e.g., A is live and has a link to B and B links to BAR). But what if the double links between nodes are never used as a route into the data?—perhaps the most appropriate action for the garbage collector would be for it to delete node B and link nodes A and C together. Without a deeper understanding of what this data structure actually *means*, it is not possible to properly garbage collect it.

The fat-elements data structure provides another, more complex, example of the

1. The fat-element entry that can be reclaimed is the one corresponding to version v_0 for element 3 of the array.

failings of the topological-reachability liveness heuristic. Determining the liveness of fat-element entries is not a simply a matter of following pointers. Similar problems also affect other persistent data structures, including the persistent arrays proposed by Hughes (1985), Aasa *et al.* (1988), Chuang (1992; 1994), Dietz (1989), and Cohen (1984).

Only tree-based implementations of arrays have the simple topology that allows garbage collection based on topological reachability, which is perhaps just as well because trees have far worse space behaviour than the approaches listed above, typically requiring $O(vn \log n)$ space to store $O(vn)$ array versions.

4.3 Using Traditional Tracing Garbage Collection

So far, we have learned that applying garbage collection to the fat-elements data structure is non-trivial and may not recover much space. So perhaps we should ask ourselves what will happen if we do nothing to address the specific needs of fat-element arrays and only use a traditional tracing garbage collector that does not understand the structure of fat-element arrays.

A tracing garbage collector that provides no special support for fat-element arrays will correctly reclaim a master array if and only if all the array versions represented by that array are unreferenced. But, if even one array version is referenced, the garbage collector will consider all $O(n)$ array versions contained in the same master array as also being referenced. If the elements of the array are simple fixed-size data types (such as integers or floating-point numbers), the uncollected garbage amounts to a constant-factor space overhead.

For arrays with elements that are themselves larger data structures (especially elements that are not constant-sized), retaining garbage array elements may be unacceptable, so in the following sections we will look at specific techniques for performing garbage collection on fat-element arrays.

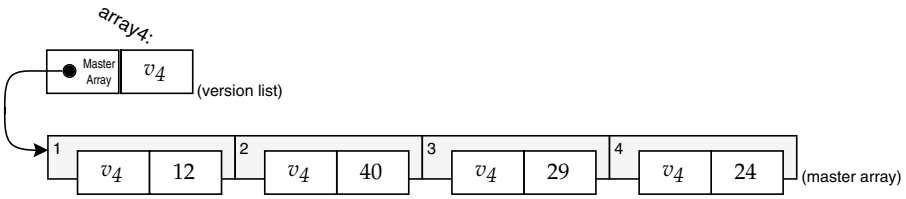
4.4 Garbage Collecting Array Versions Individually

In the previous section, we saw that a tracing garbage collector that does not understand the fat-elements data structure is inadequate for some applications of fat-element arrays. In this section, we will show how *copying collection*, a common garbage collection scheme, can be extended to support the fat-element array representation, as well as the pitfalls of such an extension. The techniques we will discuss can also be applied to other tracing garbage collectors, including mark/sweep collectors and generational collectors. (Garbage collectors based on reference counting require different techniques, which are covered in Section 4.6.)

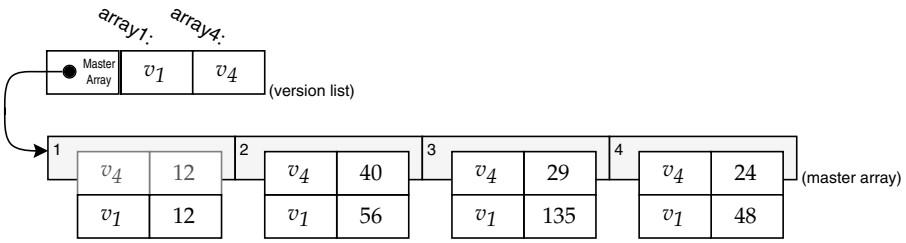
If we extend our garbage collector so that it can find the element values for an individual array version, we can ensure that the collector only copies the versions from the master array that remain referenced. The copying process itself is straightforward, following the same lines as the array creation, access, and update operations described in earlier chapters. Figure 4.3 shows an example of the copying process.

During copying, it may be possible to undo some of the fat-element entry duplication that occurs for fully persistent updates (described in Section 2.2). If an array version corresponding to a fully persistent update is no longer referenced, and the fat element entries above and below the unreferenced fat-element entry both hold the same value, we can eliminate the entry with the greater version stamp because the value it provides can be inferred from the fat-element entry for the lesser version stamp. In Figure 4.3(b), we see that the fat-element entry that had been used to infer the value for array4[1] does not need to be copied because the value for array4[1] can also be found using the fat-element entry corresponding to version stamp v_1—in the original array, this entry is obscured by the entry for array2.5[1], but at this point in the garbage-collection process there is no indication that array2.5 is referenced.

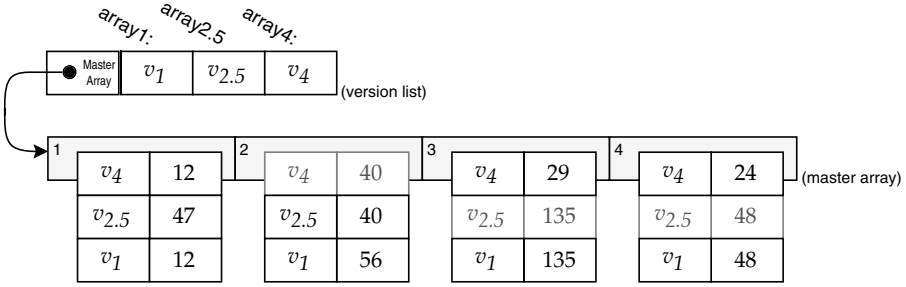
The easiest way to undo this duplication is to use a simple pointer-equality check on fat-element values that are about to be copied to make sure that their values are really needed. An alternate strategy is to perform a redundancy-elimination pass over the array after garbage collection is complete. This latter approach avoids the small constant-factor time overhead from eliminating apparent redundancy only to discover later that the fat-element entry is required after all (a situation that is shown in Figure 4.3(c)). Both



(a) In this example, we continue the situation shown in Figure 4.1(a). This figure shows the point during collection where the garbage collector has found a reference to array4, and this array version has been copied. In Figure 4.1(a), v_4 only had one explicit fat-element entry; most array elements for v_4 were inferred from earlier ones. Now all the entries must be explicit.



(b) At this point, a reference to array1 has been found, and the fat-element entries relating to v_1 have been incorporated into the copy. With this addition, one of the previously explicit v_4 entries can be removed, since it can now be inferred from the earlier entry for v_1.



(c) Finally, a reference to array2.5 is found, and its fat-element entries are added. We now need to make one v_4 entry that was previously inferred explicit.

Figure 4.3: Garbage collecting a functional-array data structure.

methods may sometimes perform additional—harmless—optimizations to the master array by eliminating other kinds of redundancy (e.g., a duplicated array value arising from updating an array element to hold the same value it already held).

Although the final result of this simple copying approach is a smaller structure, the asymptotic time complexity of this approach is poor. As we know from Figure 4.1, a master array typically has a much larger virtual size than its actual size in memory—many of the elements of a particular array version are inferred from earlier values, rather than being defined explicitly. If we copy each active array version individually as they are encountered, the time taken will be at $O(rn)$, where r is the number of versions of the array that remain referenced, and n is the size of the array. (If splay trees are used to represent fat elements, this time bound is an amortized bound, otherwise it is a worst-case bound.)

If the number of array versions that remain referenced for each master array is a constant, then this level of performance may be acceptable. For example, in single-threaded array algorithms, only the most recent array version will remain referenced; thus, the constant is one.

In the worst case, however, $r \in O(n)$, and therefore the r referenced array versions will require $O(n^2)$ time to be garbage collected. Thus, garbage collecting a functional array using this technique may take time proportional to the virtual size of the array versions, as opposed to the amount of memory they actually occupy. Ideally we would prefer a garbage-collection technique that only took time proportional to the actual size of the fat-elements data structure.

4.5 Deferred Copying

The problem with the simple array-version copying approach is that it needs to traverse the entire master array each time an array version is copied, at a cost per traversal of $O(n)$ (amortized) time. The only way to avoid this cost is to avoid traversing the array data structure for every array version that is encountered.

A solution to the problem of repeated array traversals is to defer copying array versions for as long as possible (as shown in Figure 4.4). Whenever an array version is

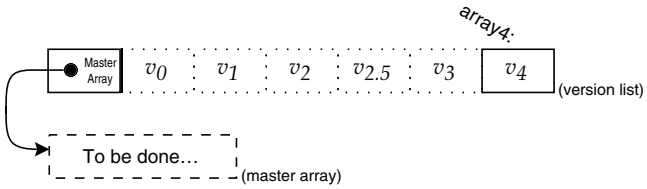
found, we tag its version stamp, copy the array version pointer, add the master array to a "deferred queue", and then continue with any other copying that needs to be done. When there is nothing else to copy, we copy a master array from the deferred list, copying only the values belonging to the tagged version stamps.

This copying process works as follows: First we replace the version stamp for each unreferenced array version u with an indirection to the nearest version stamp greater than u that belongs to a referenced array version—if there is no such version stamp, u is marked as irrelevant. (This procedure takes time proportional to the number of version stamps that exist for that master array; because there are at most $O(n)$ of these version stamps, this process takes $O(n)$ time.) Then we copy each of the fat elements of the master array: For each fat element, we copy the fat-element entries in reverse order (i.e., starting with the entry with the greatest version stamp and ending with the entry with the least version stamp), using the indirections to map version stamps from unreferenced arrays to version stamps belonging to referenced array versions. If this mapping produces a version stamp we have already encountered or an "irrelevant" version stamp, we skip that entry and move to the next one. Copying fat element i of the array in this way requires $O(e_i)$ time; thus, copying the entire array requires $\sum_{i=1}^{n} e_i = c_*$ time, and $c_* \in O(n)$.

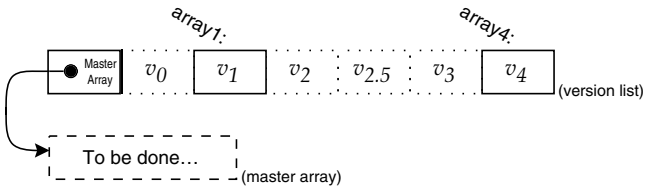
For example, consider Figure 4.4(d). The next fat element to copy is element 3 (shown in uncollected form in Figure 4.1(a)). We begin with the $(v_3, 29)$ fat element entry, which is remapped to be $(v_4, 29)$ because v_3 is the version stamp of an unreferenced array version. Then we copy $(v_1, 135)$ unchanged (because v_1 belongs to a referenced array version). Finally, we come to $(v_0, 27)$, which would be mapped to $(v_1, 27)$, but is discarded because we have already copied an entry for v_1. Thus, we produce the final array shown in 4.4(e).

Thus, by waiting to copy the array until the last moment, we can copy all of the referenced array versions from that master array in $O(n)$ time. This time contrasts strongly with the $O(n^2)$ worst-case time for garbage collecting a master array under the simple scheme described in the previous section.

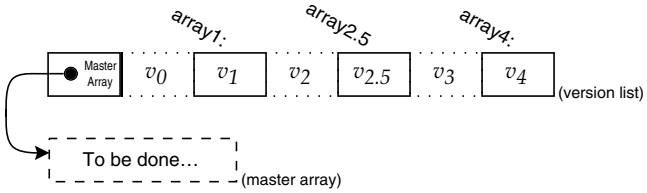
Unfortunately, the $O(n)$ time bound for the deferred copying technique only applies to arrays which are not nested in complex ways. We discuss the problems of nested arrays in the next section.



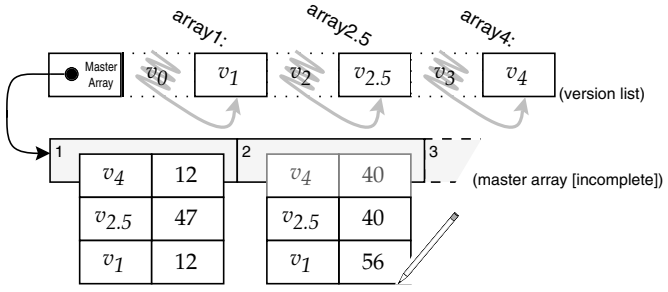
(a) As in Figure 4.3(a), the collector has found array4. In this case, however, instead of copying the array version associated with array4, we flag its version stamp, v_4, and then continue. We defer copying the array data structure until all other garbage collection is complete.



(b) The collector now encounters array1 and flags its version stamp, v_1. Copying of the array data structure remains deferred.

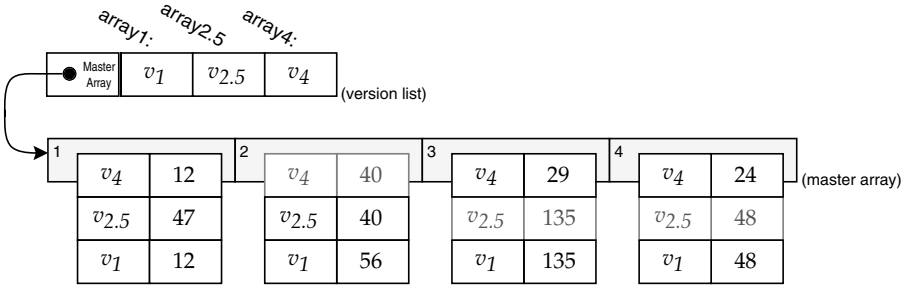


(c) Similarly, when the collector encounters array2.5, all that is done is to flag its version stamp.



(d) Once other garbage collection is done we copy any array data structures we had deferred. When copying the structure, we only copy the entries that relate to the version stamps we have flagged. This figure shows the copying process part way through.

Figure 4.4: Deferred copying improves performance. (*continued over...*)



(e) Finally we complete the copying process. Only the fat-element entries and version stamps corresponding to referenced array versions have been copied.

Figure 4.4 (*cont.*): Deferred copying improves performance.

4.5.1 Difficulties with Deferred Copying

When the live portion of a master array is copied, the element values for all of the live array versions must also be copied. In copying these values, we may find a good deal more data that had hitherto been thought to be unreferenced. If fat-element arrays can be (directly or indirectly) nested inside other fat-element arrays, we have the possibility that as we copy the array-element values we will discover a previously undiscovered array version from a master array that has already been copied—meaning that the copy is incomplete.

There are two alternative ways of handling this situation. The first is to throw away the incomplete copy of the array, add the array to the end of the deferred queue, and copy it again later; when we copy it again, we will include the previously omitted array values.[2] The second option is to use the simple copying technique we discussed in Section 4.4 to add the missing array version to the existing copy of the array data structure.

From a computational-complexity perspective, the first solution is no worse than the second, because copying the master array requires $\Theta(n)$ time, which is no worse than the $\Theta(n)$ time required to copy a single array version using the second method. Thus, in the pathological case where every deferred copy reveals another array version that needs to be copied, the time required is $O(n^2)$, which is no worse than the worst case for copying array versions individually.

2. In reality, we probably would not throw away the incomplete copy, but merely augment it when we perform subsequent deferred copies.

4.6 Using Reference Counting or Explicit Deletion

Some languages use reference counting to provide garbage-collection functionality. Reference counting is based on significantly different principles than tracing garbage collection. In reference counting, allocated data is deleted when it ceases to be referenced (rather than waiting to be discovered in the next garbage-collection cycle). Thus, to support garbage collection via reference counting, the fat-elements data structure needs to support the deletion of individual array versions.

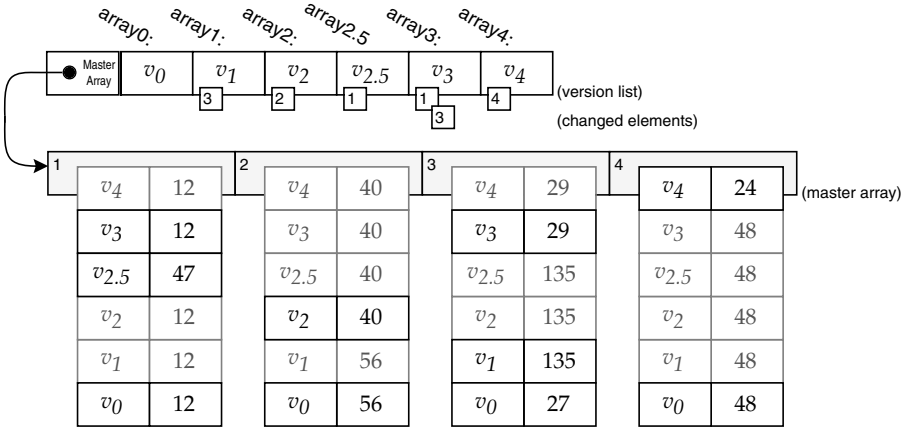
Providing deletion of individual array versions is easy if we allow the deletion process to take $O(n)$ time, where n is the size of the array. We can traverse the array, and each fat-element entry tagged with the version stamp belonging to the unwanted array version is either relabeled with the version stamp of its successor in the version list (if there is no entry for that version stamp in the fat element) or deleted (if there is such an entry or there is no successor).

Ideally, however, we would prefer to avoid traversing the array to find and update only a few of its fat-element entries. We can avoid such a traversal if we store additional housekeeping information in the master array that records exactly which array elements are changed in each successive version of the array. Figure 4.5 provides an illustrative example of deleting specific array versions.

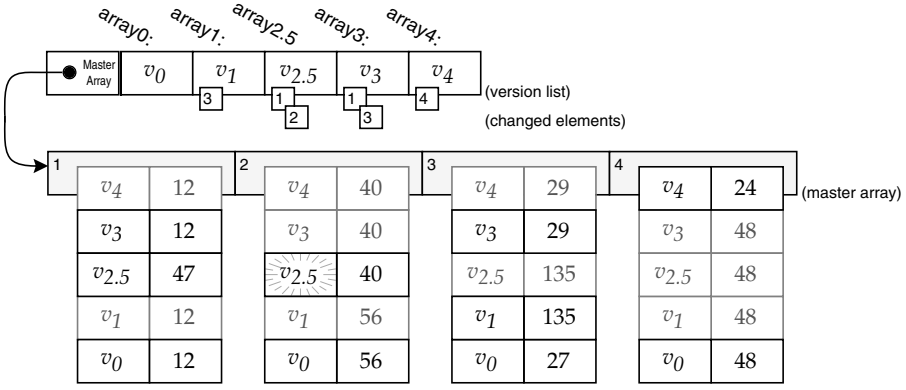
In this revised scheme, we associate a list of the elements that are changed in each array version (as compared to its predecessor) with the version stamp for that array version. By keeping this information at hand, we can avoid examining fat elements that will not need any changes, which, in the usual case, will be most of them.

Figure 4.5 also shows a second optimization to the deletion process: Sometimes an array version provides a large number of element values for subsequent array versions, making updating the fat-element entries a costly task. If the successor to the deleted version defines fewer element values, it is cheaper to base the update process on the successor's list of changes, rather than the deleted array's list of changes.

With this optimization, we can support deletions that mirror a single-threaded array access pattern with constant-time overhead. In the worst case, deletion costs $O(e \log n)$, where e is the number of element values defined in that version.

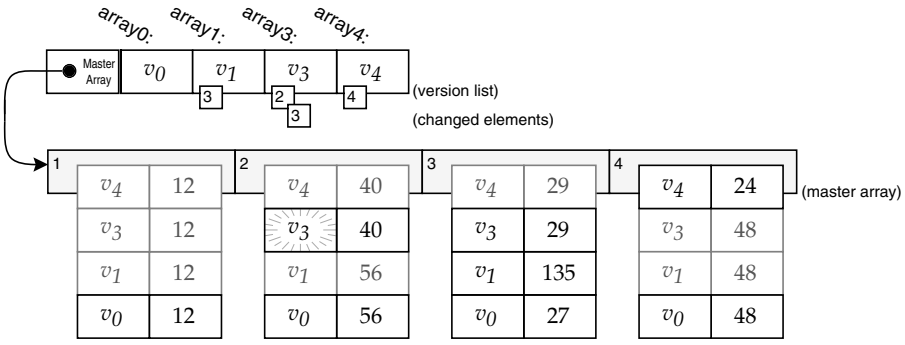


(a) A fat-element array with housekeeping information to support deletes. In previous chapters, we stored only the number of changes made with each version, but to support explicit deletes we also need to store exactly which array elements are changed in each array version. This housekeeping information is redundant, as it could be found by simply traversing the each element of the master array, but such a traversal would require $O(n)$ time.

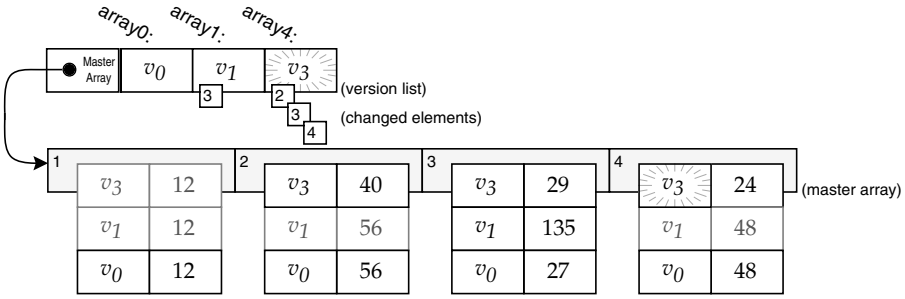


(b) The array after deleting array2. Although array2 is no longer accessible, no information is actually removed from the master array because the changes made in array2 are required by subsequent array versions. The version stamp for array2, v_2, is deleted from the version list, and the fat-element entry for array2[2] is updated to attribute its contents to array2's successor in the version list, $v_{2.5}$.

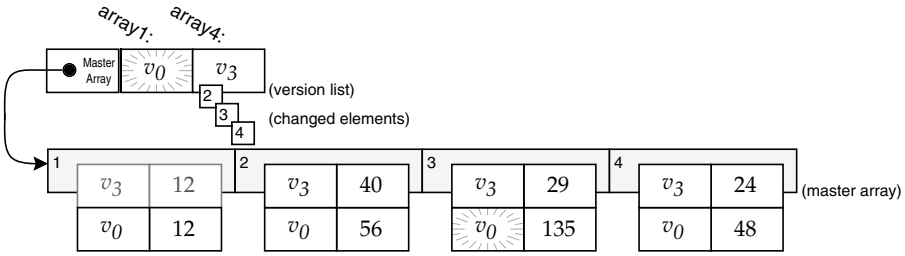
Figure 4.5: Explicit deletion of array versions. (*continued over...*)



(c) The array after deleting array2.5. In this case, we can remove the element values corresponding to array2.5[1] from the master array because no subsequent array versions depend on it—in fact, when the entry for array2.5[1] is removed, the entry for array3[1] is also removed because it is the same as its predecessor, array1[1]. The fat-element entry for array2.5[2] cannot be removed because it is needed by array3, so it is retagged with v_3.



(d) The array after deleting array3. The values for array3[2] and array3[3] are needed by array4, so these values cannot be deleted. Although we could update these fat-element entries to be tagged with v_4, it is less work to discard the v_4 version stamp and adopt v_3 as the version stamp for array4. By doing so, we only have to retag one fat-element entry rather than two.



(e) Finally, array0 is deleted. As in the previous example, it is less work to adjust array1 to use v_0 as its version stamp than to traverse the array and change fat-element entries tagged with v_0 to be tagged with v_1.

Figure 4.5 (*cont.*): Explicit deletion of array versions.

4.7 Conclusion

In this chapter, we have developed techniques that can garbage collect a non-nested master array of size $\Theta(n)$ (containing $O(n)$ array versions) in $\Theta(n)$ time. Arrays that are nested within arrays may take longer than this time bound, and in the worst case a master array of size n may contain nested references to n versions of itself and thus require $O(n^2)$ time to collect—in other words, it may take time proportional to the virtual size of the collected arrays rather than their actual size in memory. This pathological case is fairly unlikely for most normal uses of arrays.

Thus, in normal use, the fat-elements data structure is very amenable to garbage collection, even if the process is somewhat more complex than the process for simpler data structures (in particular, we must have some mechanism at our disposal to make the garbage collector "smarter" about specific data structures). The complexities that arise for fat elements are not unique to this data structure—other data structures whose link topology does not reflect their liveness may suffer similar problems under garbage collectors that use simple tracing techniques. Thus, it seems reasonable to expect that similar strategies to the ones we have discussed may be applicable to other data structures.

Chapter 5

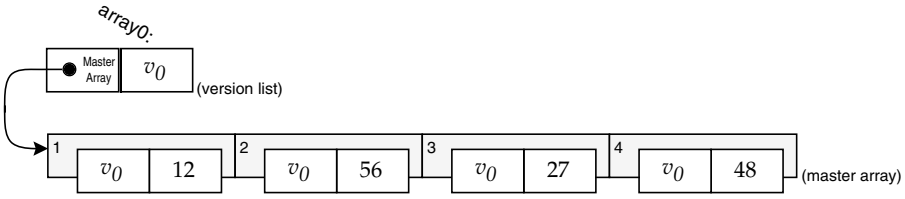
Optimizations to the Fat-Elements Method

This chapter sketches some useful optimizations that can be applied to the fat-elements method for functional arrays. Some of these optimizations can improve performance without requiring any changes to programs that expect a typical functional array interface, whereas others require that programs adopt new strategies for using functional arrays. Although these optimizations do not improve the theoretical complexity of the fat-elements method, they can be useful in reducing the overheads involved in using fat-element arrays in practice.

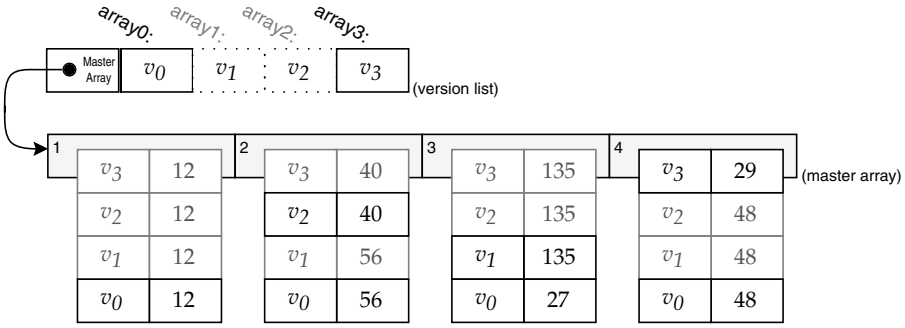
5.1 Batched Update

Currently, to change three elements of an array we must make three calls to the array-update function and create three new versions of the array. Figure 5.1 shows the results of executing the following code to change three elements of an array:

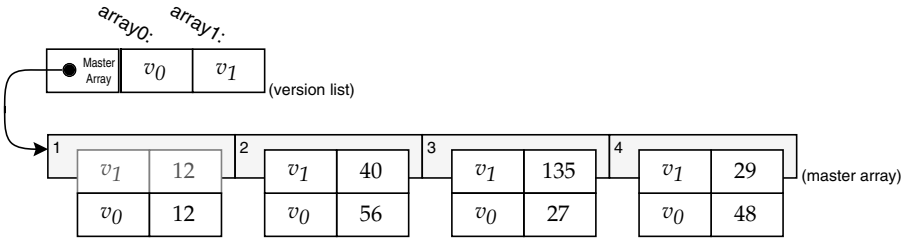
```
let array1 = update(array0, 3, 135)
    array2 = update(array1, 2, 40)
    array3 = update(array2, 4, 29)
in  array3
```



(a) The initial contents of the array.



(b) The array after three updates using the conventional update mechanism. Even though array1 and array2 were only created as stepping stones to array3, they nevertheless exist in the master array as separate array versions. These array versions will only be removed when garbage collection takes place.



(c) This figure shows the array after a three-element batched update. Note that only one version stamp has been allocated but three element values have been changed.

Figure 5.1: Batched array update.

Because the two intermediate versions of the array that are created by these calls to update will never be used afterwards, it is somewhat wasteful to create them.

If we extend the array interface to allow batched array updates, we can avoid creating superfluous array versions. Figure 5.1(c) shows an array that has undergone the following update sequence:

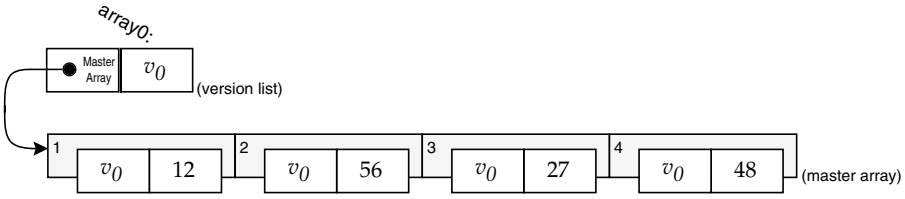
```
let array1 = batch_update(array0, [(3, 135), (2, 40), (4, 29)])
in  array1
```

Here a new function, batch_update, is used. This function takes as its argument an array version and a list of changes to make and returns a new array version with the specified changes. It operates in the obvious way: Like the single-element update described in Section 2.2, it allocates a single version stamp and then applies all of the updates given in the list (thus the only difference between this function and the update technique described in Section 2.2 is that batch_update changes several array elements, whereas update changes just one). Similarly, the usual rules for array splitting apply—if adding the newly created array version would make the master array "too fat", the master array is split into two independent master arrays (see Section 2.3 for complete details on the splitting process).

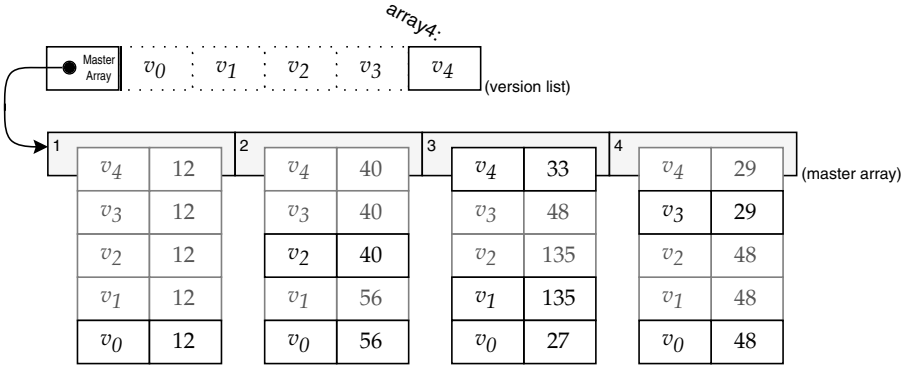
In addition to this batch_update function, it may also be appropriate to provide other operations that process many array elements en masse. For example, the arraymap function discussed in Section 1.3.1 can also avoid wasteful creation of array versions.

5.2 Destructive Update

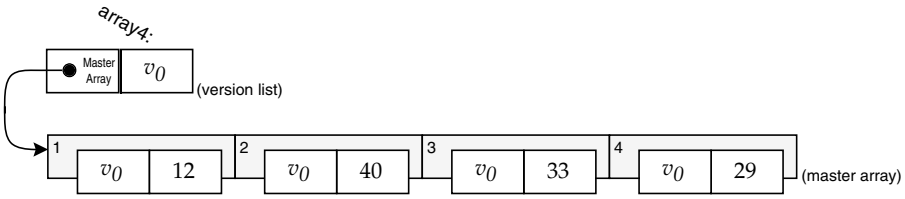
As we discussed in Chapter 1, array algorithms born in the imperative world access data single threadedly, regardless of whether they act on an ephemeral imperative array or a persistent functional array. In fact, even algorithms that use arrays persistently may have sequences of single-threaded updates—the update sequence we examined in the previous section is just one example of a single-threaded access sequence (after each update, we only accessed the newly created array version, and discarded the old



(a) The initial contents of the array.



(b) The array after four updates using the conventional update mechanism. The previous array versions remain in the master array even though they are no longer referenced and will only be removed when garbage collection takes place.



(c) This figure shows the array after the same sequence of single-threaded updates as (b), but this time destructive update was used. Note that no new fat-element entries have been allocated and the version stamp remains the same.

Figure 5.2: The promise of destructive update.

one). Although the fat-elements method provides good (amortized constant-time) performance for single-threaded array use, it is worth considering whether we can improve the performance of single-threaded array algorithms further.

The same techniques we discussed in Sections 1.2.1, 1.2.2, and 1.3.2 that allow us to use destructive update on ordinary arrays (i.e., linear types, monads, update analysis, and single-bit reference counting) can also be applied to functional-array data structures such as the fat-element arrays. In the case of monads or linear types, a single-threaded array-access sequence is directly specified by the programmer; whereas in update analysis and single-bit reference counting, single-threaded update sequences are discovered by the program's support environment (the compiler for update analysis and the runtime system for single-bit reference counting). For example, we could allow monadic array-access sequences to be applied to functional arrays by providing a function with the following interface:

$$\mathsf{apply\_arrayop} \quad : \quad \big(\mathit{array\text{-}op}\,(\alpha,\beta),\mathit{array}\,(\alpha)\big) \rightarrow \big((\mathit{array}\,(\alpha),\beta)\big)$$

This function applies a monadic operation to an existing array and yields a new array version and any other result returned by the monadic code. As with the batch_update function discussed in the previous section, the operation of this function is straightforward: A new version stamp is created for the new array version, r, that will be returned by apply_arrayop, then the monadic array operation is executed. Each single-threaded update performed by the monadic code provides a new value for some element of r. Because the monadic code executes before r has been returned by apply_arrayop, the changes made to r by the monadic code cannot be observed by functional code. Similarly, the monadic code is allowed to perform conventional functional-array updates and include the resulting array versions in its return value (the type variable β in the interface shown above).[1]

1. In a lazy functional language, where the batch_update function takes a lazy list (or stream) of array updates to make, it is also possible for ordinary persistent updates to take place during a batch update sequence. The batch_update primitive is not quite as powerful as apply_arrayop, however, because it does not allow the new array version to be read while the updates are partway through—there is no way to access the new array until it is returned. The monadic version, on the other hand, can use the read array operation to access the resulting array version as it is being formed.

An advantage of single-threaded access sequences is that they may allow us to destructively modify an existing fat-element entry rather than add a new one. We can only overwrite fat-element entries that are exclusively associated with the specific array version we wish to update and not used to infer values for other array versions. This property is true of all the entries in the example shown in Figure 5.2, but need not always be true. When we cannot overwrite a fat-element entry, we must add a new entry following the rules outlined in Chapter 2. Figure 5.3 gives an example of destructive updates that could not use fat-element overwriting.

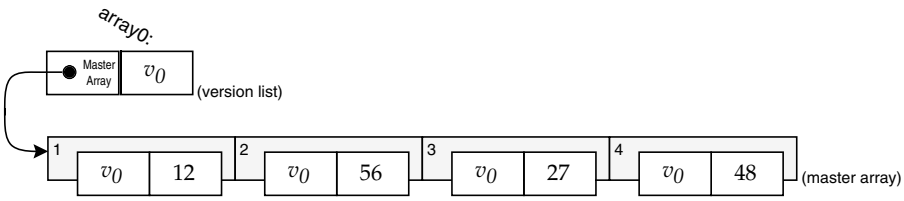
To support destructive updates, we simply need to extend the interface to the fat-elements method with a procedure, destructive_update, that safely changes an element value in an existing array version (i.e., without harming the values stored in prior or subsequent array versions). The destructive_update procedure cannot be made available to programmers in languages that guarantee referential transparency, but it can be called by programming-language support code as part of any facility that detects or enforces single-threadedness.

5.3 Single-Entry Compression

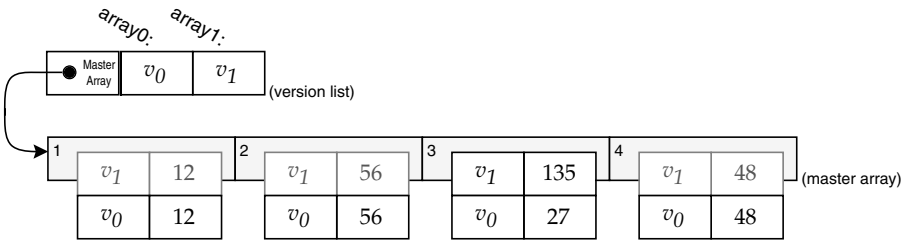
We will now examine a technique that aims to reduce the amount of space required to represent fat elements. Each fat element is a tree-based data structure that provides a mapping from version stamps to values. When a fat element stores several different values, a tree is a fairly efficient data structure, but when a fat-element entry holds only a single value, a tree-based implementation may be less space efficient than we desire.

In a typical implementation, a tree node representing a fat-element entry will contain pointers to left and right subtrees, a version-stamp pointer, and a value (or value pointer). In contrast, an array element in an imperative array would just store the value (or value pointer). Thus, if a master array happens to contain only a single array version, it may use more than four times the space of an imperative array (which, by its very nature, can only store a single array version).

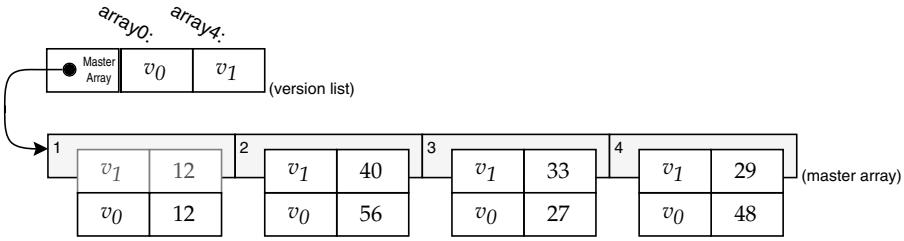
When there is only a single entry in a fat element, maintaining a data structure to map version stamps to values is unnecessary because all version stamps map to the same



(a) The initial contents of the array (same as those in Figure 5.2(a)).

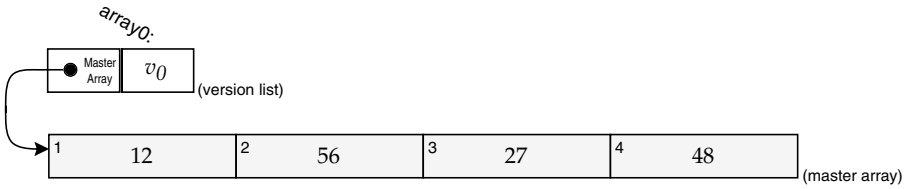


(b) The array after one update. Destructive update cannot be used for this update because array0 is still referenced, and thus the update is not single threaded.

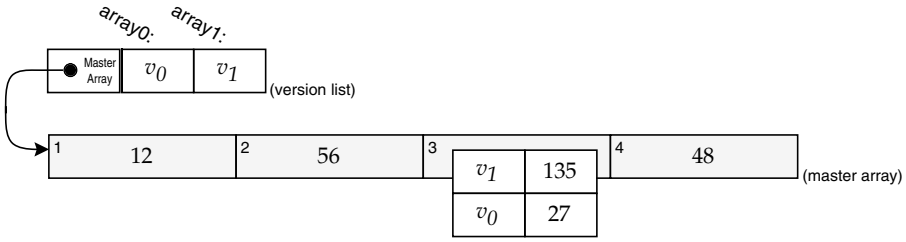


(c) This figure shows the array data structure after a sequence of three single-threaded updates to array1. Although we have been able to avoid allocating any new version stamps for these array versions, and have been able to use overwriting in creating array4[3], the other two updates required fat-element additions (following the procedures outlined in Section 2.2).

Figure 5.3: A more typical case of destructive update.



(a) When all array versions have the same value for an array element, we can avoid the cost of the full-blown fat-element data structure.



(b) When one element of the array is updated, we must create a fat element for that array element. The other array elements remain unchanged.

Figure 5.4: Single-entry compression.

value. Instead, we can defer creation of the fat-element data structure and replace the pointer to the top fat element in the tree with the value that the single entry would have stored (see Figure 5.4). This technique requires us to able to distinguish between fat-element pointers and values that may be stored in the array. Systems that use runtime type tagging may be able to make the necessary distinction with no overhead; otherwise, we may require an additional array of single-bit flags that mirrors the master array.

5.4 Conclusion

In this chapter, we have examined optimizations that may yield small but useful constant-factor gains over the general fat-elements method. These optimizations are orthogonal: Batched array-update facilities may be present in a system that uses update analysis to detect other single-threaded array accesses and provides facilities that make single-entry compression easy to implement. When update-in-place and single-entry compression optimizations are applied together, single-threaded algorithms may run at close to the same speed they would with imperative arrays, and without significant memory turnover.

Part II

Deterministic Asynchronous Parallelism

Chapter 6

Background for Determinacy Checking

Parallel programming has a reputation for being more difficult than serial programming, and with good reason. Parallelism not only provides new kinds of mistake for programmers to make (in addition to the numerous mistakes that can occur as easily in parallel code as they can in sequential code), but these new errors may be difficult to discover and correct.

Testing and debugging parallel programs that have multiple threads of control is notoriously difficult. Timing issues and scheduler behavior can mask problems, resulting in code that appears to be free of bugs during testing but fails mysteriously at some later time. Tracking down such problems with conventional debugging tools can be difficult; the particular interaction between threads that triggers the bug may be nearly impossible to reproduce reliably.[1]

One approach to address the problem of accessing data safely in a parallel program is to use a locking mechanism to serialize access to vulnerable data. These techniques—such as *semaphores* (Dijkstra, 1968) or *monitors* (Hoare, 1974)—are well known, but locking is not a panacea. For example, programmers working with locks must guard against

1. In my own experience, I have encountered bugs in parallel code where the window of opportunity for two threads to interact improperly was tiny (about 5 ns, the time to execute one machine instruction). On a uniprocessor machine, the code *never* showed any errors, but on a four-processor machine, the error showed up once in every twenty-four hours of continuous execution of the program. In addition, when the program did finally fail, it was not at the point where the erroneous access actually occurred but much later.

deadlock, while also ensuring that their locking strategy preserves their program's parallel scalability.

More importantly, locking does little to ensure that a program's data dependencies are safe. A simple locking strategy may prevent task *A* from reading data before task *B* has finished writing it, but such a strategy does not address the question of whether task *A* should even be allowed to depend on data written by task *B*. A parallel program that is written without regard to intertask data dependencies—even one that uses locks to protect data—can suffer determinacy problems, causing program failures or unexpected variability of output.

In this part of my dissertation, I will show how *Bernstein's conditions* (1966) can be checked at runtime, ensuring that a program behaves deterministically. Data that is accessed in compliance with Bernstein's conditions avoids read/write and write/write contention between tasks and does not need to be protected by locks. On a multiprocessor shared-memory machine with processor caches, obeying Bernstein's conditions may also reduce interprocessor memory contention, eliminating one possible source of poor performance.

In the remainder of this chapter, I will define the terminology that underpins the discussion in this and subsequent chapters, then review prior work on the topic of determinacy checking. Chapter 7 describes the LR-tags method—my technique for determinacy checking—and the basic theory that underlies it. Chapters 8 and 9 discuss additional theoretical aspects of the LR-tags algorithm: Chapter 8 provides a correctness proof; Chapter 9 discusses the algorithm's time and space complexity. Chapters 10 and 11 discuss more practical matters: Chapter 10 describes some useful optimizations to the LR-tags technique; Chapter 11 examines the performance of an implementation of the algorithm when running realistic benchmarks. Finally, Chapter 12 presents my conclusions and outlines opportunities for further work.

6.1 Describing Determinacy

Various terms have been used to describe parallel determinacy problems, including *harmful shared-memory accesses* (Nudler & Rudolph, 1986), *race conditions* causing *indeterminacy* (Steele, 1990), *access anomalies* (Dinning & Schonberg, 1990), *data races* (Mellor-Crummey, 1991), and *determinacy races* (Netzer & Miller, 1992).

Listing 6.1: Examples of parallel programs exhibiting determinacy and indeterminacy. The first program exhibits indeterminacy, whereas the middle two are clearly deterministic. The fourth program exhibits internal indeterminacy while remaining externally deterministic.

(a)

```
a := 2;
cobegin
   b := a;
   a := a + 1;
coend;
```

(b)

```
a := 1;
b := 2;
cobegin
   c := a + b;
   if a > b then
      d := a;
   else
      d := b;
coend;
cobegin
   a := c / 2;
   d := b + c + d;
coend;
```

(c)

```
a := 1;
b := 2;
cobegin
   begin
      c := a + 3;
      cobegin
         a := (a + b) / c;
         d := b / c;
      coend;
   end;
   e = (b / 2) + 1;
coend;
```

(d)

```
cobegin
   a := -2;
   a := 2;
coend;
a := a * a;
```

Feng and Leiserson (1997) list most of the above terms and recommend the term "determinacy race". We will adopt this term for our discussion, but also follow Steele's preferences by using "indeterminacy" to describe a problematic lack of determinacy caused by determinacy races and "nondeterminacy" to describe a lack of determinacy intended by the programmer (such as McCarthy's nondeterministic *amb* operator (1963)).

Programs do not need to be fully deterministic to generate consistent results from run to run—for example, an operating system may run unrelated tasks in nondeterministic order without causing any problems; or a search algorithm may choose its initial search direction nondeterministically and yet generate unwavering results. Thus programs can be *externally deterministic* even if they are not *internally deterministic* (Emrath & Padua, 1988) so long as the operations that are performed in nondeterministic order *commute* (Steele, 1990).

Checking external determinacy is difficult. For example, suppose $m = 2.0$ and we perform two atomic actions, $m \leftarrow 1/m$ and $m \leftarrow m - 2.5$, in nondeterministic order. For these values, either order results in $m = -2.0$, although addition and division do not usually commute. (Steele (1990) discusses *commuting with respect to a memory state* versus commuting in general.) Similarly, far more complex instances of nondeterminacy

could eventually cancel out, and the duration of the internal nondeterminacy could be arbitrarily long.

Checking internal determinacy is easier, because we need only be concerned about the local effects of program behavior. Internally deterministic programs are easier to reason about than their externally deterministic and nondeterministic cousins, making debugging easier and formal verification more practical. Even nondeterministic programs can benefit from ensuring that their nondeterminacy is limited to those places where it is intended.

Listing 6.1 provides some code fragments with and without indeterminacy. Listing 6.1(a) shows a simple example[2] of indeterminacy: Inside the cobegin construct, one task reads the value of a while the other task updates a. Listings 6.1(b) and 6.1(c) are both deterministic; how tasks are scheduled has no effect on the final result. Listing 6.1(d) is a rather ambiguous case—the code is externally deterministic, but during its execution, a has a nondeterministic value.

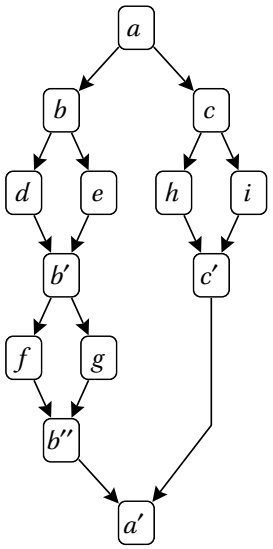
I will focus on the problem of preventing indeterminacy by ensuring internal determinacy. If the programs in Listing 6.1 were tested for internal determinacy, only (b) and (c) would pass the test.

6.2 Parallel Model

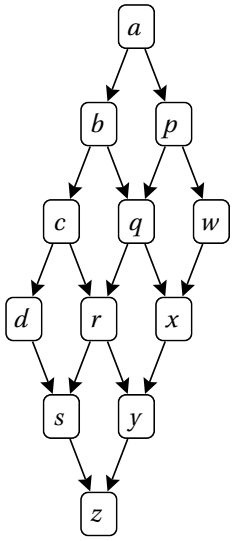
We focus our discussion on mechanisms to provide determinacy checking for programs that execute on a shared-memory MIMD machine (Flynn, 1972). To describe such programs, we will use a model of parallel computation in which parallel programs are composed of *tasks*—sequential sections of code that contain no synchronization operators. All parallelism comes from executing multiple tasks concurrently.

The scheduling dependencies of tasks can be described using a directed acyclic graph (DAG) (Valdes, 1978). When a task y cannot begin until another task x has completed, we place an edge in the DAG from x to y and describe x as a *parent* of y and y as a *child* of x. When a task has multiple children, we call it a *fork node*; when a task has multiple par-

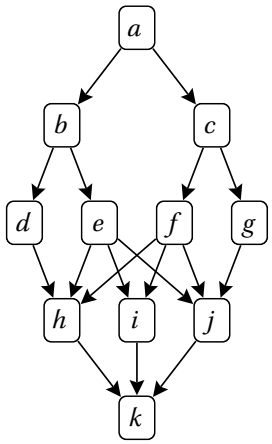
2. All four examples are simple and could be checked for determinacy statically. More complex programs, especially ones involving array accesses with dynamically computed subscripts (e.g., parallel sorting algorithms) can be difficult or impossible to check statically.



(a) Tasks that fit the nested-parallelism model (e.g., cobegin and coend constructs), described using a series–parallel DAG.



(b) A pipeline of tasks. Nodes *a*, *b*, *c*, and *d* together represent a single producer; nodes *p*, *q*, *r*, and *s* represent a (UNIX-style) filter; and nodes *w*, *x*, *y*, and *z* represent a consumer. The producer, filter, and consumer are each broken into four tasks, representing four synchronization points in their execution.



(c) Tasks synchronized in unusual ways. These tasks use a general, nonplanar DAG to represent their synchronization requirements.

Figure 6.1: Parallel task models. In all three graphs, a task can begin only after all tasks that have arrows pointing to it have completed.

ents, we call it a *join node*. Nothing precludes a node from being both a fork node and a join node. Figure 6.1 shows three such graphs with increasingly complex synchronization requirements. The DAG representation of tasks is a *reduced* graph because it contains no transitive edges—transitive edges would be redundant because synchronization is implicitly transitive.

The DAG representation cannot fully represent all kinds of parallelism and synchronization. For example, programs that are inherently nondeterministic, synchronize using locks, or expect lazy task execution, will have some of their parallel structure uncaptured

by the a task DAG.[3] We will restrict our discussion to the kinds of parallelism where the DAG representation is useful.

The scheduling dependencies of a task DAG can also be equivalently expressed using a partial order among tasks. We will use the relation $\trianglelefteq$, pronounced "ancestor of", to denote such a partial order. For two distinct tasks, x and y, $x \trianglelefteq y$ means that y may not begin until x has completed. This relation exactly mirrors the DAG model since $x \trianglelefteq y$ is equivalent to saying that there is a (possibly zero-length) path from x to y in the DAG. We also define a relation $\triangleleft$, pronounced "proper ancestor of", such that

$$(x \triangleleft y) \Leftrightarrow \big((x \trianglelefteq y) \wedge (x \neq y)\big).$$

We define two metrics for a task DAG: The *breadth* of the DAG is the maximum number of tasks that could execute concurrently if the program were run using an infinite number of processors. The *size* of the DAG is the total number of tasks it contains.

We use three terms to describe the life cycle of an individual task from a task scheduler's perspective:

- **Nascent tasks** are tasks that are not yet represented in the task scheduler.
- **Effective tasks** are tasks that must be represented by the task scheduler, because they are executing, waiting to execute, or their children are still nascent. By definition, a task cannot be executed by the scheduler until it has become effective (a scheduler cannot execute a task without representing that task in some way). A task cannot become effective until one of its parents has completed, and must remain effective until it has completed its execution and all its children have ceased to be nascent. This specification of when a task may be effective is deliberately loose—the exact details depend on the scheduler.
- **Ethereal tasks** are tasks that have ceased to be effective, and no longer need to be represented in the task scheduler.

3. It is nevertheless possible to draw a task DAG for such programs. For example, a program in which tasks should only be run when their results are needed (corresponding to lazy evaluation) can use the task DAG to represent the task dependencies, but not the lazy execution requirement. Similarly, sometimes we can draw a more restrictive graph that corresponds to the properties of a given run of the program, rather than the properties of all runs. For example, some nondeterministic programs can be viewed as having a

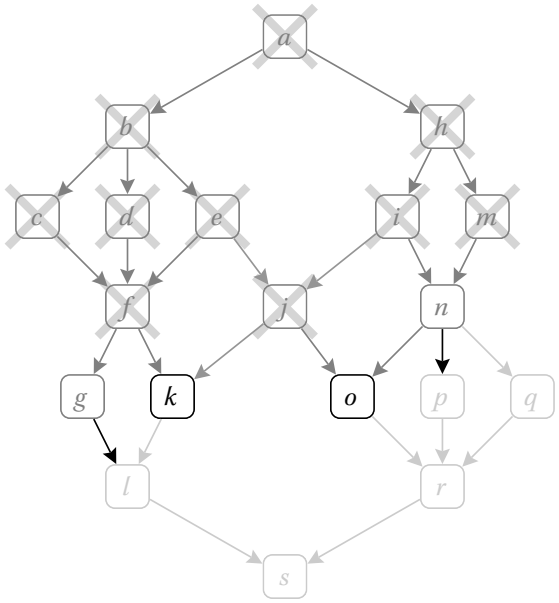


Figure 6.2: Understanding nascent, effective, and ethereal tasks. If we consider a moment in time where the light-grey tasks have yet to execute, the black tasks are executing, and the dark-grey tasks have completed, then all the light grey tasks are nascent and the crossed-out tasks are ethereal. The remaining tasks, *g*, *k*, *n*, and *o*, are effective: k and o are effective because they are executing, whereas n and g are effective because they have nascent children. This is not the only possible arrangement of effective tasks—if l were made effective, g could become ethereal.

The number of effective tasks varies as the program executes (see Figure 6.2), and is a function of both the scheduling-algorithm implementation and the structure of the task DAG. Typically, the peak number of effective tasks would be higher for a breadth-first task scheduler than a depth-first scheduler. For our discussion, we will assume that the task scheduler requires $\Omega(e)$ memory to execute a program, where e is the peak number of effective tasks during execution under that scheduler.

Note that a programming language's definition of a task, thread, process, or coroutine may be different from a task in this model but nevertheless compatible with it. Some languages, for instance, allow a thread to spawn a subthread and allow both the spawner thread and its subthread to execute in parallel, and later have the spawner wait for its

set of DAGS, where the size of this set grows exponentially with the number of nondeterministic choices. In this case, a given run of the program will follow one of the DAGS from the set.

subthread to terminate. As Steele (1990) observes, we can represent such a mechanism in our model by representing the spawner thread as three nodes in the DAG: a fork node, a child node, and a join node, where the child node represents that part of the spawner thread's execution that occurs concurrently with its subthread.

The full generality of the DAG model is not required in many cases. Parallel programs that use *nested parallelism* fit the constraints of a series–parallel DAG (MacMahon, 1892; Riordan & Shannon, 1942; Duffin, 1965), a simple and elegant form of planar DAG (see Figure 6.1(a)). Producer–consumer problems cannot be described using series–parallel graphs, but they can be described using another simple planar DAG form (see Figure 6.1(b)). In both of these cases, the $\trianglelefteq$ relation describes a planar lattice and the DAG is subsumed by a type of planar DAG known as a planar *st*-graph (which we define in Section 7.2). Some parallel programs require an arbitrary nonplanar DAG to precisely represent their synchronization requirements. The graph shown in Figure 6.1(c), for example, cannot be drawn as a planar DAG.[4]

We will devote most of our discussion to the problems faced in checking parallel programs that use nested parallelism, but the LR-tags technique presented in subsequent chapters can also be applied to a more general class of parallel program, including parallel programs whose $\trianglelefteq$ relation forms a planar lattice (see Section 7.2).

6.3 Conditions for Parallel Determinacy

Determining statically whether a general parallel program will avoid parallel indeterminacy problems for all inputs is undecidable.[5] Sufficient conditions for determinacy

4. Astute readers may notice that we can add a node to Figure 6.1(c) to turn it into a planar graph without altering its behavior by adding a node at the point where two lines cross. This transformation is not applicable in general, however, because it may add synchronization behavior and describe data dependencies not required by the original program. Interestingly, the graphs that *can* be trivially transformed into a planar *st*-graph by inserting dummy nodes can also be directly represented in the efficient graph representation introduced in Section 7.2 without using any dummy nodes.

5. It is easy to recast other undecidable problems as determinacy-checking problems. For example, consider a program that initializes a shared variable v to hold U, reads in an input string, s, and then spawns two tasks, t_1 and t_2, that attempt to recognize s using context-free grammars G_1 and G_2, respectively. If t_1 recognizes s using G_1, it writes Y into v otherwise it does not write to v. If t_2 does not recognize s, it writes N into v, otherwise it does not write to v. The program only avoids indeterminacy (and satisfies Bernstein's conditions) when at most one of the two tasks writes a value into v; if they

exist, however, that are not unduly restrictive, and can either be used as a discipline for programmers or imposed as a condition in language design.

6.3.1 Milner's Confluence Property

Milner (1989) points out that even if two systems are deterministic, allowing unconstrained communication between them immediately allows the possibility of nondeterminacy. Milner develops a *confluence* property sufficient to ensure parallel determinacy,[6] but Milner's parallelism follows a CSP (Hoare, 1985) model, which has its greatest applicability in distributed systems (it can represent shared memory, but the mapping is not ideal). My interest in this dissertation is shared-memory parallel computations, so we will now turn our attention to specific attempts to express shared-memory determinacy conditions.

6.3.2 I-Structures

One method for ensuring that shared-memory is accessed deterministically is to instigate a write-once policy and suspend any task that attempts to read a yet-to-be-written memory location until that location is written. This approach is taken by Arvind et al. (1989) in their *I-structure* data structure. This condition is simple and easy to enforce, yet fits a number of parallel algorithms. One problem with this approach is that, although indeterminate results are not possible with I-structures, read/write races remain possible and can result in deadlock.

6.3.3 Bernstein's Conditions

Bernstein (1966) developed conditions for *noninterference* between parallel tasks. *Bernstein's conditions* are sufficient conditions to prevent indeterminacy in a pair of parallel tasks. If t and t' are tasks that may run in parallel; W_t and $W_{t'}$ are the set of memory

both write to v, the final value of v may be either Y or N (i.e., indeterminacy arises when $s \in L(G_1)$ and $s \notin L(G_2)$). Thus, the program is deterministic for all inputs iff $L(G_1) \subseteq L(G_2)$. Determining whether $L(G_1) \subseteq L(G_2)$ for arbitrary G_1 and G_2 is a well-known undecidable problem.

6. Hoare (1985) also suggests conditions for determinacy for communicating parallel tasks, but his conditions are less interesting and less useful than Milner's.

locations written by t and t', respectively; and R_t and $R_{t'}$ are similarly defined to be the set of memory locations read by t and t', respectively; then t and t' are noninterfering iff $(W_t \cap W_{t'} = \varnothing) \wedge (R_t \cap W_{t'} = \varnothing) \wedge (W_t \cap R_{t'} = \varnothing)$.

Bernstein's conditions generalize to a set of tasks executing in parallel, $T = \{t_1, \ldots, t_n\}$. If we use $t_i \nleftrightarrow t_j$ to mean that there is no interference between t_i and t_j, then there is no interference between the tasks in T if $\forall t_i, t_j \in T : t_i \nleftrightarrow t_j \ \vee \ t_i \trianglelefteq t_j \ \vee \ t_j \trianglelefteq t_i$.

Like Milner's restrictions for confluence, Bernstein's noninterference conditions ensure determinacy, but are a sufficient rather than a necessary condition. It is possible for tasks to violate Bernstein's conditions but nevertheless be deterministic—for example, if two tasks write the same value to memory, the action is classed as interference even though it has no negative ramifications for determinacy.

6.3.4 Steele's Conditions

Steele (1990) proposes a more generous scheme that subsumes Bernstein's noninterference condition.[7] In Steele's terminology, two tasks, t and t', are *causally related* iff $t \trianglelefteq t' \ \vee \ t' \trianglelefteq t$. Steele's condition for avoiding indeterminacy is that any two accesses must either be causally related or *commute*.[8] Intuitively, two operations commute if the order in which they are performed does not matter; thus, multiple reads to a memory location commute with each other, but reads and writes to the same location do not. Similarly, multiple writes to the same location do not commute. Some useful operations that change memory do commute with one another—for example, an atomic increment commutes with an atomic decrement, but not with a read.

Interestingly, most algorithms intended to enforce Bernstein's conditions can be generalized in a straightforward manner to enforce Steele's conditions instead, because reads generalize to operations that commute, and writes generalize to operations that do not commute. We will simplify our discussion by focusing on the problem of reads and writes, but all the checking mechanisms we discuss are suited to this generalization.

7. Steele does not explicitly cite Bernstein's paper, perhaps because he was not aware of it—the conditions themselves are fairly straightforward and can be derived from first principles.

8. Steele's conditions are stated differently; I have put them in the same framework as our discussion of Bernstein's conditions to show the parallels between them.

Algorithm	Parallel Checking	Space Required	Time for Fork/Join	Time for Data Access
English-Hebrew Labeling	Yes	$O(vb + \min(ns, nbv))$	$O(n)$	$O(nb)$
Task Recycling	Yes	$O(vb + pb)$	$O(b)$	$O(b)$
Offset-Span Labeling	Yes	$O(e + v + \min(ns, nv))$	$O(n)$	$O(n)$
SP-Bags	No	$O(e + v)$	$O(\alpha(e + v, e + v))$	$O(\alpha(e + v, e + v))$
LR-Tags	Yes	$O(e + v)$	$O(1)$	$O(1)$

where
b = breadth of the task DAG
e = maximum number of effective tasks during execution
n = maximum depth of nested parallelism
p = maximum number of running tasks during execution
s = size of the task DAG (number of tasks)
v = number of shared locations being monitored
$\alpha \equiv$ Tarjan's (1975) functional inverse of Ackermann's function

Note: The time complexities given are for execution on a single processor; multiprocessor implementations may incur additional loss of parallelism due to synchronization. Also, the time complexity of the SP-Bags method is an amortized bound. The space complexities include the need to represent effective tasks, any of which may be the subject of thread operations (usually $e \ll v$, so e is frequently omitted when space complexities are described in the literature).

Table 6.1: Properties of runtime determinacy checkers.

6.3.5 Multiple Conditions

Restrictions for deterministic parallelism in parallel systems using communication channels and restrictions for deterministic access to shared memory do not necessarily conflict with each other. Brinch Hansen's *SuperPascal* (1994a; 1994b) has both of the above features, although the determinacy-enforcement facilities of SuperPascal are not as sophisticated as Milner's confluent parallel systems or Steele's safe asynchronous parallelism.

6.4 Enforcing Determinacy Conditions

Traditionally, parallel languages and runtime systems have done little to ensure determinacy. Programmers have instead had to rely on their intuitions about correct program behavior and depend on established (and frequently low-level) constructs for serializing

access to their data structures, such as locks. As we learned earlier, these basic mutual-exclusion mechanisms are not a complete answer—checks for determinacy (based on Bernstein's or Steele's conditions for shared-memory accesses, or Milner's confluence property for channel-based communications) have their place too.

Static checks at the language level (Taylor; Callahan & Subhlok, 1989; Emrath & Padua, 1988; Balasundaram & Kennedy, 1989; Beguelin, 1990; Beguelin & Nutt, 1994; Xu & Hwang, 1992; Bagheri, 1994; Brinch Hansen, 1994a), can ensure deterministic execution, but these checks rule out those algorithms that cannot (easily) be statically shown to be deterministic. For example, parallel tasks working on a shared array are likely to decide at runtime which array elements they should access, making static analysis of whether these accesses will lead to nondeterminacy difficult or impossible. Brinch Hansen encounters just this problem in his SuperPascal language, which attempts to enforce determinacy statically. SuperPascal includes an annotation that allows programmers to turn off static determinacy checking when necessary. Brinch Hansen notes that he has used this construct exclusively for dealing with tasks accessing a shared array.

A runtime test for determinacy races allows a wider range of programs to be executed with assured determinacy than similar compile-time tests. Obviously, compile-time tests should be used whenever practical, because they may both provide earlier detection of errors and reduce the amount of runtime testing required, but when compile-time tests cannot pass an algorithm as deterministic, runtime checking can at least assure us that the program is deterministic for a given input. Like other runtime tests, such as null-pointer checks and array-bounds checking, we may either use runtime determinacy checks as a mechanism for catching coding errors during development, turning checking off in production code for better performance; or leave checking on permanently, trading some performance for safety.

For some programs, runtime determinacy checking on a few inputs is enough to confirm that the program will always operate correctly. For example, the memory access patterns and parallel task structure of code that performs a parallel matrix multiply may depend only on the size of the matrices, not the actual contents of each matrix. Thus, checking that the code generates no determinacy errors for two matrices of a particular size also assures us that no multiplications using arrays of that size will ever generate a determinacy error.

There are two classes of dynamic determinacy checks: *on-the-fly checking* and *post-mortem analysis*. In on-the-fly checking (Nudler & Rudolph, 1986; Emrath & Padua, 1988; Schonberg, 1989; Steele, 1990; Dinning & Schonberg, 1990, 1991; Min & Choi, 1991; Mellor-Crummey, 1991; Feng & Leiserson, 1997), accesses to data are checked as they happen, and errors are signaled quickly. In post-mortem analysis (Miller & Choi, 1989), a log file is created during execution that is checked after the run to discover whether any accesses were invalid. Both methods have problems, however: On-the-fly detection can slow program execution significantly and may not accurately pinpoint the source of indeterminacy,[9] and post-mortem analysis faces the difficulty that logs may be voluminous (and wasteful, because the log becomes an unreliable indicator after its first error) and errors may not be detected in a timely way.

I will focus on using Bernstein's conditions as the basis of an on-the-fly dynamic determinacy checker. As we discussed in Section 6.3.4, it would be a simple matter to use my determinacy checker to enforce Steele's conditions as well.

6.4.1 Enforcing Bernstein's Conditions

Bernstein's conditions provide us with a basis for runtime checking, but although they are algebraically elegant, they do not, by themselves, provide an efficient determinacy checking algorithm. Implementations of runtime determinacy checking concentrate on tagging each shared memory cell with enough information for it to be checked, independent of other memory locations.

Previous algorithms for runtime determinacy checking have run into one of two problems: non–constant-factor overheads or a serial-execution requirement. *English-Hebrew Labeling* (Nudler & Rudolph, 1986), *Task Recycling* (Dinning & Schonberg, 1990) and *Offset-Span Labeling* (Mellor-Crummey, 1991) suffer from the first problem, whereas the *SP-Bags algorithm* (Feng & Leiserson, 1997) suffers from the second problem. Table 6.1 compares the properties of various determinacy-checking algorithms, including my own: the *LR-Tags method*.

9. If two accesses are made to a location, one correct and one erroneous, it may be that the error is not detected at the point the erroneous access is made, but is instead detected when the entirely correct access is made. The extent of detection and error reporting depends largely on the technique; typically we will know what task last accessed the datum, but not when, where, or why it performed that access. Choi & Min (1991) propose a software debugging assistant that can be helpful for those methods where this poor reporting would be a problem.

Feng and Leiserson's SP-Bags algorithm, embodied in their *Nondeterminator* determinacy checker for the CILK language (1997), provides the most time- and space-efficient determinacy-race detector to date for programs using nested parallelism. Their method is inspired by Tarjan's (1979) nearly linear-time least-common-ancestors algorithm.

The SP-Bags algorithm requires $O(T\,\alpha(e + v, e + v))$ time to run a program that runs in T time on one processor with checking turned off and uses v shared-memory locations, where α is Tarjan's (1975) functional inverse of Ackermann's function. For any practical situation, α has a constant upper bound of 4, so we may regard Feng and Leiserson's algorithm as an "as good as constant" amortized-time algorithm. The SP-Bags algorithm is also space efficient, needing $O(v + e)$ space to execute, where e is the maximum number of effective tasks in the CILK program. Execution of the program under CILK without determinacy checking also requires $\Omega(v + e)$ space; thus the algorithm has constant-factor space overheads.

Feng and Leiserson's method is a serial method, however. It can find the internal indeterminacies that would arise if the program were executed in parallel, but does so by running the program serially. Although this restriction may not be the crushing disadvantage it first appears to be (as developers often develop and debug their code on uniprocessor systems before running it on a more expensive parallel machine), it is nevertheless an unfortunate limitation. The SP-Bags method is also restricted to programs that use nested parallelism and so cannot enforce Bernstein's conditions for programs using producer–consumer parallelism or other, more esoteric, forms of parallelism.

Chapter 7

The LR-tags Method for Runtime Determinacy Checking

In the preceding chapter, we examined previous attempts at enforcing Bernstein's conditions for determinacy, and saw that techniques that enforce Bernstein's conditions may be simply extended to enforce Steele's slightly broader conditions. In this chapter, I will introduce my method for determinacy checking: the LR-tags method.

We will begin by examining a naïve approach to determinacy checking, and then refine that method to create an efficient algorithm.

7.1 A Simple Determinacy Checker

This section introduces a simple determinacy checker that is neither time nor space efficient. This checker will act as a stepping stone to the final LR-tags determinacy checker. Sections 7.2 and 7.3 will explain how the inefficiencies of this simple determinacy checker can be eliminated.

To enforce internal determinacy at runtime, we need to check two conditions. We must ensure that

1. Each read is valid, given previous writes
2. Each write is valid, given previous reads and writes

We need to consider prior reads when checking writes because reads might be scheduled in a different order on other runs—even when tasks are scheduled in a simple serial fashion, error detection should not be influenced by the order of task execution (see Listing 6.1(a)).

We saw in the previous section that Bernstein's noninterference conditions provide us with a simple rule to ensure deterministic execution; we will now express those conditions in a slightly different way. We will associate each datum d with both the last writer for that datum, $w(d)$, and the set of tasks that have accessed that datum since it was written, $R(d)$. We may think of $R(d)$ as the "reader set" for d, but $R(d)$ also includes the task that last wrote d. (Sometimes, when d is obvious from context or irrelevant to the discussion, I will write R and w rather than $R(d)$ and $w(d)$. I will also apply this convention to $r_f(d)$ and $r_b(d)$, which we define later.)

- *Reads* — A read is valid if the task that last modified the data item is an ancestor of (or is itself) the task performing the read. Expressing this restriction algebraically, a task t may read a datum d if

$$w(d) \trianglelefteq t \tag{Bern-1}$$

  where $w(d)$ is the task that wrote the value in d. When a read is valid, we update $R(d)$ as follows:

$$R(d) := R(d) \cup \{t\}$$

- *Writes* — Writes are similar to reads, in that the task that last modified the data item must be an ancestor of the task performing the write, but writes also require that all reads done since the last write are also ancestors of the task performing the write. Thus, a task t may write a datum d if

$$\forall r \in R(d) : r \trianglelefteq t \tag{Bern-2}$$

  where $R(d)$ is the set of tasks that have accessed d since it was last written (including the task that performed that write). If the write is valid, we update $R(d)$ and $w(d)$ as follows:

$$\begin{aligned} w(d) &:= t \\ R(d) &:= \{t\} \end{aligned}$$

Even this simple method provokes some interesting implementation questions, such as "How do we provide $R(d)$ and $w(d)$ for a datum d?". One way to store $R(d)$ and $w(d)$ is to add a writer field and a readers field to each determinacy-checked object. Feng and Leiserson (1997) provide an alternate way to address this question with their *shadow-spaces* instrumentation technique. In their approach, determinacy-checking information is stored in additional memory (shadow space) that mirrors the memory where the determinacy-checked objects are stored. Calculating where $R(d)$ and $w(d)$ are stored in this scheme is a simple function of the memory address of d. The shadow-spaces approach has the advantage that the way program data is laid out in memory need not change when determinacy checking is being used. Unfortunately, the shadow-spaces approach can waste memory because some objects take up more space than others (e.g., extended-precision complex numbers may take twenty-four times more space than eight-bit integers), leading to unused gaps in the shadow spaces.

But considering implementation questions is perhaps a little premature, as this naïve determinacy checker has poor asymptotic time and space requirements. For example, $R(d)$ can contain an arbitrary number of tasks, potentially causing tests involving $R(d)$ to be slow. There are also performance issues for ancestor queries. Traditional representations such as adjacency lists or adjacency matrices do not offer good computational complexity, especially given that the full definition of the $\trianglelefteq$ relation may not be known at the time the program begins—the parallel structure of the program may be determined by its input data. A linked data structure that mirrors the structure of the task DAG yields similarly poor performance.

Many parallel algorithms can enjoy better determinacy-checking performance, however, because they do not need the full power of an arbitrary DAG to describe how their tasks relate. In the next section, I will introduce restrictions to the task DAG that allow it to be represented efficiently.

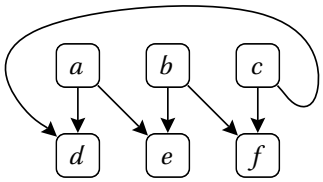


Figure 7.1: A graph for which the $\wr$ relation cannot be defined.

7.2 Defining a Graph with Two Relations

As we have seen, task DAGs form a reduced graph that can be adequately represented by a partial order ($\trianglelefteq$). As we saw in Chapter 2, partial orders are not always the most convenient representation—sometimes it is helpful to provide some additional structure if doing so yields an efficient algorithm. In this section, we will discuss how adding structure and restricting the class of graphs that we can represent enables us to use a compact representation for tasks and sets of tasks; this representation facilitates fast ancestor queries, which may either compare one task with another or determine whether all members of a set of tasks are ancestors of a particular task.

Definition 7.1 *An* LR graph *is a reduced* DAG *where there is a* (*possibly zero-length*) *path from x to y in the* DAG *iff $x \trianglelefteq y$, and those nodes that are not related by $\trianglelefteq$ are related by a second relation, $\wr$* (*pronounced "is to the left of"*)*, where*

- *$\wr$ is transitive*
- *$\wr$ does not relate nodes related by $\trianglelefteq$*
- *Any two nodes are related by either $\trianglelefteq$ or $\wr$*

Intuitively, we can imagine $\wr$ as the order in which a serial scheduler would execute tasks that could be executed concurrently under a parallel scheduler.

For some graphs, there is no way to define the $\wr$ relation because the two requirements for $\wr$ conflict. For example, in Figure 7.1, if $d \wr e$ and $e \wr c$, then $d \wr c$, which contradicts $c \triangleleft d$; other attempts to define $\wr$ for this graph will lead to similar failures.

The class of graphs for which we *can* define $\wr$ is a large one, however. All the graphs shown in Figure 6.1 are LR-graphs, in fact these graphs are drawn such that the left-to-right ordering of nodes on the page can be used as the basis for defining the $\wr$ relation

(provided that we remember to define $\wr$ only for nodes unrelated by $\trianglelefteq$). LR-graphs include a class of graphs known as *planar* st-*graphs*, which subsumes both series–parallel graphs and producer–consumer graphs.

A planar *st*-graph is a planar DAG with one *source node* and one *sink* (or *terminal*) *node*, both of which lie on the external face of the graph. A source node, conventionally labeled *s*, is a node with no inbound edges; a sink node, conventionally labeled *t*, is a node with no outbound edges. Di Battista et al. (1999, page 96) shows how $\wr$ may be defined for a planar *st*-graph using the dual of the graph, and proves that any pair of distinct nodes in the graph are either related by $\wr$ or related by $\lhd$.

Although the fundamental concepts underlying $\wr$ (as well as $\leftthreetimes$ and $\rightthreetimes$, which we will indroduce shortly) are not new (Birkhoff, 1967; Kelly & Rival, 1975; Kameda, 1975), LR-graphs avoid the planarity requirement present in earlier treatments of this subject. These earlier works show that it is always possible to define $\wr$ (or $\leftthreetimes$ and $\rightthreetimes$) for planar lattices (or planar *st*-graphs). Such a proof is tautological for LR-graphs because LR-graphs are characterized as reduced graphs for which $\wr$ can be defined. Thus, whereas all planar lattices are LR-graphs (Birkhoff, 1967, page 32, exercise 7(c)), there are potentially other (nonplanar) graphs that are also LR-graphs (e.g., Figure 6.1(c)). Nevertheless, LR-graphs have similar expressive power to planar *st*-graphs and planar lattices.[1]

A more straightforward definition of $\wr$ (for those graphs where $\wr$ can be defined) is: $x \wr y$ iff x comes before y in a left-to-right traversal of the DAG and $x \not\trianglelefteq y$ and $y \not\trianglelefteq x$. (Because we only use the traversal order to compare cousins, it is irrelevant whether the traversal is preorder or postorder, depth-first or breadth-first).

Task DAGs are *reduced* graphs because they contain no transitive edges, thus if we restrict the task DAG to be an *st*-graph, it becomes a reduced planar *st*-graph. Reduced planar *st*-graphs have the useful property that the $\wr$ relation completely defines the planar embedding of the graph. The prohibition of transitive edges means that none of the parents or children of a node can be related by $\lhd$. Thus, the clockwise order of inbound edges to a node is identical to the order of their sources (the parent nodes)

1. It is fairly straightforward to transform a planar *st*-graph into an LR-graph or vice versa by adding dummy nodes to the graph. When transforming a planar *st*-graph to an LR-graph, dummy nodes may need to be added because an LR-graph is a reduced graph (see Di Battista et al., 1999, page 127, for an algorithm), whereas nonplanar LR-graphs (or LR-graphs without a single source or sink node) may requre the addition of dummy nodes to form a planar *st*-graph (see Di Battista et al., 1999, page 20, for a review common planarization algorithms).

under $\wr$ and the anticlockwise order of outbound edges from a node is identical to the order of their destinations (the child nodes) under $\wr$.

Formally, we can express the properties of $\lhd$ and $\wr$ as follows:

1. The usual rules for equality apply (nodes are only equal to themselves).

2. For any x and y, exactly *one* of the following is true:
$$x \lhd y, \qquad y \lhd x, \qquad x \wr y, \qquad y \wr x, \qquad x = y.$$

3. Both $\lhd$ and $\wr$ are transitive.

Like $\lhd$, the reflexive closure of $\wr$ forms a partial order. Given the above definition, we can observe that

$$(x \wr y) \wedge (y \lhd z) \Rightarrow (x \wr z) \vee (x \lhd z) \tag{7.1}$$
$$(x \wr y) \wedge (z \lhd y) \Rightarrow (x \wr z) \vee (z \lhd x) \tag{7.2}$$
$$(y \wr x) \wedge (y \lhd z) \Rightarrow (z \wr x) \vee (x \lhd z) \tag{7.3}$$
$$(y \wr x) \wedge (z \lhd y) \Rightarrow (z \wr x) \vee (z \lhd x) \tag{7.4}$$

because any other relationships between x and z lead to a contradiction with the definitions of $\wr$ and $\lhd$.

Now we will define the two relations that form the core of the LR-tags method. Let us define $\swarrow$ (pronounced "is to the left or below") and $\searrow$ (pronounced "is to the left or above") as follows,

$$x \swarrow y \equiv (x \wr y) \vee (y \lhd x) \vee (x = y) \tag{7.5}$$
$$x \searrow y \equiv (x \wr y) \vee (x \lhd y) \vee (x = y) \tag{7.6}$$

It is a simple matter to prove that $\swarrow$ and $\searrow$ are transitive relations by a trivial examination of cases. It is similarly easy to show that $\swarrow$ and $\searrow$ provide total orders, which, for a graph G, we term $G_{\swarrow}$ and $G_{\searrow}$.

If these orderings seem vaguely familiar, they should. $G_{\swarrow}$ and $G_{\searrow}$ are, in fact, topological sortings of the nodes, and represent the order in which nodes are encountered under a left-to-right postorder traversal and a reversed right-to-left postorder traversal,

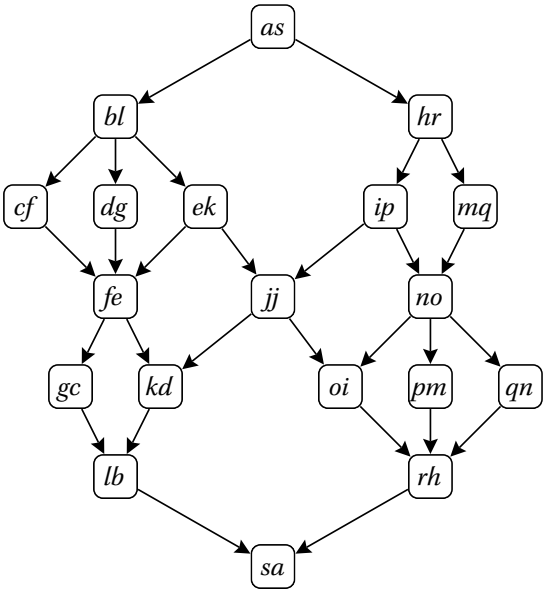


Figure 7.2: An *st*-graph. The tasks are labeled to reveal the left-to-right postorder traversal (the first letter of the label) and the reversed right-to-left postorder traversal (the second letter of the label).

respectively. The numerical sequence of nodes in $G_{\swarrow}$ and $G_{\searrow}$ also corresponds to the x and y coordinates of those nodes in a dominance drawing of the graph, as shown in Figure 7.3. The discussion that follows does not rely on either of these properties, although the results described in this section are well known for dominance drawings (Di Battista et al., 1999) and have their foundations in the properties of planar lattices (Kelly & Rival, 1975).

Some readers may wonder whether a reversed right-to-left postorder traversal is the same as a left-to-right preorder traversal. It is not. Although these traversals are equivalent in the case of trees, they are not equivalent in more general graphs. Consider the example given in Figure 7.2. In this case, $G_{\swarrow} = [sa, lb, gc, kd, fe, cf, dg, rh, oi, jj, ek, bl, pm, qn, no, ip, mq, hr, as]$ and $G_{\searrow} = [as, bl, cf, dg, ek, fe, gc, hr, ip, jj, kd, lb, mq, no, oi, pm, qn, rh, sa]$. In contrast, the left-to-right preorder traversal of this graph is $[as, bl, cf, fe, gc, lb, sa, kd, dg, ek, jj, oi, rh, hr, ip, no, pm, qn, mq]$. Interestingly, the reversed right-to-left postorder traversal is equivalent to the order in which tasks would be executed by a left-to-right depth-first serial scheduler. We will return to this property in Section 10.1.

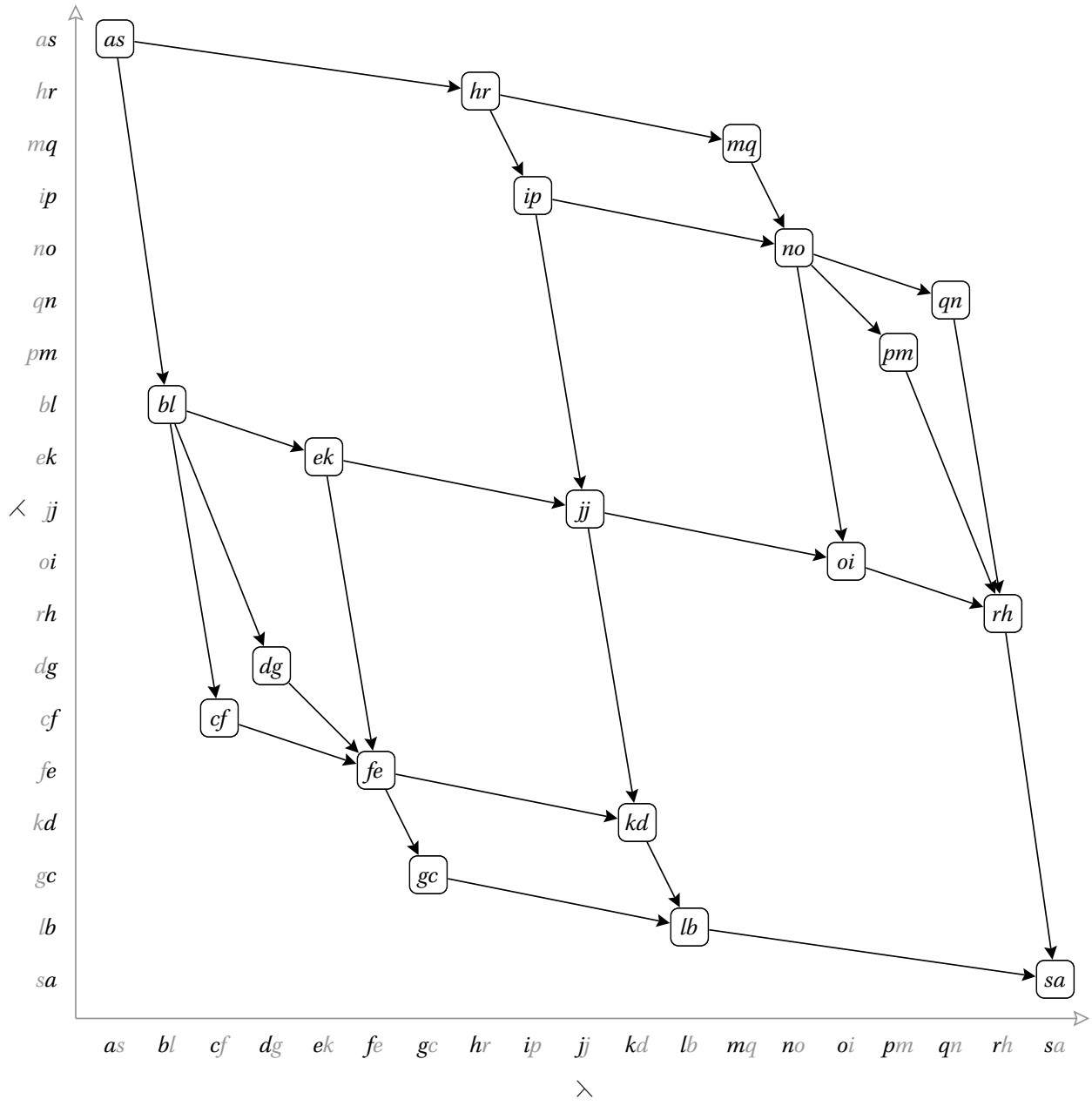


Figure 7.3: A dominance drawing of the graph from Figure 7.2. (More compact dominance drawings of this graph are possible; see Section 10.2).

From these definitions, we can either derive the $\prec$ and $\succ$ total orders from $\trianglelefteq$ and $\wr$, or do the reverse and derive $\trianglelefteq$ and $\wr$ from $\prec$ and $\succ$. Since $\prec$ and $\succ$ are total orders, and $\trianglelefteq$ and $\wr$ are not, it is preferable to represent a task DAG G entirely using $G_{\prec}$ and $G_{\succ}$. We will return to the topic of maintaining these total orders for a dynamic graph in the next section.

Given $\prec$ and $\succ$, we can easily perform ancestor queries, since

$$(y \prec x) \wedge (x \succ y) \Leftrightarrow x \trianglelefteq y \tag{7.7}$$

(from equations 7.5 and 7.6). In Figure 7.2, we can see that *bl* is an ancestor of *oi* because $bl \succ oi$ and $oi \prec bl$.

But there is another, more significant, benefit we can reap from using these traversals. Recall that the naïve determinacy checker we discussed in Section 7.1 needed to maintain a set R of all the tasks that had read a particular datum, and check to make sure that the current task t was a descendant of all those tasks. In a graph represented by $\prec$ and $\succ$, we can avoid maintaining the set R—instead we can use two values from R, the *frontmost*, r_f, and the *backmost* r_b as surrogates for the entire set. We define r_f and r_b as the least and greatest elements of R under $\prec$ and $\succ$, respectively; thus,

$$(r_f \in R) \wedge (r_b \in R) \tag{7.8}$$

$$(\forall r \in R : r_f \prec r) \wedge (\forall r \in R : r \succ r_b). \tag{7.9}$$

In Figure 7.2, if $R = \{cf, ek, ip\}$, $r_f = cf$ and $r_b = ip$. Using comparisons against only these two nodes, we can confirm that *kd* is a descendant of the nodes in R, whereas *gc* and *jj* are not.

Theorem 7.2

$$(\forall r \in R : r \trianglelefteq t) \Leftrightarrow (t \prec r_f) \wedge (r_b \succ t) \tag{7.10}$$

Proof

The forward implication is trivial to prove:

$$\begin{aligned}
&\forall r \in R : r \trianglelefteq t \\
&\quad \Rightarrow (r_f \trianglelefteq t) \wedge (r_b \trianglelefteq t) && \text{[from 7.8]} \\
&\quad \Rightarrow \big((t \prec r_f) \wedge (r_f \succ t)\big) \wedge \big((t \prec r_b) \wedge (r_b \succ t)\big) && \text{[from 7.7]} \\
&\quad \Rightarrow (t \prec r_f) \wedge (r_b \succ t).
\end{aligned}$$

Proving in the other direction is almost as easy:

$$\begin{aligned}
&(t \prec r_f) \wedge (r_b \succ t) \\
&\quad \Rightarrow \forall r \in R, \big((t \prec r_f) \wedge (r_f \prec r)\big) \wedge \forall r \in R, \big((r \succ r_b) \wedge (r_b \succ t)\big) && \text{[from 7.9]} \\
&\quad \Rightarrow \forall r \in R, (t \prec r) \wedge \forall r \in R, (r \succ t) && \text{[transitivity]} \\
&\quad \Rightarrow \forall r \in R, r \trianglelefteq t. && \text{[from 7.7]}
\end{aligned}$$

□

Given these properties, the following rules are sufficient for checking that reads and writes performed by a task t on a datum d are deterministic:

- *Reads* — A read is valid if $(t \prec w(d)) \wedge (w(d) \succ t)$—that is, if it satisfies condition Bern-1. If the read is valid, $r_f(d)$ and $r_b(d)$ may need to be updated:
  - If $t \prec r_f(d)$ then $r_f(d) := t$
  - If $r_b(d) \succ t$ then $r_b(d) := t$
- *Writes* — A write is valid if $(t \prec r_f(d)) \wedge (r_b(d) \succ t)$—that is, it satisfies condition Bern-2. If the write is valid, $w(d)$, $r_f(d)$, and $r_b(d)$ are updated to be t.

The next section shows how it is possible to provide an efficient implementation of the $\prec$ and $\succ$ relations for a dynamically created task DAG, and thereby provide an efficient determinacy checker.

7.3 Maintaining $G_{\prec}$ and $G_{\succ}$ for a Dynamic Graph

In the previous section, we saw that if we had efficient implementations of the $\prec$ and $\succ$ relations, we could implement determinacy checking efficiently. For a static graph, it may be trivial to perform two postorder traversals to generate the orderings $G_{\prec}$ and $G_{\succ}$, but parallel programs are usually dynamic and the final form of the task DAG is often unknown until the run has completed. Thus we require an algorithm that can maintain $G_{\prec}$ and $G_{\succ}$ for an *incomplete* graph and efficiently update these orderings as nodes are added to the bottom of the graph, eventually completing it.

We will assume that the graph is a reduced graph where $\wr$ can be easily defined (i.e., the left-to-right ordering of parents and children is known and remains consistent during the construction of the graph). These assumptions hold for both series–parallel and producer–consumer DAGs. For series–parallel DAGs, *any* left-to-right ordering of parents and children correctly defines $\wr$. For producer–consumer DAGs, it is sufficient to adopt a convention that producers fall to the left of consumers (as exemplified by Figure 6.1(b)).

When we list parents or children of a node, we will list them in their left-to-right order. This node order is identical to their sorted order under $\wr$, $\prec$, and $\succ$. Thus, when a node n has parents $P = \{p_1, \ldots, p_k\}$,

$$\forall j \in \{1, \ldots, k-1\}, (p_j \ \wr \ p_{j+1}) \tag{7.11}$$

$$\forall x, \forall j \in \{1, \ldots, k-1\}, \neg(p_j \ \wr \ x) \vee \neg(x \ \wr \ p_{j+1}) \tag{7.12}$$

$$\forall x, (x \lhd n) \Rightarrow \exists p \in P, (x = p) \vee (x \lhd p) \tag{7.13}$$

and similarly, when a parent node p has children $C = \{c_1, \ldots, c_l\}$,

$$\forall j \in \{1, \ldots, l-1\}, (c_j \ \wr \ c_{j+1}) \tag{7.14}$$

$$\forall x, \forall j \in \{1, \ldots, l-1\}, \neg(c_j \ \wr \ x) \vee \neg(x \ \wr \ c_{j+1}) \tag{7.15}$$

$$\forall x, (p \lhd x) \Rightarrow \exists c \in C, \big((c = x) \vee (c \lhd x)\big) \tag{7.16}$$

(the latter condition states that no children of p are omitted).

We will also restrict ourselves to considering graphs that grow through the addition

of new terminal nodes, and assume that a node and all its inbound edges are added together. Both of these assumptions reflect the underlying purpose of the task graph—it would be nonsensical to start a task and then, after it had begun, add an additional commencement restriction.

Given these restrictions, our problem becomes one of finding the correct place to insert a node in both $G_{\prec}$ and $G_{\succ}$, because adding a terminal node to the DAG cannot affect the ordering of other nodes with respect to each other (i.e., if $x \succ y$ before the node is added, $x \succ y$ afterwards). I present a simple constant-time algorithm to perform these insertions below.

7.3.1 Insertion Strategy

For each node n in the task DAG we provide an associated *shadow node* n' where $n \lhd n'$. This node does not represent a task—its only purpose is to allow us to find insertion points in $G_{\prec}$ and $G_{\succ}$ when adding children to n. We will discuss shadow nodes in detail in Chapter 8 and prove that the insertion strategy explained below is correct. For this discussion, I will give a shorter and less formal explanation of the properties of shadow nodes.

Let us consider the case of adding a new node n with parents $P = \{p_1, \ldots, p_k\}$. After adding n, we desire that

$$\forall p \in \{p_1, \ldots, p_k\} : \big((n \prec p) \wedge (p \succ n)\big). \tag{7.17}$$

The position of the new node in $G_{\prec}$ is determined by p_1, because $\forall p \in P : (p_1 \prec p)$, and thus $(n \prec p_1) \Rightarrow \forall p \in P : (n \prec p)$. Let $c_1, \ldots, c_l$ be the children of p_1, where $c_i = n$ (i.e., n is to be the ith child, c_i, of p_1). For correct insertion, we desire that $\forall c \in \{c_1, \ldots, c_{i-1}\} : (c \prec n)$ and $\forall c \in \{c_{i+1}, \ldots, c_l\} : (n \prec c)$. To achieve this condition, we insert n' into $G_{\prec}$ immediately to the right of m, where $m = c_{i-1}$ if $(i \neq 1) \wedge (p'_1 \prec c_{i-1})$ and $m = p'_1$ otherwise. We then insert n into $G_{\prec}$ immediately to the right of n'.

We perform the converse operation for $G_{\succ}$. The position of the new node in $G_{\succ}$ is determined by p_k, because $\forall p \in P : (p \succ p_k)$, and therefore $(p_k \succ n) \Rightarrow \forall p \in P : (p \succ n)$. Let $c_1, \ldots, c_l$ be the children of p_k, where $c_i = n$ (i.e., n is to be the ith child, c_i, of p_k). For

correct insertion, we desire that $\forall c \in \{c_1, \ldots, c_{i-1}\} : (c \succ n)$ and $\forall c \in \{c_{i+1}, \ldots, c_l\} : (n \succ c)$. To achieve this condition, we insert n' into $G_{\succ}$ immediately to the left of b, where $b = c_{i+1}$ if $(i \neq l) \wedge (c_{i+1} \succ p'_n)$ and $b = p'_n$ otherwise. We then insert n into $G_{\succ}$ immediately to the left of n'.

The above scheme is general in that it allows children to be added one by one. In practice, some growth patterns allow straightforward derivatives of this algorithm that do not require shadow nodes. These optimizations can be applied to both series–parallel and producer–consumer DAGs, but since they do not affect the time or space complexity of the LR-tags method, we will defer that discussion until Chapter 10.

7.3.2 Efficient Ordered Lists

In the previous section we saw how we can dynamically maintain two total orders, $G_{\prec}$ and $G_{\succ}$, representing a task DAG, but we have not yet shown that these total orders can be maintained and queried efficiently. Thankfully, this problem is the same *list-order problem* we saw in Section 2.1. As we observed there, this problem has efficient solutions (Dietz & Sleator, 1987; Tsakalidis, 1984) that can perform insertions, deletions, and order queries in constant time and require space proportional to the number of items in the list.

As in Chapter 2, I prefer the first of Dietz and Sleator's two ordered-list algorithms. Dietz and Sleator's second, more complex, algorithm has better theoretical performance, because it can perform all operations in constant worst-case time, in contrast to their simpler algorithm, which can only perform ordered-list deletions and order queries in constant worst-case time, requiring constant amortized time for ordered-list insertions. In practice, however, Dietz and Sleator's first algorithm is less complex, much faster (in total execution time), easier to implement, and better suited to parallelization than its completely real-time counterpart.

Parallel Access to Ordered Lists

For determinacy checking to run in parallel, it is necessary to allow parallel access to the ordered-list structure. Sometimes, however, these accesses must be serialized to ensure that the ordered list is not inadvertently corrupted.

To describe the locking strategy required, we need to expose a little of the underlying details of the Dietz and Sleator's ordered-list data structure (complete details are provided in Appendix D).

Each item in the ordered list is tagged with two integers: an *upper tag*, which provides a coarse-grained ordering for tasks; and a *lower tag*, which orders nodes that have the same upper tag. When an item is inserted into the list, it is given the same upper tag as its predecessor and a suitable new lower tag. Sometimes the ordered list "runs out" of lower tags, at which point all the ordered-list entries with that upper tag are renumbered and given new upper and lower tags. Let us call this process a *local reorganization*. Sometimes the algorithm also runs out of room in the upper-tag space, and so must renumber the upper tags so that they are more evenly distributed. Let us call this process a *global reorganization*. Both kinds of reorganization can only be triggered by ordered-list insertions and are fairly rare in practice (e.g., for the benchmarks in Chapter 11, the average is about one reorganization per thousand ordered-list insertions).

The most common operation for the ordered lists in the determinacy checker will be comparisons between items in the list. Other list operations, such as insertions and deletions are less common. To reduce the chances of the ordered list becoming a bottleneck, we desire a locking strategy that maximizes the amount of parallelism while nonetheless remaining correct.

In my parallelization of the ordered-list structure (described in more detail in Appendix E), order queries are lock free, although they may have to be retried if the ordered list undergoes a local or global reorganization while the order query is being performed.

Ordered-list insertions are performed while holding a shared lock, allowing multiple insertions to take place concurrently. This lock is upgraded to a local or global exclusive lock if a local or global reorganization is required. Deletions from the ordered list are performed while holding an exclusive lock.

I do not claim that this parallelization scheme is the most efficient one possible, but it does appear to work well in practice, especially when the optimizations detailed in Section 10.5.3 are implemented.

Chapter 8

The Shadowed Task Graph

In Section 7.3 of the preceding chapter, I outlined an algorithm to maintain $G_{\prec}$ and $G_{\succ}$ for a dynamic graph. In this chapter, we will examine the foundations of that algorithm in more detail. These details will be of most interest to readers who wish to assure themselves that the algorithm presented is correct—this chapter can be skipped or skimmed through in a first reading of this dissertation, or ignored entirely by readers interested in more practical matters.

8.1 Introduction

As we saw in Chapter 7, an LR-graph can be defined by two total orders (defining relations $\prec$ and $\succ$). Although any two total orders can define a LR-graph, an important question is whether we can quickly and correctly insert a new node into each total order to add a new node to the graph at a particular position.

Extending the graph is not difficult if efficiency is not important. It is relatively straightforward to construct an algorithm that will add a node to a graph of size n in $O(n)$ time.[1] In Chapter 7, we examined an algorithm to add new leaves to an LR-Graph efficiently by using *shadow nodes* to determine insertion points in $\prec$ and $\succ$.

In this chapter, I will show how it is possible to construct a **shadowed LR-Graph**,

1. One possible algorithm involves creating $\wr$ and $\trianglelefteq$ from $\prec$ and $\succ$, extending $\wr$ and $\trianglelefteq$ appropriately, and then creating updated versions of $\prec$ and $\succ$ from $\wr$ and $\trianglelefteq$.

G^*, from a given LR-graph, G; discuss the properties of the shadow nodes it contains; and show how these properties lead to the insertion algorithm of Chapter 7.

8.2 Notation

In our discussion, we will refer to an arbitrary graph, G, and to G^*, the graph resulting from augmenting G by adding shadow nodes. G^* subsumes G, containing all G's nodes and relationships, but adds nodes such that, for each node x in G, there is an additional node x' in G^*. Hence, if G is of size n, G^* will be of size $2n$. Similarly, G' is defined as the graph of shadow nodes in G^*—that is, $G' = G - G^*$. Notationally, we will use $a, b, c, \ldots$ for variables that can only be non-shadow nodes, $a', b', c', \ldots$ for shadow nodes, and $a^*, b^*, c^*, \ldots$ for nodes that may be either.

Let us also define a few notational conveniences, $\not\prec$, $\not\succ$, $\not\wr$, and $\not\lhd$, where

$$x^* \not\prec y^* \equiv \neg(y^* \prec x^*) \equiv (x^* \wr y^*) \vee (y^* \lhd x^*) \tag{8.1a}$$

$$x^* \not\succ y^* \equiv \neg(y^* \succ x^*) \equiv (x^* \wr y^*) \vee (x^* \lhd y^*) \tag{8.1b}$$

$$x^* \not\lhd y^* \equiv \neg(x^* \lhd y^*) \tag{8.1c}$$

$$x^* \not\wr y^* \equiv \neg(x^* \wr y^*). \tag{8.1d}$$

8.3 Below-Left-Of Sets

This section defines **below-left-of sets** and **below-right-of sets**, which we will use to formally define shadow nodes in the next section, and proves some useful properties of these sets.

For an arbitrary LR-graph G let us define the following:

Definition 8.1 *For any $x \in G$,*

$$\downarrow\mathrm{L}_G(x) \equiv \{l \in G \mid \exists s \in G : (s \wr x) \wedge (s \trianglelefteq l)\} \tag{8.2a}$$

$$\downarrow\mathrm{R}_G(x) \equiv \{r \in G \mid \exists s \in G : (x \wr s) \wedge (s \trianglelefteq r)\} \tag{8.2b}$$

In English, $\downarrow\mathrm{L}_G(x)$ (pronounced "the below-left-of set of x") contains all the nodes in G

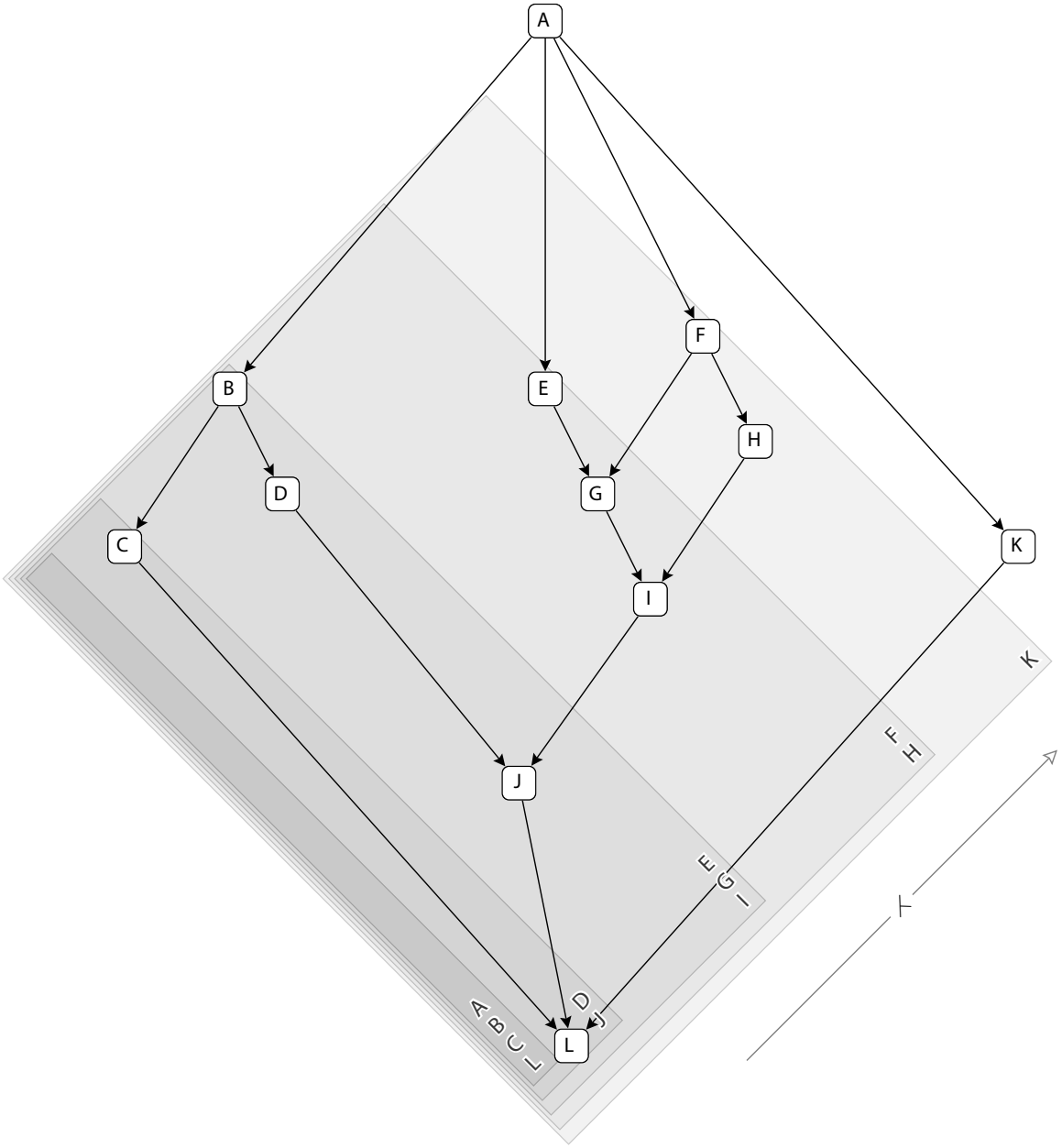


Figure 8.1: An illustration of below-left-of sets. Each shaded area is the below-left-of set for the nodes listed in its corner.

to the left of x, and all the descendants of the nodes to the left of x. Similarly, ${}^{\downarrow}\mathrm{R}_G(x)$ contains all the nodes to the right of x, and all the descendants of the nodes to the right of x.

Figure 8.1 provides an example of below-left-of sets. We can see, for example, that ${}^{\downarrow}\mathrm{L}_G(\mathsf{H}) = {}^{\downarrow}\mathrm{L}_G(\mathsf{F}) = \{\mathsf{L}, \mathsf{C}, \mathsf{J}, \mathsf{D}, \mathsf{B}, \mathsf{I}, \mathsf{G}, \mathsf{E}\}$. Notice that I is included in the set not because it is a descendant of F, but because it is a descendant of E, and $\mathsf{E} \wr \mathsf{F}$. (The graph is drawn as a dominance drawing, making it easy to see that $G_{\searrow} = [\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}, \mathsf{J}, \mathsf{K}, \mathsf{L}]$ and $G_{\swarrow} = [\mathsf{L}, \mathsf{C}, \mathsf{J}, \mathsf{D}, \mathsf{B}, \mathsf{I}, \mathsf{G}, \mathsf{E}, \mathsf{H}, \mathsf{F}, \mathsf{K}, \mathsf{A}]$.)

From definitions 8.2a and 8.2b, we can trivially observe that a node is never contained within its own below-left-of set; that is,

$$\forall x \in G : x \notin {}^{\downarrow}\mathrm{L}_G(x) \tag{8.3a}$$

$$\forall x \in G : x \notin {}^{\downarrow}\mathrm{R}_G(x), \tag{8.3b}$$

and that all nodes in the below-left-of set of some node x are strictly to-the-left-of or below x; that is,

$$\forall l \in {}^{\downarrow}\mathrm{L}_G(x) : (l \not\swarrow x) \tag{8.4a}$$

$$\forall r \in {}^{\downarrow}\mathrm{R}_G(x) : (x \not\swarrow r). \tag{8.4b}$$

Figure 8.1 also reveals another property of below-left-of sets that we will express formally in the following lemma:

Lemma 8.2 *For any $x \in G$, ${}^{\downarrow}\mathrm{L}_G(x)$ forms a (possibly null) prefix of $G_{\swarrow}$, and, similarly, ${}^{\downarrow}\mathrm{R}_G(x)$ forms a (possibly null) suffix of $G_{\searrow}$. That is,*

$$\forall l \in {}^{\downarrow}\mathrm{L}_G(x) : m \swarrow l \Rightarrow m \in {}^{\downarrow}\mathrm{L}_G(x) \tag{8.5a}$$

$$\forall r \in {}^{\downarrow}\mathrm{R}_G(x) : r \searrow n \Rightarrow n \in {}^{\downarrow}\mathrm{R}_G(x) \tag{8.5b}$$

Proof

We will prove equation 8.5a. The proof for equation 8.5b is analogous.

$$(l \in {\downarrow}\mathrm{L}_G(x)) \wedge (m \prec l)$$

expand using definition 8.2a

$$\Rightarrow \big(\exists s : (s \wr x) \wedge (s \trianglelefteq l)\big) \wedge (m \prec l)$$

rearrange, widen (using definition of $\prec$)

$$\Rightarrow \exists s : (s \wr x) \wedge (m \prec l) \wedge (l \prec s)$$

eliminate l (transitivity)

$$\Rightarrow \exists s : (s \wr x) \wedge (m \prec s)$$

expand (definition of $\prec$)

$$\Rightarrow \exists s : (s \wr x) \wedge \big((s \trianglelefteq m) \vee (m \wr s)\big)$$

distribute

$$\Rightarrow \big(\exists s : (s \wr x) \wedge (m \wr s)\big) \vee \big(\exists s : (s \wr x) \wedge (s \trianglelefteq m)\big)$$

eliminate quantification in first term (transitivity)

$$\Rightarrow (m \wr x) \vee \big(\exists s : (s \wr x) \wedge (s \trianglelefteq m)\big)$$

reintroduce quantification

$$\Rightarrow \big(\exists s : (s \wr x) \wedge (s = m)\big) \vee \big(\exists s : (s \wr x) \wedge (s \trianglelefteq m)\big)$$

simplify (remove duplication)

$$\Rightarrow \exists s : (s \wr x) \wedge (s \trianglelefteq m)$$

simplify using definition 8.2a

$$\Rightarrow m \in {\downarrow}\mathrm{L}_G(x)$$

□

We will now show some obvious corollaries of the above lemma.

Corollary 8.3

$$\forall x, y \in G : x \in {\downarrow}\mathrm{L}_G(y) \;\Rightarrow\; {\downarrow}\mathrm{L}_G(x) \subset {\downarrow}\mathrm{L}_G(y) \tag{8.6a}$$

$$\forall x, y \in G : x \in {\downarrow}\mathrm{R}_G(y) \;\Rightarrow\; {\downarrow}\mathrm{R}_G(x) \subset {\downarrow}\mathrm{R}_G(y) \tag{8.6b}$$

Proof

We will prove equation 8.6a; equation 8.6b is analogous. First, we will prove that ${\downarrow}\mathrm{L}_G(x) \subseteq {\downarrow}\mathrm{L}_G(y)$ by showing that $l \in {\downarrow}\mathrm{L}_G(x) \Rightarrow l \in {\downarrow}\mathrm{L}_G(y)$:

$$\begin{aligned}
&(x \in {\downarrow}\mathrm{L}_G(y)) \wedge (l \in {\downarrow}\mathrm{L}_G(x)) \\
&\quad \text{apply equation 8.4a} \\
&\quad \Rightarrow (x \in {\downarrow}\mathrm{L}_G(y)) \wedge (l \not\asymp x) \\
&\quad \text{apply equation 8.5a} \\
&\quad \Rightarrow l \in {\downarrow}\mathrm{L}_G(y)
\end{aligned}$$

Now, all that remains is to show that ${\downarrow}\mathrm{L}_G(x) \neq {\downarrow}\mathrm{L}_G(y)$. This task is trivial, because $(x \in {\downarrow}\mathrm{L}_G(y))$ and $x \notin {\downarrow}\mathrm{L}_G(x)$, thus x is one element that is in ${\downarrow}\mathrm{L}_G(y)$ but not ${\downarrow}\mathrm{L}_G(x)$. □

Corollary 8.4

$$\forall x, y \in G : (x \wr y) \;\Rightarrow\; {\downarrow}\mathrm{L}_G(x) \subset {\downarrow}\mathrm{L}_G(y) \tag{8.7a}$$

$$\forall x, y \in G : (y \wr x) \;\Rightarrow\; {\downarrow}\mathrm{R}_G(x) \subset {\downarrow}\mathrm{R}_G(y) \tag{8.7b}$$

Proof

We will prove equation 8.7a; equation 8.7b is analogous.

$$\begin{aligned}
&x \wr y \\
&\quad \text{apply equation 8.2a} \\
&\quad \Rightarrow x \in {\downarrow}\mathrm{L}_G(y) \\
&\quad \text{apply equation 8.6a} \\
&\quad \Rightarrow {\downarrow}\mathrm{L}_G(x) \subset {\downarrow}\mathrm{L}_G(y)
\end{aligned}$$

□

Given this result, we can expect that distinct nodes that share the same below-left-of set must be related by $\lhd$, because they cannot be related by $\wr$. Figure 8.1 provides several instances of this property—for example, $\downarrow \mathrm{L}_G(\mathsf{E}) = \downarrow \mathrm{L}_G(\mathsf{G}) = \downarrow \mathrm{L}_G(\mathsf{I})$, and $\mathsf{E} \lhd \mathsf{G} \lhd \mathsf{I}$. We will express this property formally in the following corollary:

Corollary 8.5 *It is possible for two distinct nodes in G to have the same below-left-of sets, but only if one is an ancestor of the other.*

$$\forall x, y \in G : (\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(y)) \;\Rightarrow\; \big((x = y) \vee (x \lhd y) \vee (y \lhd x)\big) \tag{8.8a}$$

$$\forall x, y \in G : (\downarrow \mathrm{R}_G(x) = \downarrow \mathrm{R}_G(y)) \;\Rightarrow\; \big((x = y) \vee (x \lhd y) \vee (y \lhd x)\big) \tag{8.8b}$$

Proof

It is trivial to rule out the other possibilities, $x \wr y$ and $y \wr x$, because they contradict Corollary 8.4 (e.g., $(x \wr y) \Rightarrow (\downarrow \mathrm{L}_G(x) \subset \downarrow \mathrm{L}_G(y))$). □

If you examine Figure 8.1, you may also notice that $\downarrow \mathrm{L}_G(\mathsf{G}) \subset \downarrow \mathrm{L}_G(\mathsf{F})$ and $\mathsf{F} \lhd \mathsf{G}$. Interestingly, these two properties imply that $\mathsf{G} \in \downarrow \mathrm{L}_G(\mathsf{F})$—the following lemma states this property in the general case:

Lemma 8.6

$$\forall x, y \in G : (\downarrow \mathrm{L}_G(x) \subset \downarrow \mathrm{L}_G(y)) \wedge (y \lhd x) \;\Rightarrow\; x \in \downarrow \mathrm{L}_G(y) \tag{8.9a}$$

$$\forall x, y \in G : (\downarrow \mathrm{R}_G(x) \subset \downarrow \mathrm{R}_G(y)) \wedge (y \lhd x) \;\Rightarrow\; x \in \downarrow \mathrm{R}_G(y) \tag{8.9b}$$

(*Proof begins on next page...*)

Proof

We will prove equation 8.9a; equation 8.9b is analogous.

$(\downarrow \mathrm{L}_G(x) \subset \downarrow \mathrm{L}_G(y)) \wedge (y \lhd x)$

expand using properties of subsets

$\Rightarrow (\exists l \in \downarrow \mathrm{L}_G(y) : l \notin \downarrow \mathrm{L}_G(x)) \wedge (y \lhd x)$

expand using equation 8.2a

$\Rightarrow \big(\exists l, s \in G : (s \wr y) \wedge (s \trianglelefteq l) \wedge (l \notin \downarrow \mathrm{L}_G(x))\big) \wedge (y \lhd x)$

rearrange (and duplicate one term)

$\Rightarrow \exists l, s \in G : (s \wr y) \wedge \big((s \wr y) \wedge (y \lhd x)\big) \wedge (s \trianglelefteq l) \wedge (l \notin \downarrow \mathrm{L}_G(x))$

eliminate y from one term

$\Rightarrow \exists l, s \in G : (s \wr y) \wedge \big((s \wr x) \vee (s \lhd x)\big) \wedge (s \trianglelefteq l) \wedge (l \notin \downarrow \mathrm{L}_G(x))$

rearrange and discard

$\Rightarrow \big(\exists l, s \in G : (s \wr x) \wedge (s \trianglelefteq l) \wedge (l \notin \downarrow \mathrm{L}_G(x))\big) \vee \big(\exists s \in G : (s \wr y) \wedge (s \lhd x)\big)$

simplify using equation 8.2a

$\Rightarrow \big(\exists l \in G : (l \in \downarrow \mathrm{L}_G(x)) \wedge (l \notin \downarrow \mathrm{L}_G(x))\big) \vee (x \in \downarrow \mathrm{L}_G(y))$

only the right-hand term can be true

$\Rightarrow x \in \downarrow \mathrm{L}_G(y)$ □

8.4 Construction of a Shadowed Graph

We will define the shadow graph by defining the relationships of shadow nodes to non-shadow nodes under $\prec$ and $\succ$. In informal English, the shadow node x' lies after the last node in $\downarrow \mathrm{L}_G(x)$ and before the first non-shadow node that follows $\downarrow \mathrm{L}_G(x)$; in cases where several shadow nodes are adjacent (because they have the same below-left-of sets), we define the order of the shadow nodes using the ordering of their non-shadow counterparts (i.e., $x' \prec y'$ if $x \lhd y$).

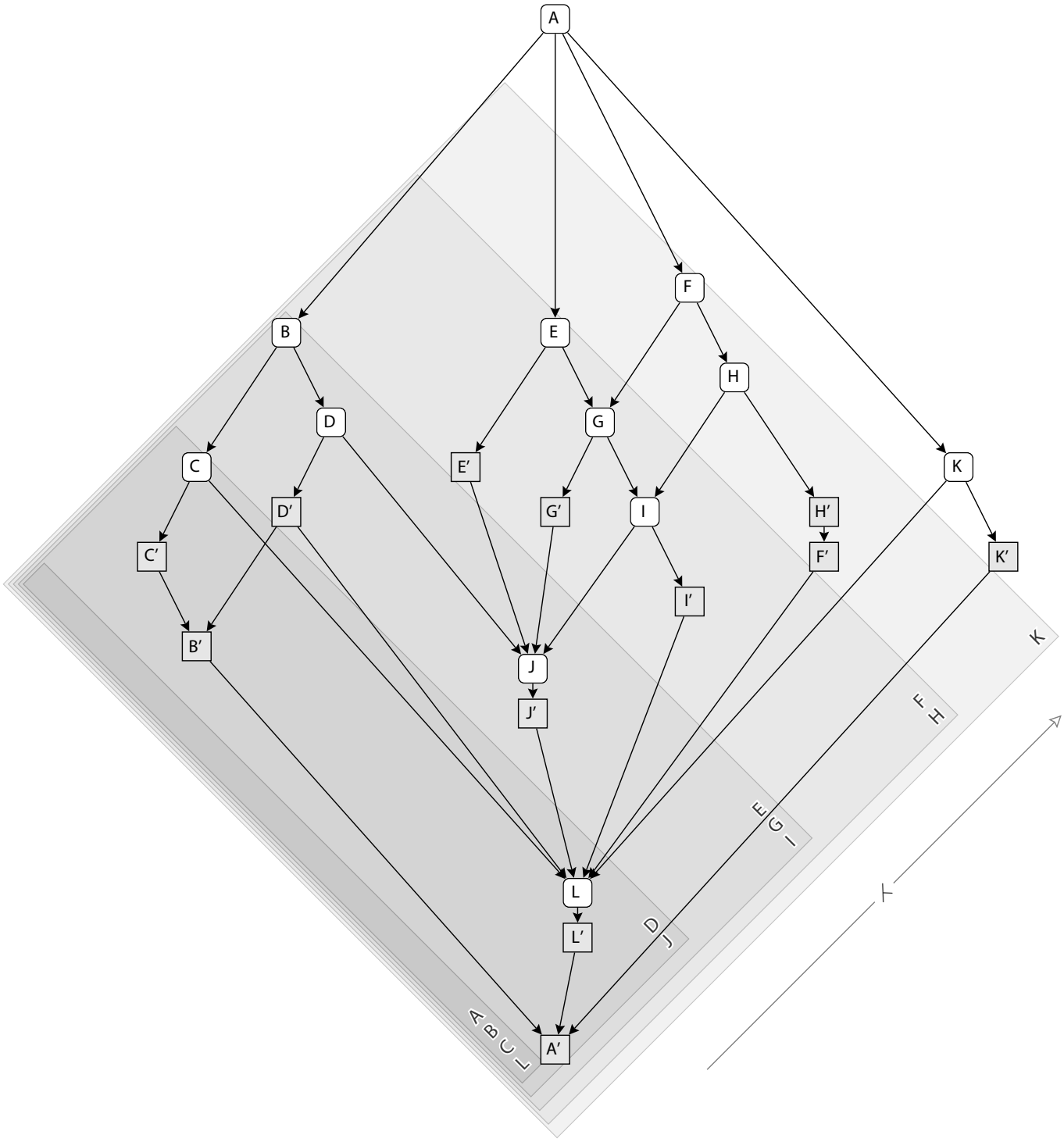


Figure 8.2: An illustration of below-left-of sets and shadow nodes.

Definition 8.7

$$\forall l, x \in G: \ \big(l \in {}^{\downarrow}\mathrm{L}_G(x) \Leftrightarrow (l \not\prec x')\big) \tag{8.10a}$$

$$\forall r, x \in G: \ \big(r \in {}^{\downarrow}\mathrm{R}_G(x) \Leftrightarrow (x' \not\succ r)\big) \tag{8.10b}$$

$$\forall x, y \in G: \ \Big(\big({}^{\downarrow}\mathrm{L}_G(x) = {}^{\downarrow}\mathrm{L}_G(y)\big) \ \Rightarrow \ \big((x' \prec y') \Leftrightarrow (x \trianglelefteq y)\big)\Big) \tag{8.11a}$$

$$\forall x, y \in G: \ \Big(\big({}^{\downarrow}\mathrm{R}_G(x) = {}^{\downarrow}\mathrm{R}_G(y)\big) \ \Rightarrow \ \big((y' \succ x') \Leftrightarrow (x \trianglelefteq y)\big)\Big) \tag{8.11b}$$

$$\forall x \in G, y' \in G' : x \neq y' \tag{8.12}$$

These equations show how to unambiguously create a shadowed graph from an existing graph. Figure 8.2 adds shadow nodes to the graph shown in Figure 8.1 following these rules. In the resulting graph, the total orders that define the graph are $G^*_{\succ} = [\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{C}', \mathsf{D}, \mathsf{D}', \mathsf{B}', \mathsf{E}, \mathsf{E}', \mathsf{F}, \mathsf{G}, \mathsf{G}', \mathsf{H}, \mathsf{I}, \mathsf{J}, \mathsf{J}', \mathsf{I}', \mathsf{H}', \mathsf{F}', \mathsf{K}, \mathsf{K}', \mathsf{A}']$, and $G^*_{\prec} = [\mathsf{A}', \mathsf{B}', \mathsf{C}', \mathsf{C}, \mathsf{D}', \mathsf{J}, \mathsf{J}', \mathsf{D}, \mathsf{B}, \mathsf{E}', \mathsf{G}', \mathsf{I}', \mathsf{I}, \mathsf{G}, \mathsf{E}, \mathsf{F}', \mathsf{H}', \mathsf{H}, \mathsf{F}, \mathsf{K}', \mathsf{K}, \mathsf{A}]$.

8.5 Properties of Shadow Nodes

Shadow nodes are useful because they express certain useful information about a graph in a "handy" way. We will cover some simple properties and then move on to more complex ones. Put simply, a node's shadow always lies beneath it, and the nodes in the graph between x and x' can only be reached by passing through x. We will prove these two properties in the following lemmas:

(*Lemma begins on next page...*)

Lemma 8.8 *A node's shadow is always beneath it.*

$$\forall x \in G : x \lhd x' \tag{8.13}$$

Proof

First, let us observe that $x' \not\prec\!\!\!\!\!\!\diagup\; x$, because its negation, $x \prec x'$, leads to a contradiction:

$$(x \in G) \wedge (x \prec x')$$

apply equation 8.12

$$\Rightarrow (x \in G) \wedge (x \not\prec\!\!\!\!\!\!\diagup\; x')$$

apply equation 8.10a

$$\Rightarrow x \in {}^{\downarrow}\mathrm{L}_G(x))$$

contradiction with equation 8.3a

By a symmetric argument, $x \not\succ\!\!\!\!\!\!\diagdown\; x'$. Combining these relationships, we have

$$(x' \not\prec\!\!\!\!\!\!\diagup\; x) \wedge (x \not\succ\!\!\!\!\!\!\diagdown\; x')$$

simplify

$$\Rightarrow x \lhd x'$$ □

Figure 8.2 also shows the close correspondence between shadow nodes and below-left-of (and below-right-of) sets. Thus, the following lemma should be very apparent:

(*Lemma begins on next page...*)

Lemma 8.9

$$\downarrow \mathrm{L}_G(x) \subset \downarrow \mathrm{L}_G(y) \;\Rightarrow\; x' \not\swarrow y' \tag{8.14a}$$

$$\downarrow \mathrm{R}_G(x) \subset \downarrow \mathrm{R}_G(y) \;\Rightarrow\; y' \not\swarrow x' \tag{8.14b}$$

Proof

As usual, we will prove equation 8.14a; equation 8.14b is analogous.

$$\downarrow \mathrm{L}_G(x) \subset \downarrow \mathrm{L}_G(y)$$

rules for proper subsets

$$\Rightarrow \exists l \in \downarrow \mathrm{L}_G(y) : l \notin \downarrow \mathrm{L}_G(x)$$

expand using definition 8.10a

$$\Rightarrow \exists l \in G : (l \not\swarrow y') \wedge \neg(l \not\swarrow x')$$

remove negation

$$\Rightarrow \exists l \in G : (l \not\swarrow y') \wedge (x' \prec l)$$

eliminate l (transitivity)

$$\Rightarrow x' \not\swarrow y'$$ □

Given the way that shadows lurk beneath their regular counterparts, we should not be surprised that there is a strong connection between the $\wr$ relationships for regular nodes and the $\wr$ relationships for shadows. We will examine this connection in the following two lemmas:

(*Lemma begins on next page...*)

Lemma 8.10

$$\forall x, y \in G : (x \ \wr \ y) \ \Rightarrow \ (x \ \wr \ y') \tag{8.15a}$$

$$\forall x, y \in G : (x \ \wr \ y) \ \Rightarrow \ (x' \ \wr \ y) \tag{8.15b}$$

Proof

As usual, we will prove equation 8.15a; equation 8.15b is analogous.

$$x \ \wr \ y$$

apply equation 8.2a

$$\Rightarrow x \in {\downarrow}\mathrm{L}_G(y)$$

apply equation 8.10a

$$\Rightarrow x \not\asymp y'$$

expand

$$\Rightarrow (x \ \wr \ y') \vee (y' \lhd x) \tag{8.16}$$

On the other hand, from equation 8.13 we know that $y \lhd y'$. From that, and $x \ \wr \ y$, we can determine

$$(x \ \wr \ y) \wedge (y \lhd y')$$

eliminate y

$$\Rightarrow (x \ \wr \ y') \vee (x \lhd y') \tag{8.17}$$

If we combine equations 8.16 and 8.17, the only valid solution is $x \ \wr \ y'$. □

(*Lemma begins on next page...*)

Lemma 8.11

$$\forall x' \in G', y \in G : (x' \wr y) \Rightarrow (x' \wr y') \tag{8.18a}$$

$$\forall x \in G, y' \in G' : (x \wr y') \Rightarrow (x' \wr y') \tag{8.18b}$$

Proof

As usual, we will prove equation 8.18a; equation 8.18b is analogous. First, let us observe that

$$x' \wr y$$

apply equation 8.13

$$\Rightarrow (x' \wr y) \wedge (y \lhd y')$$

eliminate y

$$\Rightarrow (x' \wr y') \vee (x' \lhd y') \tag{8.19}$$

We will show that of these two cases, only the first case, $x' \wr y'$, can be true. The second case, $x' \lhd y'$, leads to a contradiction:

$$x' \lhd y'$$

widen

$$\Rightarrow y' \not\asymp x'$$

use negated form

$$\Rightarrow \neg(x' \not\not\asymp y')$$

apply equation 8.14a (backwards)

$$\Rightarrow \neg(\downarrow \mathrm{L}_G(x) \subset \downarrow \mathrm{L}_G(y))$$

apply equation 8.7a (backwards)

$$\Rightarrow \neg(x \wr y)$$

contradicts equation 8.15b

Thus, $x' \lhd y'$ leads to a contradiction and, hence, from equation 8.19, $x' \wr y'$. □

We can combine the preceding two lemmas in two useful ways:

Corollary 8.12

$$\forall x^* \in G^*, y \in G : (x^* \wr y) \Rightarrow (x^* \wr y') \tag{8.20a}$$
$$\forall x \in G, y^* \in G^* : (x \wr y^*) \Rightarrow (x' \wr y^*) \tag{8.20b}$$

Proof

Trivially from Lemmas 8.10 and 8.11. □

Corollary 8.13

$$\forall x, y \in G : (x \wr y) \Rightarrow (x' \wr y') \tag{8.21}$$

Proof

Trivially from the Lemmas 8.10 and 8.11. □

8.5.1 Adjacency Properties

In this section, we will show that leaves in G have the property that they lie adjacent to their shadows in the $\prec$ and $\succ$ orders defining G^*.

Lemma 8.14 *If c is a leaf in G, there is no shadow node that lies between c and c' under $\prec$ or $\succ$.*

$$(\nexists d \in G : c \lhd d) \Rightarrow$$
$$\forall x' \in G' : (x' \not\prec c) \Rightarrow (x' \prec c') \tag{8.22a}$$
$$\forall x' \in G' : (c \not\succ x') \Rightarrow (c' \succ x') \tag{8.22b}$$

Proof

As usual, we will prove equation 8.22a; equation 8.22b is analogous. We will begin by

observing that

$$x' \not\prec c$$

weaken

$$\Rightarrow c \not\prec x'$$

apply equation 8.15a backwards

$$\Rightarrow c \not\prec x$$

and

$$(c \not\prec x) \wedge (\nexists d \in G : c \lhd d)$$

weaken

$$\Rightarrow (c \not\prec x) \wedge (c \not\lhd x)$$

restate in the positive

$$\Rightarrow (x = c) \vee (x \wr c) \vee (x \lhd c)$$

We will consider each possibility, $(x = c)$, $(x \wr c)$, and $(x \lhd c)$, in turn. First,

$$(x = c) \Rightarrow (x' = c') \Rightarrow (x' \prec c') \tag{8.23}$$

Similarly,

$$(x \wr c)$$

apply equation 8.21

$$\Rightarrow (x' \wr c')$$

weaken

$$\Rightarrow (x' \prec c') \tag{8.24}$$

Finally, we will consider the last case, $x \lhd c$. First, we will show that if $x \lhd c$ with $x' \not\prec c$, then $\downarrow \mathrm{L}_G(x) \subseteq \downarrow \mathrm{L}_G(c)$. We will do this by showing that $\forall y \in G : (y \wr x) \Rightarrow (y \wr c)$.

$(x \lhd c) \wedge (y \ \wr \ x) \wedge (x' \not\asymp c)$

apply equation 8.10a (backwards) to last term

$\Rightarrow (x \lhd c) \wedge (y \ \wr \ x) \wedge (c \notin {\downarrow}\mathrm{L}_G(x))$

expand last term using definition 8.2a

$\Rightarrow (x \lhd c) \wedge (y \ \wr \ x) \wedge \big(\nexists s \in G : (s \ \wr \ x) \wedge (s \trianglelefteq c)\big)$

rearrange (duplicate one term)

$\Rightarrow (y \ \wr \ x) \wedge \big((x \lhd c) \wedge (y \ \wr \ x)\big) \wedge \big(\nexists s \in G : (s \ \wr \ x) \wedge (s \trianglelefteq c)\big)$

eliminate x in second term

$\Rightarrow (y \ \wr \ x) \wedge \big((y \lhd c) \vee (y \ \wr \ c)\big) \wedge \big(\nexists s \in G : (s \ \wr \ x) \wedge (s \trianglelefteq c)\big)$

simplify

$\Rightarrow (y \ \wr \ c) \vee \big(\big((y \ \wr \ x) \wedge (y \lhd c)\big) \wedge \big(\nexists s \in G : (s \ \wr \ x) \wedge (s \trianglelefteq c)\big)\big)$

simplify (remove contradictory second term)

$\Rightarrow y \ \wr \ c$

and, finally,

$({\downarrow}\mathrm{L}_G(x) \subseteq {\downarrow}\mathrm{L}_G(c)) \wedge (x \lhd c)$

expand

$\Rightarrow \big(({\downarrow}\mathrm{L}_G(x) = {\downarrow}\mathrm{L}_G(c)) \vee ({\downarrow}\mathrm{L}_G(x) \subset {\downarrow}\mathrm{L}_G(c))\big) \wedge (x \lhd c)$

rearrange, discard

$\Rightarrow ({\downarrow}\mathrm{L}_G(x) \subset {\downarrow}\mathrm{L}_G(c)) \vee \big(({\downarrow}\mathrm{L}_G(x) = {\downarrow}\mathrm{L}_G(c)) \wedge (x \lhd c)\big)$

apply equation 8.14a (and weaken)

$\Rightarrow (x' \prec c') \vee \big(({\downarrow}\mathrm{L}_G(x) = {\downarrow}\mathrm{L}_G(c)) \wedge (x \lhd c)\big)$

apply equation 8.11a

$\Rightarrow (x' \prec c') \vee (x' \prec c')$

simplify

$\Rightarrow x' \prec c'$ (8.25)

Thus, equations 8.23, 8.24, and 8.25 and together show that $x' \prec c'$. □

Lemma 8.15 *If c is a leaf in G, then c and c' are adjacent in $\prec$ and $\succ$.* (*This lemma is a generalization of the preceding lemma, which only applied to shadow nodes.*)

$$(\nexists d \in G : c \lhd d) \Rightarrow$$
$$\forall x^* \in G^* : (x^* \not\prec c) \;\Rightarrow\; (x^* \prec c') \tag{8.26a}$$
$$\forall x^* \in G^* : (c \not\succ x^*) \;\Rightarrow\; (c' \succ x^*) \tag{8.26b}$$

Proof

We will show only equation 8.26a; equation 8.26b is analogous.

First, let us observe that x^* must be either a shadow or non-shadow node:

$$x^* \in G^* \;\Rightarrow\; (\exists x \in G : x = x^*) \vee (\exists x' \in G' : x' = x^*)$$

If x^* is a shadow node, then we can use the preceding lemma (equation 8.22a applies). Therefore, we only need to consider the case where $\exists x \in G : x = x^*$:

$$(x^* \not\prec c) \wedge (\exists x \in G : x = x^*) \wedge (\nexists d \in G : c \lhd d)$$

simplify (eliminate x^* using equality)

$$\Rightarrow (x \not\prec c) \wedge (\nexists d \in G : c \lhd d)$$

expand (definition of $\not\prec$)

$$\Rightarrow \big((x \;\wr\; c) \vee (c \lhd x)\big) \wedge (\nexists d \in G : c \lhd d)$$

simplify (remove impossible cases)

$$\Rightarrow x \;\wr\; c$$

apply equation 8.15a

$$\Rightarrow x \;\wr\; c'$$

widen

$$\Rightarrow x \prec c'$$

□

Thus, if c is a leaf, c and c' are adjacent in both total orders. We can restate this adjacency condition more tersely: A node and its shadow are adjacent in both orderings

if and only if $\forall z^* \in G^* : (c \lhd z^*) \Rightarrow (c' \trianglelefteq z^*)$, as we show in the following corollary:

Corollary 8.16

$$\begin{aligned}&\big(\forall x^* \in G^* : (x^* \not\prec\!\!\!\prec c) \Rightarrow (x^* \prec c')\big) \wedge \big(\forall y^* \in G^* : (c \not\succ\!\!\!\succ y^*) \Rightarrow (c' \succ y^*)\big) \\ &\quad \Leftrightarrow \ \big(\forall z^* \in G^* : (c \lhd z^*) \Rightarrow (c' \trianglelefteq z^*)\big)\end{aligned} \tag{8.27}$$

Proof

First, the forwards direction:

$$\big(\forall x^* \in G^* : (x^* \not\prec\!\!\!\prec c) \Rightarrow (x^* \prec c')\big) \wedge \big(\forall y^* \in G^* : (c \not\succ\!\!\!\succ y^*) \Rightarrow (c' \succ y^*)\big) \wedge (c \lhd z^*)$$

expand $\lhd$ as $\not\prec\!\!\!\prec$ and $\not\succ\!\!\!\succ$

$$\Rightarrow \big(\forall x^* \in G^* : (x^* \not\prec\!\!\!\prec c) \Rightarrow (x^* \prec c')\big) \wedge \big(\forall y^* \in G^* : (c \not\succ\!\!\!\succ y^*) \Rightarrow (c' \succ y^*)\big) \wedge \big((z^* \not\prec\!\!\!\prec c) \wedge (c \not\succ\!\!\!\succ z^*)\big)$$

simplify

$$\Rightarrow (z^* \prec c') \wedge (c' \succ z^*)$$

simplify

$$\Rightarrow c' \trianglelefteq z^*)$$

(*Proof continues on next page...*)

Second, the backwards direction. In this case, we will only show a proof for $\prec$ (the proof for $\succ$ is analogous).

$$\big(\forall z^* \in G^* : (c \lhd z^*) \Rightarrow (c' \trianglelefteq z^*)\big) \wedge (x^* \not\prec c)$$

expand $\not\prec$

$$\big(\forall z^* \in G^* : (c \lhd z^*) \Rightarrow (c' \trianglelefteq z^*)\big) \wedge \big((x^* \between c) \vee (c \lhd x^*)\big)$$

simplify

$$\Rightarrow \big((x^* \between c) \vee (c' \trianglelefteq x^*)\big)$$

apply equation 8.20a

$$\Rightarrow \big((x^* \between c') \vee (c' \trianglelefteq x^*)\big)$$

simplify

$$\Rightarrow (x^* \prec c') \qquad \square$$

8.6 The Inconsequentiality Condition

As we have seen, leaves in G have the property that they lie adjacent to their shadows in $\prec$ and $\succ$, and thus satisfy the condition

$$\forall x^* \in G^* : (c \lhd x^*) \Rightarrow (c' \trianglelefteq x^*) \tag{8.28}$$

I call this condition the **inconsequentiality condition** because, as we will see shortly, nodes that satisfy this condition can be added to (or removed from) the total orders that define the graph without damaging the relationships between any of the other nodes in the graph—in particular, the locations of shadow nodes remain the same whether or not such **inconsequential** nodes and their shadows are present.[2] (We will prove this property in the next section.)

For the purposes of determinacy checking, we need only be concerned with graphs that grow by adding new leaf nodes. The proofs that follow are slightly more general,

2. There may also be other nodes that can be safely removed from the graph without disrupting the placement of shadow nodes. The inconsequentiality condition is therefore a sufficient rather than a necessary condition.

however, being based on the addition and removal of inconsequential nodes, rather than the addition and removal of leaves.

In Figure 8.2, we can see that both J and L satisfy the inconsequentiality condition, while only L is a leaf (the only leaf in the graph). If we remove J from the total orders, they become $G^*_{\searrow} = [\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{C}', \mathsf{D}, \mathsf{D}', \mathsf{B}', \mathsf{E}, \mathsf{E}', \mathsf{F}, \mathsf{G}, \mathsf{G}', \mathsf{H}, \mathsf{I}, \mathsf{I}', \mathsf{H}', \mathsf{F}', \mathsf{K}, \mathsf{K}', \mathsf{A}']$, and $G^*_{\swarrow} = [\mathsf{A}', \mathsf{B}', \mathsf{C}', \mathsf{C}, \mathsf{D}', \mathsf{D}, \mathsf{B}, \mathsf{E}', \mathsf{G}', \mathsf{I}', \mathsf{I}, \mathsf{G}, \mathsf{E}, \mathsf{F}', \mathsf{H}', \mathsf{H}, \mathsf{F}, \mathsf{K}', \mathsf{K}, \mathsf{A}]$. Figure 8.3 shows the graph defined by these relations.

Let us consider what kinds of nodes, besides leaves, may satisfy the inconsequentiality condition. A non-leaf node c which satisfies the inconsequentiality condition is flanked by nodes to its left and right such that every node that is a child of c is also a child of these left and right nodes. We will express this condition more formally in the following lemma:

Lemma 8.17

$$\big(\forall x^* \in G^* : (c \lhd x^*) \Rightarrow (c' \trianglelefteq x^*)\big) \Rightarrow$$

$$\forall x \in G : (c \lhd x) \;\Rightarrow\; \big(\exists l \in G : (l \;\wr\; c) \wedge (l \lhd x)\big) \tag{8.29a}$$

$$\forall x \in G : (c \lhd x) \;\Rightarrow\; \big(\exists r \in G : (c \;\wr\; r) \wedge (r \lhd x)\big) \tag{8.29b}$$

(*Proof begins on page 127...*)

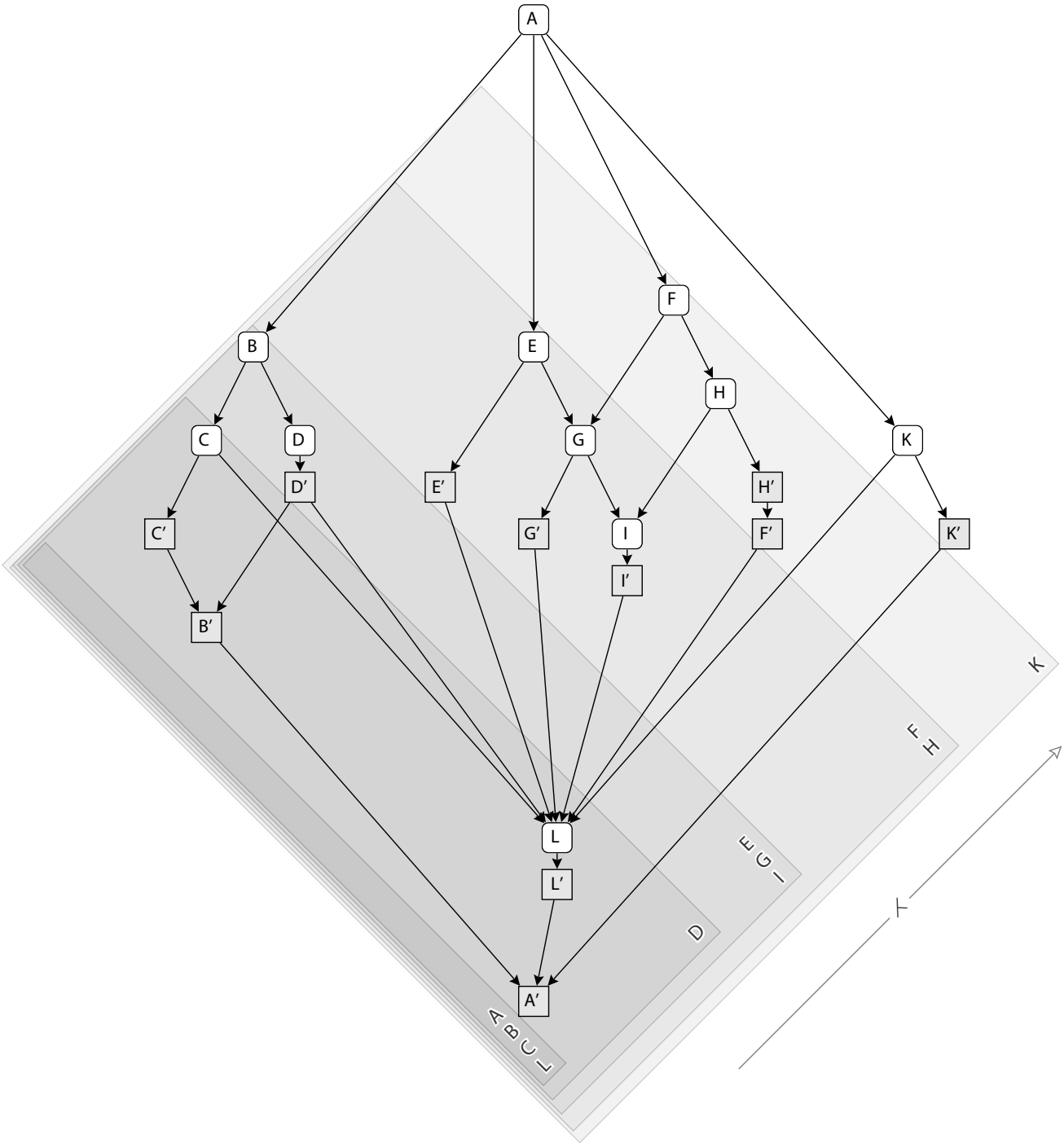


Figure 8.3: Deleting an inconsequential node. This graph is identical to the graph shown in Figure 8.2 except that the inconsequential node J has been removed.

Proof

As usual, we will prove equation 8.29a; equation 8.29b is analogous.

$$
\begin{aligned}
& c \lhd x \\
& \quad \text{add an obvious consequence} \\
& \quad \Rightarrow (c \lhd x) \wedge (x \not\wr c) \\
& \quad \text{apply inconsequentiality condition to first term} \\
& \quad \Rightarrow (c' \lhd x) \wedge (x \not\wr c) \\
& \quad \text{generalize first term} \\
& \quad \Rightarrow (x \not\precsim c') \wedge (x \not\wr c) \\
& \quad \text{apply equation 8.10a (Definition 8.7) to first term} \\
& \quad \Rightarrow (x \in {\downarrow}\mathrm{L}_G(c)) \wedge (x \not\wr c) \\
& \quad \text{apply equation 8.2a (Definition 8.1) to first term} \\
& \quad \Rightarrow \big(\exists l \in G : (l \wr c) \wedge (l \trianglelefteq x)\big) \wedge (x \not\wr c) \\
& \quad \text{simplify } (l \neq x \text{ because } x \not\wr c) \\
& \quad \Rightarrow \exists l \in G : (l \wr c) \wedge (l \lhd x)
\end{aligned}
$$

□

8.7 Incremental Graph Destruction

As a prelude to the next section, where we will discuss incremental graph construction, this section shows that it is possible to remove an inconsequential node and its shadow from the total orders that define a shadowed graph without making the location of the remaining shadow nodes incorrect (in stark contrast with the deletion of an arbitrary node from the graph, which may have consequences for the placement of shadow nodes—see Figures 8.4 and 8.5).

We will consider the case of removing a node c (and its shadow, c') from the total orders that represent G^*, resulting in a new graph Γ^*.

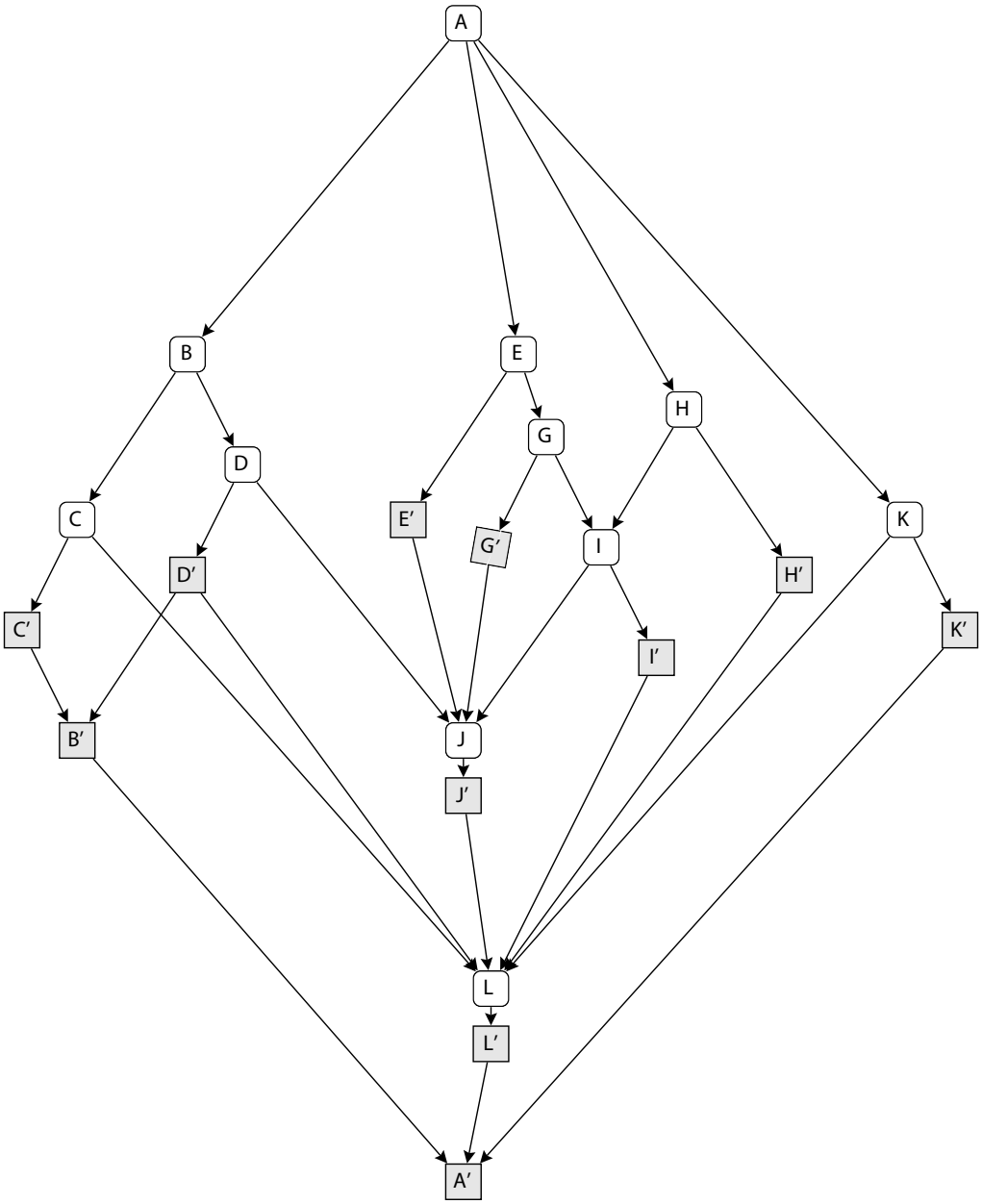


Figure 8.4: Incorrect deletion of a non-inconsequential node. This graph is identical to the graph shown in Figure 8.2 except that the F and F′ nodes have been removed from the total orders that define the graph. Unfortunately, this graph places the shadow node G′ incorrectly. The correct graph is shown in Figure 8.5

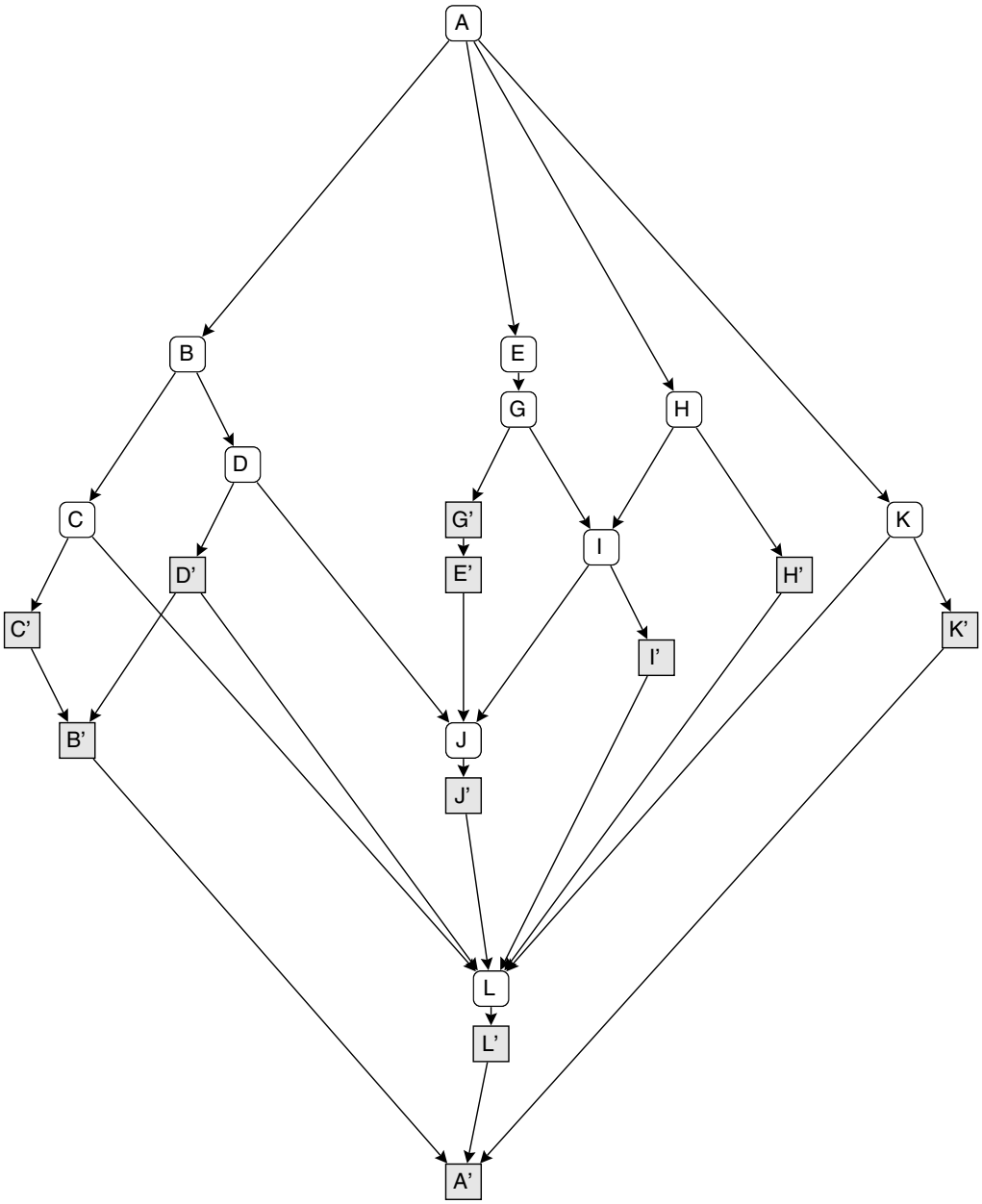


Figure 8.5: Correct deletion of a non-inconsequential node. This graph was created by removing F and all the shadow nodes from the graph shown in Figure 8.2 and then adding shadow nodes in the appropriate places. In other words, all the shadow nodes were thrown away and recreated from scratch.

Lemma 8.18 *With the exception of the removal of c, the below-left-of sets of G and Γ are identical. (And similarly for below-right-of sets).*

$$\forall x \in \Gamma : {}^{\downarrow}\mathrm{L}_{\Gamma}(x) = {}^{\downarrow}\mathrm{L}_{G}(x) - \{c\} \tag{8.30a}$$

$$\forall x \in \Gamma : {}^{\downarrow}\mathrm{R}_{\Gamma}(x) = {}^{\downarrow}\mathrm{R}_{G}(x) - \{c\} \tag{8.30b}$$

Proof

As usual, we will prove equation 8.30a; equation 8.30b is analogous. We will prove this equation 8.30a by first showing that all members of ${}^{\downarrow}\mathrm{L}_{\Gamma}(x)$ are in ${}^{\downarrow}\mathrm{L}_{G}(x) - \{c\}$ and then showing that all members of ${}^{\downarrow}\mathrm{L}_{G}(x) - \{c\}$ are in ${}^{\downarrow}\mathrm{L}_{\Gamma}(x)$. First, let us consider some $l \in {}^{\downarrow}\mathrm{L}_{\Gamma}(x)$:

$l \in {}^{\downarrow}\mathrm{L}_{\Gamma}(x)$

apply equation 8.2a (Definition 8.1)

$\Rightarrow (l \in \Gamma) \wedge \big(\exists s \in \Gamma : (s \curlywedge x) \wedge (s \trianglelefteq l)\big)$

G subsumes Γ, thus $(s \in \Gamma) \Rightarrow (s \in G)$

$\Rightarrow (l \in G) \wedge \big(\exists s \in G : (s \curlywedge x) \wedge (s \trianglelefteq l)\big)$

apply equation 8.2a (Definition 8.1)

$\Rightarrow l \in {}^{\downarrow}\mathrm{L}_{G}(x)$

(*Proof continues on next page...*)

Second, we shall consider some $l \in (\downarrow \mathrm{L}_G(x) - \{c\})$:

$l \in (\downarrow \mathrm{L}_G(x) - \{c\})$

apply equation 8.2a (Definition 8.1)

$\Rightarrow \big(l \in (G - \{c\})\big) \wedge \big(\exists s \in G : (s \wr x) \wedge (s \trianglelefteq l)\big)$

use definition of Γ ($\forall x \in G : (x \in \Gamma) \vee (x = c)$)

$\Rightarrow (l \in \Gamma) \wedge \Big(\big(\exists s \in \Gamma : (s \wr x) \wedge (s \trianglelefteq l)\big) \vee \big((c \wr x) \wedge (c \trianglelefteq l)\big)\Big)$

apply equation 8.2a (Definition 8.1)

$\Rightarrow (l \in \downarrow \mathrm{L}_\Gamma(x)) \vee \Big((l \in \Gamma) \wedge \big((c \wr x) \wedge (c \trianglelefteq l)\big)\Big)$

but $(l \in \Gamma) \Rightarrow (l \neq c)$, thus $c \trianglelefteq l$ becomes $c \triangleleft l$

$\Rightarrow (l \in \downarrow \mathrm{L}_\Gamma(x)) \vee \Big((l \in \Gamma) \wedge \big((c \wr x) \wedge (c \triangleleft l)\big)\Big)$

apply equation 8.29a (Lemma 8.17)

$\Rightarrow (l \in \downarrow \mathrm{L}_\Gamma(x)) \vee \Big((l \in \Gamma) \wedge \big((c \wr x) \wedge \big(\exists s \in G : (s \wr c) \wedge (s \triangleleft l)\big)\big)\Big)$

apply transitivity, and also $(s \neq c) \wedge (s \in G) \Rightarrow (s \in \Gamma)$

$\Rightarrow (l \in \downarrow \mathrm{L}_\Gamma(x)) \vee \Big((l \in \Gamma) \wedge \big(\exists s \in \Gamma : (s \wr x) \wedge (s \triangleleft l)\big)\Big)$

apply equation 8.2a (Definition 8.1)

$\Rightarrow (l \in \downarrow \mathrm{L}_\Gamma(x)) \vee (l \in \downarrow \mathrm{L}_\Gamma(x))$

simplify

$\Rightarrow (l \in \downarrow \mathrm{L}_\Gamma(x))$ □

Because the position of shadow nodes is defined by the graph's below-left-of and below-right-of sets, this lemma proves that all the shadow nodes in G^* preserve their positions in Γ^*. If removing an inconsequential node does not change the ordering of the nodes that remain, then, conversely, adding an inconsequential node will not change those orders either. We will rely on this property in the next section.

8.8 Incremental Graph Construction

We have seen that it is possible to create a shadowed graph from a pre-existing graph defined by two total orders (Section 8.4). Now we will show how it is possible easily and efficiently to extend the total orders for a graph as new inconsequential nodes (and their shadows) are added to G^*.

As we have seen, leaves in G (and other inconsequential nodes) have the property that they lie adjacent to their shadows. Thus, if we can determine where a new leaf's shadow c' lies in the two total orders, we will also have determined where c lies.

For simplicity, the following discussion will describe adding c' and c to the $\prec$ ordering (the $\succ$ ordering is analogous). Because it is inconvenient to talk about the position of c and c' in a graph that does not yet contain them, we will use a "backwards approach" and imagine that c and c' have already been added to the graph. Since c is an inconsequential node, c' immediately precedes c in the $\prec$ ordering. Suppose we find the node m that immediately precedes c' in $G^*_{\prec}$ and then delete c' and c. As we saw in Lemma 8.18, deleting c' and c does not invalidate the rest of the graph. Similarly, given m, we can reinsert c' and c into the graph. More importantly, given a means to find m, we can insert c' and c into the graph for the first time.

It is vital that c' and c be adjacent (and, thus, that c is an inconsequential node); otherwise, finding the insertion point for c' would not reveal c's position in the total orders.

Lemma 8.19 *If c satisfies the inconsequentiality condition, then the only nodes that may have the same below-left-of set (or below-right of set) as c are its ancestors.*

$$\begin{aligned}\big(\forall x^* \in G^* : (c \lhd x^*) \Rightarrow (c' \trianglelefteq x^*)\big) \Rightarrow \\ \forall l : (\downarrow \mathrm{L}_G(l) = \downarrow \mathrm{L}_G(c)) \Rightarrow (l \trianglelefteq c) \qquad (8.31a)\\ \forall r : (\downarrow \mathrm{R}_G(r) = \downarrow \mathrm{R}_G(c)) \Rightarrow (r \trianglelefteq c) \qquad (8.31b)\end{aligned}$$

Proof

This lemma is trivially true for leaves in G. To show that it is true for any node that satisfies the inconsequentiality condition requires a little more work. We will only prove

equation 8.31a; equation 8.31b is analogous.

$$(\downarrow \mathrm{L}_G(l) = \downarrow \mathrm{L}_G(c))$$

apply equation 8.8a

$$\Rightarrow (l \trianglelefteq c) \vee (c \lhd l) \tag{8.32}$$

but

$$(c \lhd l) \wedge \big(\forall x^* \in G^* : (c \lhd x^*) \Rightarrow (c' \trianglelefteq x^*)\big)$$

simplify (and discard)

$$\Rightarrow c' \trianglelefteq l$$

use equation 8.13 (and transitivity)

$$\Rightarrow c' \lhd l'$$

widen using definition of $\not\asymp$

$$\Rightarrow l' \not\asymp c'$$

from equation 8.11a (given $\downarrow \mathrm{L}_G(l) = \downarrow \mathrm{L}_G(c)$)

$$\Rightarrow l \lhd c$$

contradiction (8.33)

Combining equation 8.32 and contradiction 8.33 yields equation 8.31a. □

8.8.1 Parents and Siblings

As we shall see, the insertion algorithm for inconsequential nodes uses information about the leftmost and rightmost parents of the newly added node, and its left and right siblings. In this section we will formally define what is meant by these terms.

Parents

Definition 8.20 *When c has at least one parent, $\mathrm{P}_{\mathrm{L}}(c)$ (the leftmost parent of c) and $\mathrm{P}_{\mathrm{R}}(c)$ (the rightmost parent of c) are defined to be the unique nodes satisfying:*

$$\forall p, c \in G : (p \lhd c) \Rightarrow (\mathrm{P}_{\mathrm{L}}(c) \lhd c) \wedge (\mathrm{P}_{\mathrm{L}}(c) \lessdot p) \tag{8.34a}$$

$$\forall p, c \in G : (p \lhd c) \Rightarrow (\mathrm{P}_{\mathrm{R}}(c) \lhd c) \wedge (p \gtrdot \mathrm{P}_{\mathrm{R}}(c)) \tag{8.34b}$$

($\mathrm{P}_{\mathrm{L}}(c)$ *and* $\mathrm{P}_{\mathrm{R}}(c)$ *may refer to the same node.*)

A more conventional way to state the definition above is

$$\mathrm{P}_{\mathrm{L}}(c) \equiv \min_{\lessdot}\{p \in G \mid (p \lhd c)\}$$

$$\mathrm{P}_{\mathrm{R}}(c) \equiv \max_{\gtrdot}\{p \in G \mid (p \lhd c)\}$$

where $\min$ (or $\max$) of the empty set is undefined, meaning $\mathrm{P}_{\mathrm{L}}(c)$ (or $\mathrm{P}_{\mathrm{R}}(c)$) do not exist. The advantage of the form in Definition 8.20 is that the equations only apply when nodes have parents. In our discussion, we will assume that the graph begins with a root node so that every leaf that is added to the graph will have a parent, but the results generalize in the obvious way to the addition of inconsequential nodes that have no parent.

Parent Shadows

We also define the $\mathrm{P}'_{\mathrm{L}}(c)$ and $\mathrm{P}'_{\mathrm{R}}(c)$ to be the shadows of $\mathrm{P}_{\mathrm{L}}(c)$ and $\mathrm{P}_{\mathrm{R}}(c)$ respectively. Thus

$$p = \mathrm{P}_{\mathrm{L}}(c) \;\Leftrightarrow\; p' = \mathrm{P}'_{\mathrm{L}}(c)$$

$$p = \mathrm{P}_{\mathrm{R}}(c) \;\Leftrightarrow\; p' = \mathrm{P}'_{\mathrm{R}}(c)$$

Neighbours

Similar to our definition of leftmost and rightmost parents, we define the closest left and right neighbours of a node as follows:

Definition 8.21

$$\forall s, c \in G : (s \wr c) \Rightarrow (\mathrm{S_L}(c) \wr c) \wedge (s \prec \mathrm{S_L}(c)) \tag{8.36a}$$

$$\forall s, c \in G : (c \wr s) \Rightarrow (c \wr \mathrm{S_R}(c)) \wedge (\mathrm{S_R}(c) \succ s) \tag{8.36b}$$

A corollary of the above definition is

Corollary 8.22

$$(l \in {}^{\downarrow}\mathrm{L}_G(c)) \Rightarrow (\mathrm{S_L}(c) \in {}^{\downarrow}\mathrm{L}_G(c)) \wedge (l \prec \mathrm{S_L}(c)) \tag{8.37a}$$

$$(r \in {}^{\downarrow}\mathrm{R}_G(c)) \Rightarrow (\mathrm{S_R}(c) \in {}^{\downarrow}\mathrm{L}_G(c)) \wedge (\mathrm{S_R}(c) \succ r) \tag{8.37b}$$

Proof

Trivially from Definition 8.1. □

When $\mathrm{S_L}(c)$ and $\mathrm{S_R}(c)$ exist

$$\mathrm{S_L}(c) \equiv \max_{\prec} {}^{\downarrow}\mathrm{L}_G(c)$$

$$\mathrm{S_R}(c) \equiv \min_{\succ} {}^{\downarrow}\mathrm{R}_G(c).$$

We do not use these simpler equations because $\min$ and $\max$ are problematic if ${}^{\downarrow}\mathrm{L}_G(c)$ or ${}^{\downarrow}\mathrm{R}_G(c)$ are empty sets, whereas Definition 8.21 can only be used when these sets are not empty.

If a node c has a left sibling, $\mathrm{S_L}(c)$ will be the closest (rightmost) left sibling; otherwise, $\mathrm{S_L}(c)$ may refer to a more distant cousin of c (and analogously for $\mathrm{S_R}(c)$).

Parent Properties

The below-left-of set for the left parent can never be larger than the below-left-of set for its child (and similarly for below-right-of sets).

Lemma 8.23

$$\forall c \in G : {\downarrow}\mathrm{L}_G(\mathrm{P}_\mathrm{L}(c)) \subseteq {\downarrow}\mathrm{L}_G(c) \tag{8.39a}$$

$$\forall c \in G : {\downarrow}\mathrm{R}_G(\mathrm{P}_\mathrm{R}(c)) \subseteq {\downarrow}\mathrm{L}_G(c) \tag{8.39b}$$

Proof

As usual, we will only prove equation 8.39a, as equation 8.39b is analogous. First, we will show that $(s \wr \mathrm{P}_\mathrm{L}(c)) \Rightarrow (s \wr c)$:

$$(s \wr \mathrm{P}_\mathrm{L}(c)) \wedge (\mathrm{P}_\mathrm{L}(c) \lhd c)$$

rearrange (duplicate first term)

$$\Rightarrow (s \wr \mathrm{P}_\mathrm{L}(c)) \wedge \big((s \wr \mathrm{P}_\mathrm{L}(c)) \wedge (\mathrm{P}_\mathrm{L}(c) \lhd c)\big)$$

eliminate $\mathrm{P}_\mathrm{L}(c)$ in second term

$$\Rightarrow (s \wr \mathrm{P}_\mathrm{L}(c)) \wedge \big((s \wr c) \vee (s \lhd c)\big)$$

rearrange (and weaken)

$$\Rightarrow (s \wr c) \vee \big((s \wr \mathrm{P}_\mathrm{L}(c)) \wedge (s \lhd c)\big)$$

apply equation 8.34a

$$\Rightarrow (s \wr c) \vee \big((s \wr \mathrm{P}_\mathrm{L}(c)) \wedge (\mathrm{P}_\mathrm{L}(c) \not\prec s)\big)$$

eliminate contradiction

$$\Rightarrow s \wr c \tag{8.40}$$

Given the above, showing that ${\downarrow}\mathrm{L}_G(\mathrm{P}_\mathrm{L}(c)) \subseteq {\downarrow}\mathrm{L}_G(c)$ is now trivial:

$$l \in {\downarrow}\mathrm{L}_G(\mathrm{P}_\mathrm{L}(c))$$

apply equation 8.2a

$$\Rightarrow \exists s : (s \wr \mathrm{P}_\mathrm{L}(c)) \wedge (s \trianglelefteq l)$$

apply equation 8.40

$$\Rightarrow \exists s : (s \wr c) \wedge (s \trianglelefteq l)$$

apply equation 8.2a

$$\Rightarrow l \in {\downarrow}\mathrm{L}_G(c)$$

□

Neighbour Properties

As we have seen, $\mathrm{S_L}(c)$ is the last node in $\downarrow \mathrm{L}_G(c)$, provided $\downarrow \mathrm{L}_G(c) \neq \varnothing$. Thus,

Lemma 8.24

$$\mathrm{S_L}(c) \in \downarrow \mathrm{L}_G(x) \;\Rightarrow\; \downarrow \mathrm{L}_G(c) \subseteq \downarrow \mathrm{L}_G(x) \tag{8.41a}$$

$$\mathrm{S_R}(c) \in \downarrow \mathrm{R}_G(x) \;\Rightarrow\; \downarrow \mathrm{R}_G(c) \subseteq \downarrow \mathrm{R}_G(x) \tag{8.41b}$$

Proof

As usual, we will only prove equation 8.41a, as equation 8.41b is analogous. We will prove that $\downarrow \mathrm{L}_G(c) \subseteq \downarrow \mathrm{L}_G(x)$ by showing that $l \in \downarrow \mathrm{L}_G(c) \;\Rightarrow\; l \in \downarrow \mathrm{L}_G(x)$:

$$(l \in \downarrow \mathrm{L}_G(c)) \wedge (\mathrm{S_L}(c) \in \downarrow \mathrm{L}_G(x))$$

apply equation 8.37a

$$\Rightarrow \big((\mathrm{S_L}(c) \in \downarrow \mathrm{L}_G(c)) \wedge (l \prec \mathrm{S_L}(c))\big) \wedge (\mathrm{S_L}(c) \in \downarrow \mathrm{L}_G(x))$$

discard

$$\Rightarrow (l \prec \mathrm{S_L}(c)) \wedge (\mathrm{S_L}(c) \in \downarrow \mathrm{L}_G(x))$$

apply equation 8.5a

$$\Rightarrow l \in \downarrow \mathrm{L}_G(x)$$ □

We can now show an important relationship between a node's left parent and its left neighbour.

Lemma 8.25

$$\forall c \in G : (\exists l \in G : l \;\wr\; c) \;\Rightarrow\; \big(\mathrm{S_L}(c) \not\prec \mathrm{P'_L}(c) \;\Leftrightarrow\; \downarrow \mathrm{L}_G(c) = \downarrow \mathrm{L}_G(\mathrm{P_L}(c))\big) \tag{8.42a}$$

$$\forall c \in G : (\exists r \in G : c \;\wr\; r) \;\Rightarrow\; \big(\mathrm{P'_R}(c) \not\succ \mathrm{S_R}(c) \;\Leftrightarrow\; \downarrow \mathrm{R}_G(c) = \downarrow \mathrm{R}_G(\mathrm{P_R}(c))\big) \tag{8.42b}$$

(*Proof begins on next page...*)

Proof

As usual, we will only prove equation 8.42a, as equation 8.42b is analogous. First, we will prove it in the forwards direction:

$\mathrm{S}_{\mathrm{L}}(c)$ exists, because we are assuming $\exists l \in G : l \curlywedge c$

$\mathrm{S}_{\mathrm{L}}(c) \not\asymp \mathrm{P}'_{\mathrm{L}}(c)$

apply equation 8.10a from Definition 8.7

$\Rightarrow \mathrm{S}_{\mathrm{L}}(c) \in {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c))$

apply equation 8.41a

$\Rightarrow {\downarrow}\mathrm{L}_G(c) \subseteq {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c))$

include result of equation 8.39a

$\Rightarrow ({\downarrow}\mathrm{L}_G(c) \subseteq {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c))) \wedge ({\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subseteq {\downarrow}\mathrm{L}_G(c))$

simplify

$\Rightarrow {\downarrow}\mathrm{L}_G(c) = {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c))$

Now we shall prove it in the other direction:

$(\exists l \in G : l \curlywedge c) \wedge ({\downarrow}\mathrm{L}_G(c) = {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)))$

apply equation 8.36a

$\Rightarrow \big(\exists l \in G : (\mathrm{S}_{\mathrm{L}}(c) \curlywedge c) \wedge (l \prec \mathrm{S}_{\mathrm{L}}(c))\big) \wedge ({\downarrow}\mathrm{L}_G(c) = {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)))$

discard

$\Rightarrow (\mathrm{S}_{\mathrm{L}}(c) \curlywedge c) \wedge ({\downarrow}\mathrm{L}_G(c) = {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)))$

apply equation 8.2a

$\Rightarrow (\mathrm{S}_{\mathrm{L}}(c) \in {\downarrow}\mathrm{L}_G(c)) \wedge ({\downarrow}\mathrm{L}_G(c) = {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)))$

equality

$\Rightarrow \mathrm{S}_{\mathrm{L}}(c) \in {\downarrow}\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c))$

apply equation 8.10a

$\Rightarrow \mathrm{S}_{\mathrm{L}}(c) \not\asymp \mathrm{P}'_{\mathrm{L}}(c)$ $\square$

From equation 8.15a, we can see that when $\mathrm{S}_{\mathrm{L}}(c)$ is not a direct sibling of c but a more distant cousin—and thus $\mathrm{S}_{\mathrm{L}}(c) \wr \mathrm{P}_{\mathrm{L}}(c)$—then $\mathrm{S}_{\mathrm{L}}(c) \wr \mathrm{P}'_{\mathrm{L}}(c)$, and, from the above lemma, $\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) = \downarrow \mathrm{L}_G(c)$. Similarly, looking at this result from the reverse direction, if $\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \neq \downarrow \mathrm{L}_G(c)$—that is, when $\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subset \downarrow \mathrm{L}_G(c)$)—then $\mathrm{S}_{\mathrm{L}}(c) \not\wr \mathrm{P}_{\mathrm{L}}(c)$ and thus $\mathrm{P}_{\mathrm{L}}(c) \lhd \mathrm{S}_{\mathrm{L}}(c)$ (because $(\mathrm{S}_{\mathrm{L}}(c) \wr c) \wedge (\mathrm{P}_{\mathrm{L}}(c) \lhd c) \Rightarrow (\mathrm{S}_{\mathrm{L}}(c) \wr \mathrm{P}_{\mathrm{L}}(c)) \vee (\mathrm{P}_{\mathrm{L}}(c) \lhd \mathrm{S}_{\mathrm{L}}(c))$).

8.8.2 The Two Cases for Insertion

In Lemma 8.23, we saw that $\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subseteq \downarrow \mathrm{L}_G(c)$. We will consider the two possibilities of this result separately.

1. $\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subset \downarrow \mathrm{L}_G(c)$ — We will call this case the **germane sibling** case.[3] In this case, c has a sibling, s, where $s = \mathrm{S}_{\mathrm{L}}(c)$ and $\mathrm{P}_{\mathrm{L}}(s) = \mathrm{P}_{\mathrm{L}}(c)$.

2. $\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) = \downarrow \mathrm{L}_G(c)$ — We will call this case the **no germane sibling** case. In this case, c either has no sibling at all, or it has an **avuncular sibling**—that is, a sibling that is also a child of a parent that is to the left of $\mathrm{P}_{\mathrm{L}}(c)$ (a child of c's leftward uncle).

(analogously, right siblings are also germane or avuncular—for simplicity, we will focus on left siblings and left parents in our discussion).

Figure 8.6 provides examples of both of these cases. For example, if F were to spawn a child between nodes G and H, then G would be an avuncular left sibling of the new node, because G is a child of both E and F. In contrast, if this new node were to be inserted in the same place but had both E and F as parents, G would be a germane sibling.

To complete the picture, Figure 8.7 shows all of the possible scenarios for inserting a new node into a graph, showing how each of them neatly falls into one or other of the two cases given above.

We will show that the position of c' in $\prec$ is solely determined by the left sibling of c (which is $\mathrm{S}_{\mathrm{L}}(c)$) and the shadow of the left parent of c, $\mathrm{P}'_{\mathrm{L}}(c)$—$c'$ will lie after the greater of these two nodes (under $\prec$).[4] This disarmingly simple rule is the essence of the insertion

3. The root of germane is the Middle-English word *germain*, which literally means "having the same parents", making the term a relevant choice because $\mathrm{S}_{\mathrm{L}}(c)$ has the same left parent as c in this case. More importantly, only these kinds of sibling will turn out to be relevant.

4. This property that may appear "obvious" from Figure 8.7, but such an examination of cases is an informal approach, whereas we seek a more formal proof.

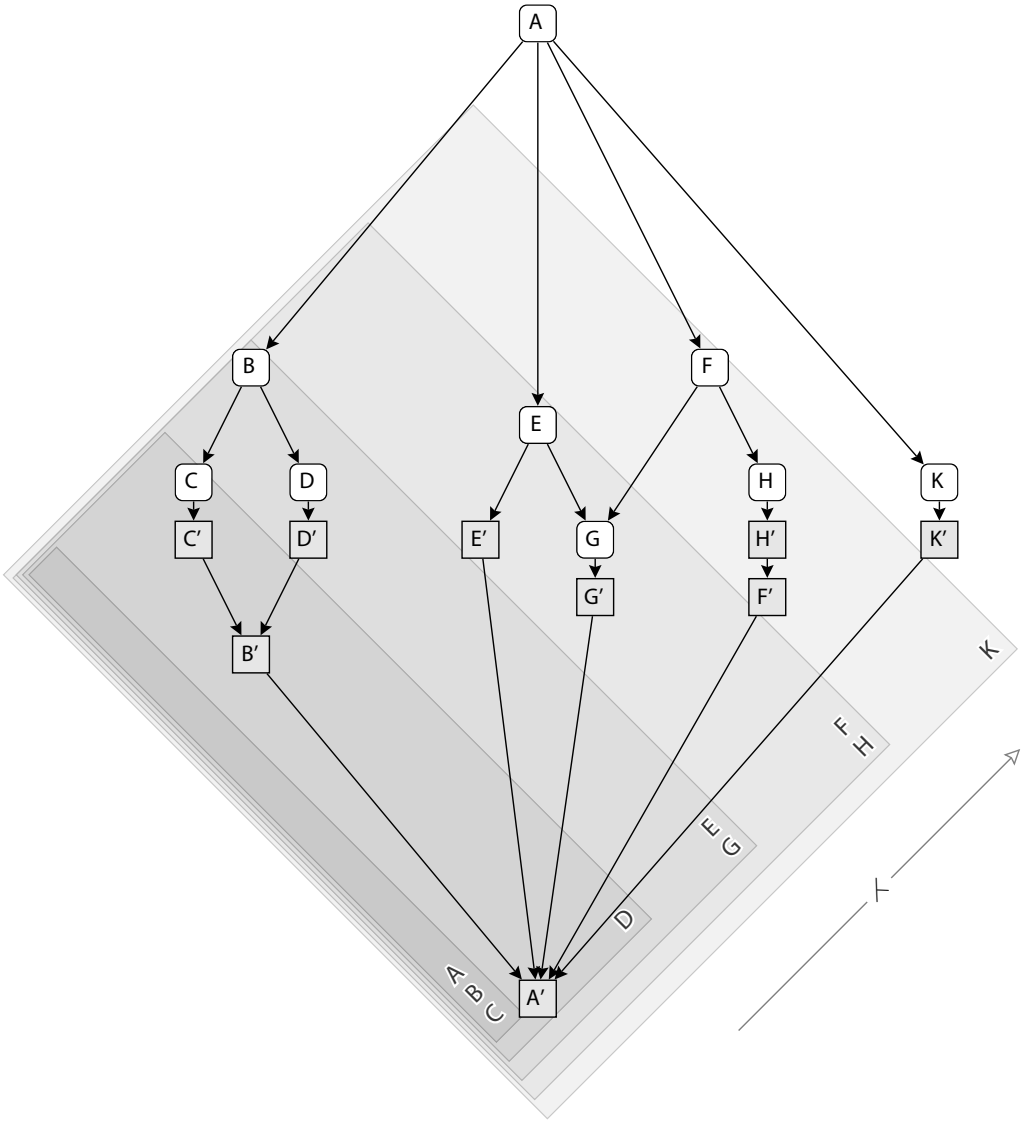
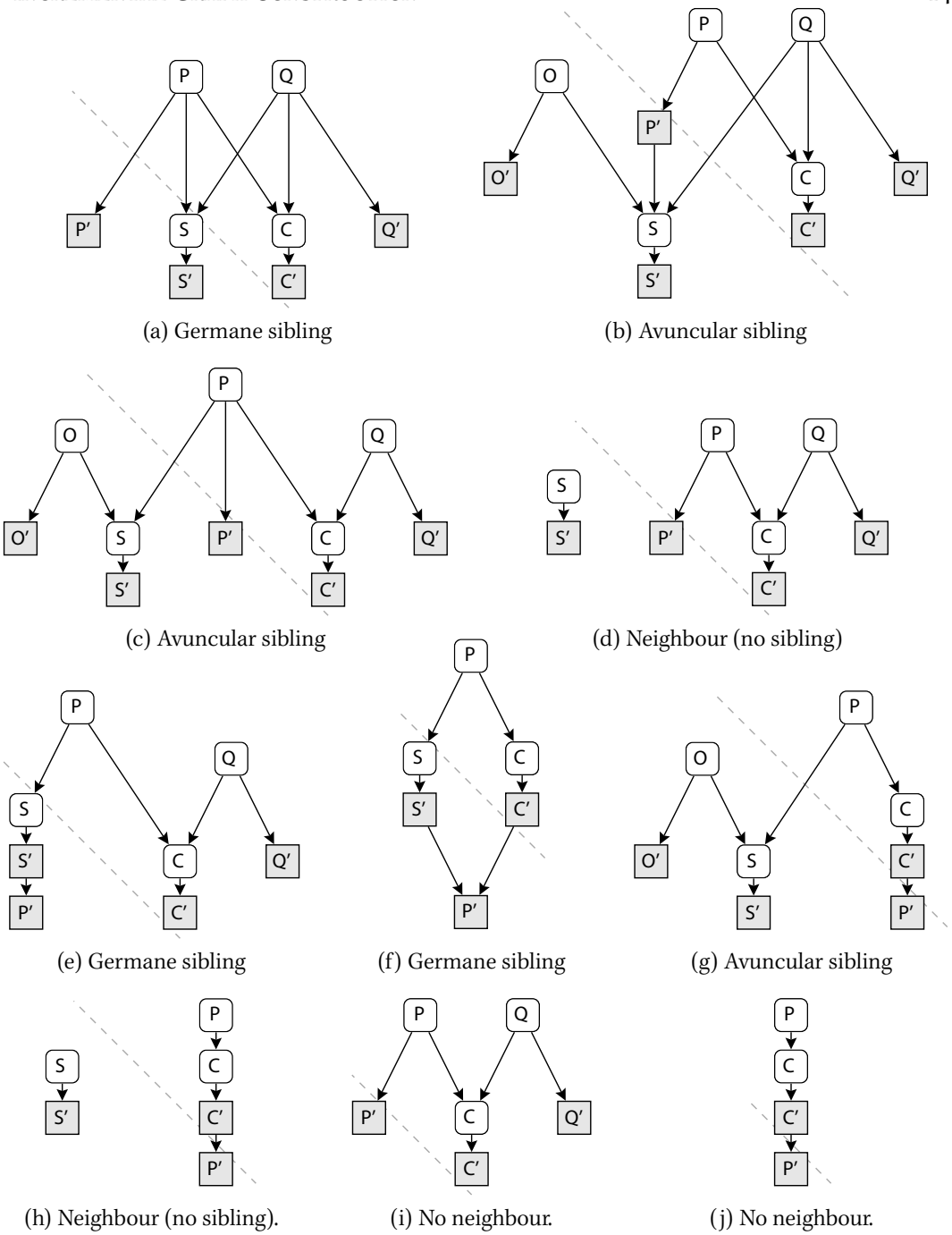


Figure 8.6: A partially complete graph. This graph is identical to the graph shown in Figure 8.2 except that nodes I, J, and L have yet to be added.



(a) Germane sibling

(b) Avuncular sibling

(c) Avuncular sibling

(d) Neighbour (no sibling)

(e) Germane sibling

(f) Germane sibling

(g) Avuncular sibling

(h) Neighbour (no sibling).

(i) No neighbour.

(j) No neighbour.

Figure 8.7: Cases for insertion. In these examples, P is the left parent of C, and S (if present) is the left neighbour of C. Rotating the page 45° clockwise allows us to see the node ordering under $\prec$ (running left-to-right)—the dotted lines are provided as a reader aid for determining which node precedes C′ under $\prec$.

algorithm given in Section 7.3 of Chapter 7. Showing that this rule is correct has been our task in this chapter, and we are now in the final stages of that proof.

Lemma 8.26 *If c satisfies the inconsequentiality condition and has no germane left sibling then $\mathrm{P}'_{\mathrm{L}}(c)$ is the last of all the shadow nodes (excluding c') following $\downarrow\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c))$ in the $\prec$ ordering. That is,*

$$\big(\forall x^* \in G^* : (c \lhd x^*) \Rightarrow (c' \trianglelefteq x^*)\big) \Rightarrow$$

$$(\downarrow\mathrm{L}_G(p) = \downarrow\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) = \downarrow\mathrm{L}_G(c)) \;\Rightarrow\; \big((p = c) \vee (p \trianglelefteq \mathrm{P}_{\mathrm{L}}(c))\big) \tag{8.43a}$$

$$(\downarrow\mathrm{R}_G(p) = \downarrow\mathrm{R}_G(\mathrm{P}_{\mathrm{R}}(c)) = \downarrow\mathrm{R}_G(c)) \;\Rightarrow\; \big((p = c) \vee (p \trianglelefteq \mathrm{P}_{\mathrm{R}}(c))\big) \tag{8.43b}$$

Proof

As usual, we will show only equation 8.43a; equation 8.43b is analogous. We will prove this lemma by showing that of all the nodes that share the same below-left-of set as $\mathrm{P}_{\mathrm{L}}(c)$ (except c), $\mathrm{P}_{\mathrm{L}}(c)$ is the lowest in the graph (i.e., all the other nodes are ancestors of $\mathrm{P}_{\mathrm{L}}(c)$). This property, coupled with equation 8.11a of Definition 8.7, gives our desired result. Let us first observe that equation 8.31a (Lemma 8.19) applies, and so we have

$$\downarrow\mathrm{L}_G(p) = \downarrow\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) = \downarrow\mathrm{L}_G(c)$$

equality

$$\Rightarrow \big(\downarrow\mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) = \downarrow\mathrm{L}_G(p)\big) \wedge \big(\downarrow\mathrm{L}_G(c) = \downarrow\mathrm{L}_G(p)\big)$$

apply equation 8.8a to first term and and equation 8.31a to second term

$$\Rightarrow \big((p \trianglelefteq \mathrm{P}_{\mathrm{L}}(c)) \vee (\mathrm{P}_{\mathrm{L}}(c) \lhd p)\big) \wedge \big((p \lhd c) \vee (p = c)\big)$$

rearrange (distribute and discard)

$$\Rightarrow \Big(\big((p \trianglelefteq \mathrm{P}_{\mathrm{L}}(c)) \vee (\mathrm{P}_{\mathrm{L}}(c) \lhd p)\big) \wedge (p \lhd c)\Big) \vee (p = c)$$

apply equation 8.34a

$$\Rightarrow \Big(\big((p \trianglelefteq \mathrm{P}_{\mathrm{L}}(c)) \vee (\mathrm{P}_{\mathrm{L}}(c) \lhd p)\big) \wedge (\mathrm{P}_{\mathrm{L}}(c) \prec p)\Big) \vee (p = c)$$

simplify

$$\Rightarrow (p \trianglelefteq \mathrm{P}_{\mathrm{L}}(c)) \vee (p = c)$$

□

Thus, if c has no germane left sibling, then c' lies directly after p' in $G_{\prec}$ and c lies directly after c'.

Lemma 8.27 *If c satisfies the inconsequentiality condition and has a germane left sibling, then there is no other node in G that has the same below-left-of set as c.*

$$\Big(\forall x^* \in G^* : (c \lhd x^*) \Rightarrow (c' \trianglelefteq x^*)\Big) \Rightarrow$$
$$\Big((\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subset \downarrow \mathrm{L}_G(c)) \wedge (\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c))\Big) \;\Rightarrow\; (x = c) \tag{8.44a}$$
$$\Big((\downarrow \mathrm{R}_G(\mathrm{P}_{\mathrm{R}}(c)) \subset \downarrow \mathrm{R}_G(c)) \wedge (\downarrow \mathrm{R}_G(x) = \downarrow \mathrm{R}_G(c))\Big) \;\Rightarrow\; (x = c) \tag{8.44b}$$

Proof

We will show only equation 8.44a; equation 8.44b is analogous. Let us first observe that Lemma 8.19 applies, and thus we have

$$(\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c))$$
$$\Rightarrow (x = c) \vee (x \lhd c)$$

We will show that the $x \lhd c$ case leads to a contradiction.

$$(x \lhd c)$$

apply equation 8.34a

$$\Rightarrow (\mathrm{P}_{\mathrm{L}}(c) \prec x)$$

expand according to definition of $\prec$

$$\Rightarrow (\mathrm{P}_{\mathrm{L}}(c) = x) \wedge (\mathrm{P}_{\mathrm{L}}(c) \wr x) \wedge (\mathrm{P}_{\mathrm{L}}(c) \lhd x)$$

Each of these possibilities leads to a contradiction. First, suppose $\mathrm{P}_{\mathrm{L}}(c) = x$:

$$\mathrm{P}_{\mathrm{L}}(c) = x$$

equality

$$\Rightarrow \downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) = \downarrow \mathrm{L}_G(x)$$

contradicts $(\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subset \downarrow \mathrm{L}_G(c)) \wedge (\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c))$

Second, suppose $\mathrm{P}_{\mathrm{L}}(c) \wr x$:

$$(\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c)) \wedge (\mathrm{P}_{\mathrm{L}}(c) \wr x)$$

apply equation 8.2a

$$\Rightarrow (\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c)) \wedge (\mathrm{P}_{\mathrm{L}}(c) \in \downarrow \mathrm{L}_G(x))$$

equality

$$\Rightarrow (\mathrm{P}_{\mathrm{L}}(c) \in \downarrow \mathrm{L}_G(c))$$

apply equation 8.4a

$$\Rightarrow \mathrm{P}_{\mathrm{L}}(c) \not\asymp c$$

contradicts $\mathrm{P}_{\mathrm{L}}(c) \lhd c$

Finally, let us consider the last case, $x \lhd \mathrm{P}_{\mathrm{L}}(c)$:

$$(\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c)) \wedge (\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subset \downarrow \mathrm{L}_G(c)) \wedge (x \lhd \mathrm{P}_{\mathrm{L}}(c))$$

apply rules for equality

$$\Rightarrow (\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c)) \wedge \big((\downarrow \mathrm{L}_G(\mathrm{P}_{\mathrm{L}}(c)) \subset \downarrow \mathrm{L}_G(x)) \wedge (x \lhd \mathrm{P}_{\mathrm{L}}(c))\big)$$

apply equation 8.9a

$$\Rightarrow (\downarrow \mathrm{L}_G(x) = \downarrow \mathrm{L}_G(c)) \wedge (\mathrm{P}_{\mathrm{L}}(c) \in \downarrow \mathrm{L}_G(x))$$

apply rules for equality

$$\Rightarrow \mathrm{P}_{\mathrm{L}}(c) \in \downarrow \mathrm{L}_G(c)$$

apply equation 8.4a

$$\Rightarrow \mathrm{P}_{\mathrm{L}}(c) \not\asymp c$$

contradicts $\mathrm{P}_{\mathrm{L}}(c) \lhd c$

All the cases stemming from $x \lhd c$ lead to a contradiction, thus the only possibility that remains is $x = c$. □

Therefore, if c has a germane left sibling, c' will lie immediately after that sibling.

8.8.3 The Rule for Insertion

We have shown that, in the $\prec$ total order, c' will lie immediately

1. After the germane left sibling, if there is one
2. After the shadow of the left parent, if there is no germane left sibling

From equation 8.42a (Lemma 8.25), we can see if $\mathrm{S}_{\mathrm{L}}(c) \not\prec \mathrm{P}'_{\mathrm{L}}(c)$, then $\mathrm{S}_{\mathrm{L}}(c)$ is not a germane left sibling. Thus, it is easy to determine whether or not a given left sibling is germane. Similarly, if c has no left sibling ($\mathrm{S}_{\mathrm{L}}(c)$ is merely neighbour, or is undefined), then the second rule applies.

We have found the insertion point for c', and, because c lies immediately after c', the insertion point for c.

The rule for inserting c' and c into $\succ$ is analogous: c' will lie immediately

1. Before the germane right sibling, if there is one
2. Before the shadow of the right parent, if there is no germane left sibling

and c lies immediately before c'.

8.9 Conclusion

We have developed a theory to support the dynamic creation of LR-graphs by maintaining shadow nodes. The proof is rather long, but I hope it provides some insight into why the algorithm presented in Chapter 7 works.

Chapter 9

Space and Time Complexity of the LR-tags Method

In the preceding chapters, I presented the foundations of the LR-tags method for determinacy checking. In this chapter, I will discuss its time and space complexity.

9.1 Space Complexity

The space overheads of determinacy checking come from two sources: the space needed to maintain w, r_f, and r_b for each determinacy-checked shared object, and the space needed for the ordered lists that represent task relationships. Clearly, the first overhead is constant, so our discussion will focus on showing that the space required to represent tasks in the ordered lists is also constant.

Task relationships are entirely expressed by the $\prec$ and $\succ$ relations, embodied in the $G_\prec$ and $G_\succ$ ordered lists. Ordered lists require space proportional to the length of the list, so the space required to represent task relationships is proportional to the size of $G_\prec$ and $G_\succ$. A loose bound for the size of $G_\prec$ and $G_\succ$ is $O(s)$, where s is the final size of the task DAG they represent, but some applications have very large task DAGs without requiring much memory to execute, so we will seek a tighter bound.

If we remove those elements of $G_\prec$ and $G_\succ$ that cannot be referenced by the determinacy checker, we can obtain a tighter bound on the size of these ordered lists. By

definition, nascent tasks do not need to be included in $G_{\prec}$ and $G_{\succ}$ because they cannot perform any actions—tasks only need to be added when they become effective. Similarly, a simple reference-counting strategy is sufficient to ensure that ethereal tasks that are no longer referenced as readers or writers of any data item are removed from $G_{\prec}$ and $G_{\succ}$. (Reference counting adds a constant space overhead to each ordered-list item.)

Thus, if there are v determinacy-checked objects in the program, those objects could refer to at most $3v$ tasks. The only other references into $G_{\prec}$ and $G_{\succ}$ come from effective tasks, which, by definition, are either currently running or are represented in the task scheduler for other reasons (see Section 6.2).

The space overheads of determinacy checking are thus $O(v + e)$, where v is the number of determinacy-checked objects and e is the maximum number of effective tasks that existed during the program's run.

Usually, $e \ll v$, in which case the space overheads simplify to $O(v)$, but we should nevertheless consider that e could be greater than v.[1] As we discussed in Section 6.2, the upper bound for e depends on both the task scheduler and the program being executed. Because the effective tasks of a program are defined to be exactly those that the task scheduler must represent, it is reasonable to assume that a general-purpose task scheduler will store information about the effective tasks of the program, requiring $\Omega(e)$ space to do so. Thus the $O(e)$ space required to represent tasks in $G_{\prec}$ and $G_{\succ}$ is mirrored by $\Omega(e)$ space used in task scheduler. (Highly specialized task schedulers may be able to avoid storing information about the relationships between tasks they schedule, but such schedulers may also admit highly specialized implementations of $\prec$ and $\succ$; these specialized systems are beyond the scope of this dissertation.)

Figure 9.1 illustrates some of these memory-use issues by returning to the example first shown in Figure 6.2. It shows the per-object overheads of determinacy checking, as well as revealing which items in the ordered list may be deleted. In this example, we can see that the number of effective tasks is dwarfed by the amount of determinacy checked data. (Notice also how different objects use determinacy checking at different granularities; the string str is checked as a single object, whereas the array chr treats each character as a separate determinacy-checked object.)

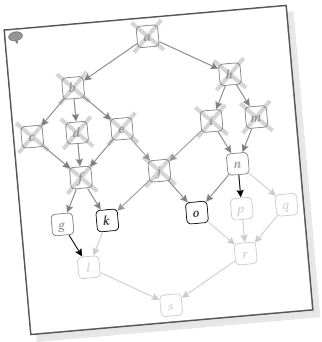
1. Perhaps this common case explains why most published literature on the subject fails to take account of the space required to represent the effective tasks of the program when assessing the space requirements of determinacy checking

Task State

Tasks Executing : *k o*

Other Effective Tasks : *g n*

Ethereal Tasks : *a b c d e f h i j m*



Determinacy-Checked Shared Objects

re :	0.6811	10.321	2.1111	1.1703	210.81	0.5107	101.03
im :	21.121	4.1010	171.22	12.210	81.011	15.063	0.3333
r_f :	*o*	*o*	*n*	*n*	*n*	*n*	*n*
r_b :	*o*	*n*	*n*	*n*	*n*	*n*	*n*
w :	*o*	*n*	*n*	*n*	*n*	*n*	*n*

str :	The sun was shining on the sea
r_f :	*a*
r_b :	*a*
w :	*a*

chr :	S	h	i	n	i	n	g		W	i	t	h		A	l	l		i	t	s		m	i	g	h	t
r_f :	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*d*	*d*	*e*	*e*	*e*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
r_b :	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*d*	*d*	*e*	*e*	*e*	*e*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
w :	*c*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*d*	*a*	*a*	*a*	*a*	*e*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*

Ordered Lists

↘ : *a b c c' d d' e f g g' f' e' b' h i j k k' j' i' m n o o' n' m' h' a'*

↙ : *a' b' c' f' g' g k' k f c d' d e' j' o' o j e b h' i' n' n i m' m h a*

Figure 9.1: Memory use for the LR-tags method. This figure shows a moment during the execution of a program that shares an array of complex numbers, a string, and an array of characters between multiple threads. In the ordered lists, entries shown faintly in light grey can be removed because they are no longer referenced. (The small figure "pinned" to the page is a repeat of Figure 6.2.)

Thus the space overheads of determinacy checking both general nested-parallel and producer–consumer programs mirror the overheads of the programs themselves, resulting in constant-factor space overhead for determinacy checking.

9.2 Serial Time Complexity

For serial execution of a parallel program, the time overheads are governed by the time complexity of the ordered-list algorithm. The best currently known algorithm (Dietz & Sleator, 1987) is a real-time algorithm. Thus, the LR-tags algorithm can run in constant time for a parallel program executed using a serial scheduler.

As I mentioned in Section 7.3.2, the actual algorithm I use in my implementation of the LR-tags algorithm uses constant amortized time for list insertions. In normal use, comparisons (which require constant worst-case time) dominate insertions by a significant margin (see Table 11.1 in Chapter 11), so the impact of the amortization in the simpler of Dietz and Sleator's algorithms is slight.

9.3 Parallel Time Complexity

The time complexity of the LR-tags method is more difficult to assess when it runs in parallel. When running on a multiprocessor machine, multiple processors may attempt to access the same ordered list at the same time—these accesses must be arbitrated and sometimes serialized, as we discussed in Section 7.3.2. Serialization can introduce delays in accessing the ordered list and prevent accesses from taking constant-time.

As I show in Chapter 11, the actual overheads of serialization depend on the program—for many programs, serialization does not cause significant overhead.

9.4 Conclusion

The LR-tags method has good asymptotic time complexity, offering better time and space bounds than existing techniques (see Table 6.1). But although "constant-factor overheads" are good in theory, the size of the constant is what matters in practice. In

the chapters that follow, we will examine ways to reduce the size of this constant, and then examine the results of tests to determine whether the LR-tags technique is indeed a practical one.

Chapter 10

Optimizations to the LR-tags Technique

In this chapter we will examine some useful optimizations that can be applied to the LR-tags technique. Although these optimizations do not improve the theoretical complexity of our approach, they can be useful in reducing the overheads involved in performing determinacy checking in practice.

10.1 Speeding Serial Performance

If execution is performed serially, following a left-to-right depth-first strategy that respects the commencement restrictions of the DAG, the order of tasks in $G_{\diagdown}$ is identical to the order in which tasks are executed (as mentioned in the discussion of Figure 7.2 in Section 7.2). Thus, in this case, an ordered-list data structure is not required to represent $G_{\diagdown}$.

In fact, $G_{\diagdown}$ is completely irrelevant in the serial case, because every "$r_b(d) \diagdown t$" and "$w(d) \diagdown t$" comparison (described in Section 7.2) will evaluate to true. The only way these comparisons could evaluate to false would be for a task from later in the serial execution sequence to have already read or written the datum, and that is clearly impossible.

Thus, for serial execution, there is little point in storing $r_b(d)$ for each datum d or maintaining a $\diagdown$ relation. The following rules are sufficient for checking that reads

and writes performed by a task t on a datum d are deterministic when the program is executed following a left-to-right depth-first strategy:

- *Reads* — A read is valid if $t \prec w(d)$—that is, if it satisfies condition Bern-1. If the read is valid, $r_f(d)$ may need to be updated: If $t \prec r_f(d)$ then $r_f(d) := t$
- *Writes* — A write is valid if $t \prec r_f(d)$—that is, if it satisfies condition Bern-2. If the write is valid, $w(d)$ and $r_f(d)$ are updated to be t

These optimizations do not improve the asymptotic time or space bounds for serial execution, but they do improve performance in practice by reducing the time and space required for determinacy checking by a constant factor.

10.2 Optimizations to $G_{\prec}$ and $G_{\prec}$

In equation 7.7 of Section 7.2, we saw that

$$(y \prec x) \wedge (x \succ y) \;\Leftrightarrow\; x \preceq y;$$

and how the important properties of the LR-tags determinacy checker (such as Theorem 7.2) are derived from this equation. What may be less obvious is that there are other possible definitions for $\prec$ and $\succ$ besides those of equations 7.5 and 7.6 that also allow us to derive this equation.

A sufficient specification for $\prec$ and $\succ$ is

$$(x \lhd y) \;\Leftrightarrow\; (y \prec x) \wedge (x \succ y) \wedge (x \neq y) \tag{10.1}$$

$$(x \wr y) \;\Leftrightarrow\; (x \prec y) \wedge (x \succ y) \wedge (x \neq y) \tag{10.2}$$

$$(x = y) \;\Leftrightarrow\; (x \prec y) \wedge (y \prec x) \wedge (x \succ y) \wedge (y \succ x). \tag{10.3}$$

This specification for $\prec$ and $\succ$ contrasts strongly with the definitions we saw earlier (equations 7.5 and 7.6). Those definitions lead to the following equation for task equality:

$$(x = y) \;\Leftrightarrow\; (x \prec y) \wedge (y \prec x) \;\Leftrightarrow\; (x \succ y) \wedge (y \succ x), \qquad \text{[not necessarily true]}$$

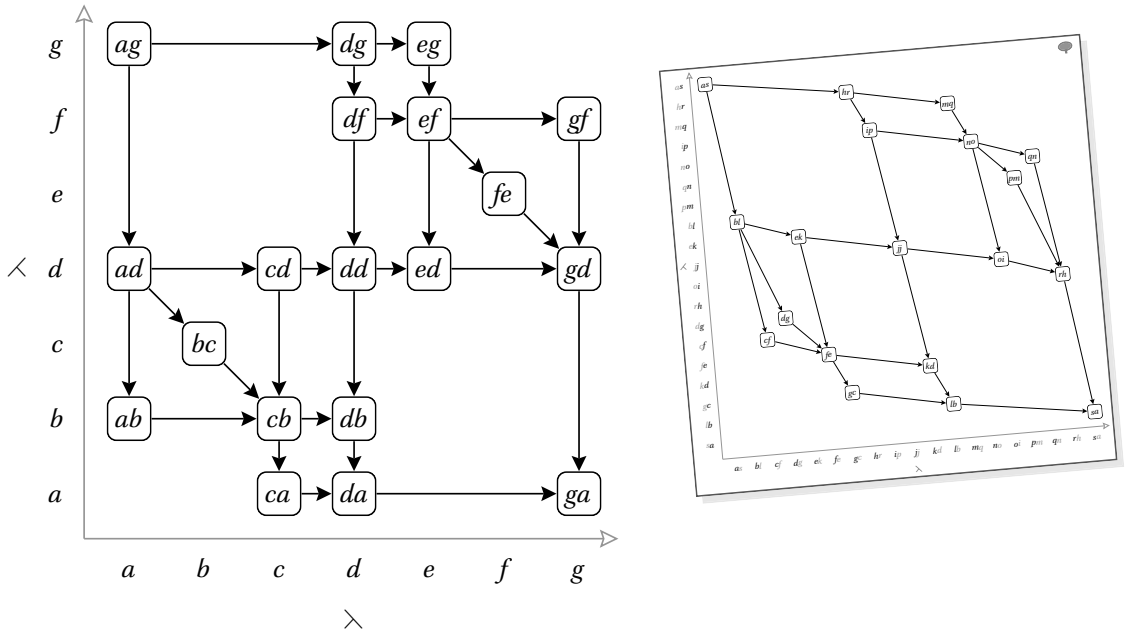


Figure 10.1: This diagram shows a graph isomorphic to the graph shown in Figure 7.3 (which is repeated here in the graph "pinned" to the page), in the form of a compact dominance drawing.

which is *not* implied by equations 10.1, 10.2 and 10.3. These looser specifications for $\prec$ and $\succ$ have the advantage that they allow us to represent a task DAG more compactly, sometimes sharing a single entry in $G_{\prec}$ or $G_{\succ}$ between several tasks. From a theoretical perspective, the $\prec$ and $\succ$ relations have become partial orders, although from an implementation perspective, $G_{\prec}$ or $G_{\succ}$ would remain total orders, with each task having a pointer to an entry in each ordered list (but some entries would be shared between multiple tasks).

Figure 10.1 shows a graph that is isomorphic to the graphs in Figures 7.2 and 7.3, in which tasks have been labeled following this compact representation. Like Figure 7.3, this graph is a dominance drawing—in fact, it is common practice to portray dominance drawings in this compact form (Di Battista et al., 1999).[1]

From this diagram and equations 10.1, 10.2, and 10.3, we can observe that a fork node

1. It is also common practice, however, to then rotate the graph by 45 degrees so that equations 7.5 and 7.6 continue to hold.

can share its entry in $G_\searrow$ with its leftmost child (provided that it is either the leftmost or only parent of that child), and share its entry in $G_\swarrow$ with its rightmost child (provided that it is either the rightmost or only parent of that child). Similarly, a join node may share the same entry in $G_\swarrow$ as its leftmost parent (provided that it is either the leftmost child or the only child of that parent) and share the same entry in $G_\searrow$ as its rightmost parent (provided that it is either the rightmost child or the only child of that parent).

The algorithm for maintaining $G_\swarrow$ and $G_\searrow$ that we examined in Chapters 7 and 8 does not use this alternative scheme, however. That algorithm allows new leaves to be added to the graph at arbitrary points, and thus cannot know whether a newly added leaf will remain the leftmost or rightmost child of that node, or whether more children will be added later.

In some cases, however, we can adopt a simpler algorithm for maintaining $G_\swarrow$ and $G_\searrow$ that allows tasks to share entries in $G_\swarrow$ and $G_\searrow$ and avoids using shadow nodes. For example, programs using nested parallelism can adopt the following strategy:

- *Fork:* When a node p forks multiple children, $c_1, \ldots, c_i$, c_1 shares the same entry as p in $G_\searrow$, while entries for $c_2, \ldots, c_i$ are inserted into $G_\searrow$ immediately after the entry shared by p and c_1. Similarly, c_i shares the same entry as p in $G_\swarrow$, with $c_{i-1}, \ldots, c_1$ inserted into $G_\swarrow$ immediately before the entry shared by p and c_i.

- *Join:* When a node c joins multiple parents, $p_1, \ldots, p_i$, c shares the same entry as p_1 in $G_\swarrow$ and the same entry as p_i in $G_\searrow$.

This particular algorithm is potentially asymptotically worse than the general algorithm in the case of graphs where the number of children per node is unbounded, because it inserts all the children of a node into $G_\swarrow$ and $G_\searrow$ together. A more complex specialization of the general insertion algorithm could no doubt address this deficiency, but such additional optimization would only be useful if the overheads of both of the algorithms we have examined were to prove significant in practice.

This optimization is orthogonal to the serial-execution optimization outlined in Section 10.1; thus both optimizations can be used together.

10.3 Relation Caching

In this section, we will examine an optimization that can reduce the number of order queries performed on the ordered lists representing $G_{\prec}$ and $G_{\succ}$. Reducing the number of queries made to the ordered list reduces the potential for contention between accesses to this shared data structure.

As we saw in Section 7.3, the relationships between existing tasks do not change as new tasks are spawned. Thus, given two tasks, x and y, we need only test $x \prec y$ once, because the result of the order query will always be the same, even if other tasks are added or removed from the ordered list (and similarly for $\succ$). (The answer *would* change if either x or y were removed from the ordered lists, but x and y are not ready to be deleted from the ordered lists if they are still being used in task comparisons—see Sections 9.1 and 10.5 for discussions of when and how we may remove tasks from the ordered lists.[2])

Although $\prec$ and $\succ$ are binary relations, the queries performed by a task t during determinacy checking are always of the form $t \prec x$ and $y \succ t$, for some x and y; thus we can replace the binary relations with task-specific unary functions, ${}_t\prec$ and $\succ_t$, where

$$\begin{aligned} {}_t\prec(x) &\equiv t \prec x \\ \succ_t(x) &\equiv x \succ t. \end{aligned}$$

These functions serve as a local cache for the $\prec$ and $\succ$ relations, and only need to exist during the execution of their selected task. Thus the number of these functions can be limited to the number of concurrently running tasks, which results in an additional $O(p)$ space overhead for constant-sized caches, where p is the number of processors.

2. It is worth noting that any scheme for relation caching should be properly integrated with the scheme used to reclaim unreferenced tasks from the ordered lists. It would not be appropriate, for example, to reclaim a task from the ordered lists while leaving a reference to that task in a relation cache (lest a new task be created at the same location, resulting in incorrect cached information). One option is to remove the relevant entries from the relation caches when tasks are deleted from the ordered lists. Another is to consider relation-cache task references as a reason to retain tasks in the ordered list. Although a little wasteful of memory, this latter approach is easy to implement and amounts to a constant space overhead for constant-sized relation caches.

Restating the rules for determinacy checking given earlier using this notation yields the following for a task t accessing a datum, d:

- *Reads* — A read is valid if ${}_t\prec(w(d)) \wedge \succ_t(w(d))$—that is, if it satisfies condition Bern-1. If the read is deterministic, $r_f(d)$ and $r_b(d)$ may need to be updated:
  - If ${}_t\prec(r_f(d))$ then $r_f(d) := t$
  - If $\succ_t(r_b(d))$ then $r_b(d) := t$
- *Writes* — A write is valid if ${}_t\prec(r_f(d)) \wedge \succ_t(r_b(d))$—that is, if it satisfies condition Bern-2. If the write is deterministic, $w(d)$, $r_f(d)$, and $r_b(d)$ are updated to be t.

Tasks need not begin with empty caches. Parents can pass their caches to their children because of the transitivity of $\prec$ and $\succ$. Specifically, if a new task n has parents $p_1, \ldots, p_k$, it can begin with the cache for ${}_n\prec$ containing the same mappings as the cache for ${}_{p_1}\prec$ and with the cache for $\succ_n$ containing the same mappings as the cache for $\succ_{p_k}$.

It is also possible to pass a cache from a task to one of its cousins, but in this case the cache must be purged of incorrect entries. When passing caches from a task t to a task u where $t \wr u$, the property $(t \wr u) \wedge (x \succ t) \Rightarrow (x \succ u)$ allows us to preserve in $\succ_u$ all entries from $\succ_t$ that map to true. On the other hand, entries in $\succ_t$ that map to false must either be verified or summarily removed from $\succ_u$, because $(t \wr u) \wedge \neg(x \succ t) \not\Rightarrow \neg(x \succ u)$. Conversely, entries in ${}_t\prec$, that map to false can be retained in ${}_u\prec$ because $(t \wr u) \wedge \neg(t \prec x) \Rightarrow \neg(u \prec x)$, whereas entries that map to true should be discarded or rechecked.

If groups of nodes are added together (as may be the case in series–parallel graphs and producer–consumer graphs; see Section 10.2), we can employ *cache preloading* to avoid querying the ordered-list data structure about close relatives, where the close relatives of a task are its ancestors and the direct children of those ancestors. We can preload the caches because the parent knows how its children relate to each other and itself. Of course, creating all children at once—together with their caches—may increase space use for nonbinary task DAGS, because doing so may make nascent child tasks become effective tasks earlier than would otherwise be necessary.

Adding a cache raises some design questions, such as cache size and replacement policy. Different applications benefit from different cache sizes. At one extreme is the

strategy of performing no caching at all. At the other extreme is using a cache that grows dynamically, never forgetting any cached information. If cache preloading is combined with the latter approach, some programs may never need to query the ordered lists at all, but the benefits of reducing the number of ordered-list queries are balanced by the increased time and space costs of managing larger caches. Such implementation questions do not have obvious theoretical answers, but, in practice, I have found that even a small cache can dramatically reduce the number of queries to the ordered list.

Whether relation caching actually improves performance in practice depends on a number of factors, including the amount of contention and the costs of accessing a shared structure. In my original implementation of the LR-tags algorithm, I used a single lock to guard the entire ordered list. Under this simple locking scheme, adding relation caching dramatically improved performance. But when I replaced this simple locking scheme with the scheme outlined in Section 7.3.2, which specifically attempts to reduce locking overheads for order queries, I found that relation caching offered little benefit on the machines I was using for my tests. Perhaps my findings would have been different on a machine with more processors or where the costs of accessing shared memory were nonuniform—parallel scalability appears to be an area where experimentation is often the only way to determine the value of a particular approach.

10.4 Write-Restricted Data

In some cases, Bernstein's conditions are needlessly general and tighter determinacy restrictions can be used. For example, it is common for a program to read some input data representing the parameters of the parallel algorithm and then perform a parallel computation using those parameters. If the parallel component of the program only *reads* these parameters, it seems wasteful to use full-blown determinacy checking to ensure that these parameters are never corrupted by parallel code. In this section I will describe an obvious specialization for handling such cases.

A *write-restricted* object is a shared object that is subject to a stronger form of determinacy checking than we have described in previous sections. A write-restricted object may only be written to by a task that has no siblings or cousins. Both the start points and end points of the DAG are such points, but it is also possible for other tasks to satisfy

this condition (such tasks occupy positions where the task DAG narrows to being a single task wide).

It is not necessary to perform *any* read checking for write-restricted data because any task can read the data without risking interference. It is similarly unnecessary for write-restricted data to store r_f, r_b, and w, because the only tasks that could have written the data are other tasks that have no cousins, and such tasks must be ancestors of the task performing the write.

Thus, the rules for a task t accessing a write-restricted object are:

- *Reads* — Reads are always deterministic; no checking is required.
- *Writes* — A write is allowed (and is deterministic) if $\forall s \in S : s \lhd t$, where S is the set of all tasks so far created by the program.

(Notice that write-restricted data does not require that any housekeeping information be stored with the data, in contrast to the normal LR-tags method, which requires that we store $r_f(d)$, $r_b(d)$, and $w(d)$ for each datum, d.)

Enforcing the above conditions for write-restricted data is orthogonal to checking determinacy using the the LR-tags method. The write condition for write-restricted data can be enforced by keeping track of the width of the task DAG, or by some scheduler-dependent means. Thus, if all shared data is write-restricted, we can avoid maintaining $G_\swarrow$ and $G_\searrow$ at all.

As with the techniques we discussed in Section 10.1, the time and space gains from making data write-restricted only amount to a constant factor, but such constant-factor gains can, nevertheless, be useful in practice.

Write-restriction is not the only restriction we could impose to gain more efficient determinacy checking. I mention it in part because it has proven useful in practice, but also because it illustrates that determinacy checking may be implemented more specifically and more efficiently when the full generality of Bernstein's conditions is not required. Other useful possibilities, such as data that can only be written at object-creation time (and thus would not need to store r_f and r_b), or data that does not allow multiple concurrent readers, can also be developed within the LR-tags determinacy-checking framework. These variations can be useful not only because they can speed

determinacy checking, but also because they can ensure that the use of shared objects matches the programmer's intent.

10.5 Storage Management

In Section 9.1, I mentioned that we may use reference counts to reclaim the entries in $G_{\swarrow}$ and $G_{\searrow}$ that are no longer required, and thereby ensure a good space bound for determinacy checking. In this section we will look more closely at storage-management issues and discuss alternatives to reference counting.

When managing storage, there are four common approaches to deallocating storage that is no longer required: *explicitly programmed deallocation*, *no deallocation*, *reference-counted deallocation*, and *tracing garbage collection*. Explicitly programmed deallocation is not a viable strategy for managing $G_{\swarrow}$ and $G_{\searrow}$ because we cannot statically determine when entries are no longer required,[3] but the three remaining approaches are all workable solutions, each with its own advantages and disadvantages.

10.5.1 Reference Counting

Reference counting has good asymptotic complexity, but can be costly in practice due to its bookkeeping overheads. Reference counting requires that objects store reference-count information and keep that reference-count information up to date, increasing both the size of objects and the time it takes to store or release a reference to an object by a constant factor. In practice, reference counts can affect the cache behaviour of a program by increasing the number of writes it must perform.

Reference-counting techniques can also have difficulty with circularly linked structures, but this problem is not relevant to the LR-tags method because the references to entries in $G_{\swarrow}$ and $G_{\searrow}$ are unidirectional.[4]

3. In my C++ implementation of the LR-tags algorithm, I use explicitly programmed deallocation whenever possible; it is only $G_{\swarrow}$ and $G_{\searrow}$ that require a more sophisticated storage-management strategy.

4. The internal representation of ordered lists $G_{\swarrow}$ and $G_{\searrow}$ may involve a circularly linked list, but these internal pointers should never be included in the reference count for the entry.

10.5.2 Tracing Garbage Collection

Tracing garbage collection can also reclaim unused entries in $G_{\swarrow}$ and $G_{\searrow}$. The exact asymptotic behavior of garbage collection depends on the collector, but, in practice, we can expect time overheads to decrease compared to reference counting, and storage requirements to increase slightly. As we are concerned with parallel code, it is worth noting that parallel garbage collectors exist with performance that scales well on multiprocessor machines (Endo et al., 1997; Blelloch & Cheng, 1999). My implementation of the LR-tags algorithm includes support for the Boehm garbage collector (1988) to show that such collection is feasible.[5]

10.5.3 No Deallocation

For many applications, the overheads of deallocating entries from $G_{\swarrow}$ and $G_{\searrow}$ outweigh the storage saved. For example, an application that spawns 64 threads over the course of its execution will use only a few kilobytes of memory for $G_{\swarrow}$ and $G_{\searrow}$. If we must link the program against 80 kilobytes of garbage collector code (or increase the code size by 15% by using reference counting), we may reasonably wonder if the effort to reclaim such a small amount of storage is worthwhile.

More formally, the wasted space from performing no deallocation of entries from $G_{\swarrow}$ and $G_{\searrow}$ is proportional to the size s of the complete task DAG for the program's execution. If $s \in \mathrm{O}(e + v)$, where e is the peak number of effective tasks that may exist during a run of the program and v is the number of determinacy-checked data items, this "leaky" approach falls within the space bound of the technique with reference counting employed.

In my experience, it tends to be quite straightforward to analyze an algorithm and determine when this approach is acceptable. See Table 11.1 and the accompanying discussion in the next chapter.

5. The ordered-list structure does present some problems for naïve garbage collectors, as ordered lists belong to the class of data structures for which topological reachability does not equate to liveness. In fact, I used this structure as the basis for an example in Chapter 4 (Figure 4.2). Ordered lists can only be handled by a tracing garbage collector that supports *weak pointers* and *finalizers*, or provides more generalized extensions to handle "awkward" data structures. The Boehm collector provides all the necessary facilities.

10.6 Conclusion

In this chapter, we have examined several practical enhancements that can be applied to the basic LR-tags determinacy-checking technique. In practice, these optimizations can improve the performance of determinacy checking by a small but noticeable constant factor. In the next chapter, we will examine the performance of an implementation of the LR-tags algorithm that uses these enhancements appropriately.

Chapter 11

Real-World Performance of the LR-tags Technique

In the preceding chapters we have explored the LR-tags technique, including its underlying principles and properties—now we will discuss how it performs in practice. We will use parallel benchmarks to investigate its performance and discuss some of the practical issues that are revealed by benchmarking. For example, we will discover that the granularity at which determinacy checking is applied can make a significant difference to the program's performance. We will also see how the "leaky" approach to storage management discussed in Section 10.5.3 can improve time performance at the cost of added space usage.

11.1 Reasons for Benchmarking

We have already discussed the theoretical complexity of the LR-tags method (in Chapter 9), but we also need to examine how it performs in practice. If the constant factors are too high, the technique is merely a theoretical curiosity, rather than a useful tool for software developers.

We will examine the performance of several parallel algorithms, both with and without determinacy checking. Given the complex interactions found on a modern computer system, with processor caches, virtual memory, and so forth, we are not trying

to find an exact constant for the overheads, but rather to discover the extent of the performance penalty in broad terms. Also, we should regard my implementation of the LR-tags algorithm as only being a prototype to show the merits of the technique—a production version would no doubt be more carefully optimized and tied more closely to the particular development platform.

11.2 Benchmarks and Platform

The tests for my performance analysis are based on the benchmark programs provided with the CILK parallel programming environment and used by Feng and Leiserson in benchmarking their Nondeterminator determinacy checker (1997). CILK is a parallel extension of C that uses a work-stealing scheduler to achieve good time and space performance (Frigo et al., 1998). For my tests, I developed COTTON/PTHIEVES, an unremarkable parallel-programming environment that can run many CILK programs after a few superficial source-code changes (see Appendix B).[1] COTTON uses the C preprocessor to perform a translation analogous to that performed by CILK's cilk2c translator, whereas PTHIEVES is a work-stealing scheduler built on top of POSIX threads.

I ported the following programs from the CILK benchmark suite:

- mmult — A matrix-multiplication test that performs a matrix multiply without using a temporary array to hold intermediate results
- lu — An LU-decomposition test, written by Robert Blumofe
- fft — A fast Fourier transform, based on the machine-generated fftw code of Matteo Frigo and Steven G. Johnson (1997).
- heat — Simulates heat diffusion using a Jacobi-type iteration, written by Volker Strumpen
- knapsack — Solves the 0-1 knapsack problem using a branch-and-bound technique

1. Initially I hoped to extend CILK and substitute my LR-tags determinacy checker for CILK's SP-Bags algorithm, but it turned out to be easier to implement a replacement for CILK than to modify it.

Benchmark	Checked Objects	Object Size	Number of Reads	Number of Writes	Total DAG Nodes	DAG Breadth	Total $G_\prec$ Insertions	Peak $G_\prec$ Entries
mmult	49,152	2048	4,194,304	2,146,304	2,843,791	49,152	947,931	65,097
lu	16,384	2048	1,398,016	723,776	1,765,174	4096	588,392	24,011
fft	12,582,913	8	65,273,856	62,914,563	932,251	262,144	310,751	28,755
heat	1,048,592	8	1,057,743,823	109,051,979	159,589	1024	12,956	263
knapsack	34	8	18,956,584	216	9,477,871	3,159,291	3,159,291	59

Table 11.1: Benchmark properties.

All of these benchmarks spend the bulk of their time accessing determinacy-checked data. Table 11.1 shows some of their properties.

11.3 Variations

We will compare several versions of the test algorithms:

- Serial execution of the algorithm with no determinacy checking
- Serial execution using Feng and Leiserson's SP-Bags determinacy-checking method
- Serial execution using the LR-tags determinacy-checking method
- Parallel execution of the algorithm running under the COTTON/PTHIEVES environment with no determinacy checking
- Parallel execution of the algorithm running under the CILK environment with no determinacy checking
- Parallel execution using the LR-tags determinacy-checking method running under the COTTON/PTHIEVES environment

A test revealing the parallel performance of algorithms under the CILK parallel system is included solely to show that the COTTON/PTHIEVES environment provides a reasonable test bed on which to run the LR-tags determinacy checker.

In the interests of fairness, the comparisons against Feng and Leiserson's SP-Bags determinacy-checking checking method use my implementation of their algorithm rather

than their freely-available code. Had I used Feng and Leiserson's code, the results would be unfairly skewed against the SP-Bags method—in tests, using CILK's standard settings, I have found the performance of CILK's Nondeterminator to be disappointing, about one-tenth the speed of my implementation of their algorithm.[2] Also, recall that the SP-Bags algorithm cannot run in parallel, thus there is no test of the SP-Bags algorithm running in parallel.[3]

11.4 Results and Commentary

Figures 11.1–11.5 show the performance of the five benchmarks, running both serially and in parallel on a 64-processor Sun Enterprise 10000, with 28 GB of memory running *Solaris 7* (the benchmarks only use up to 16 of the 64 processors because I did not have exclusive access to the machine).[4] In each figure, graph (a) shows the performance of the programs using a linear scale; graph (b) "zooms in" on the 0-100% portion of the graph in (a), which is otherwise hard to see; and graph (c) shows the performance plotted on a logarithmic scale.

11.4.1 mmult and lu

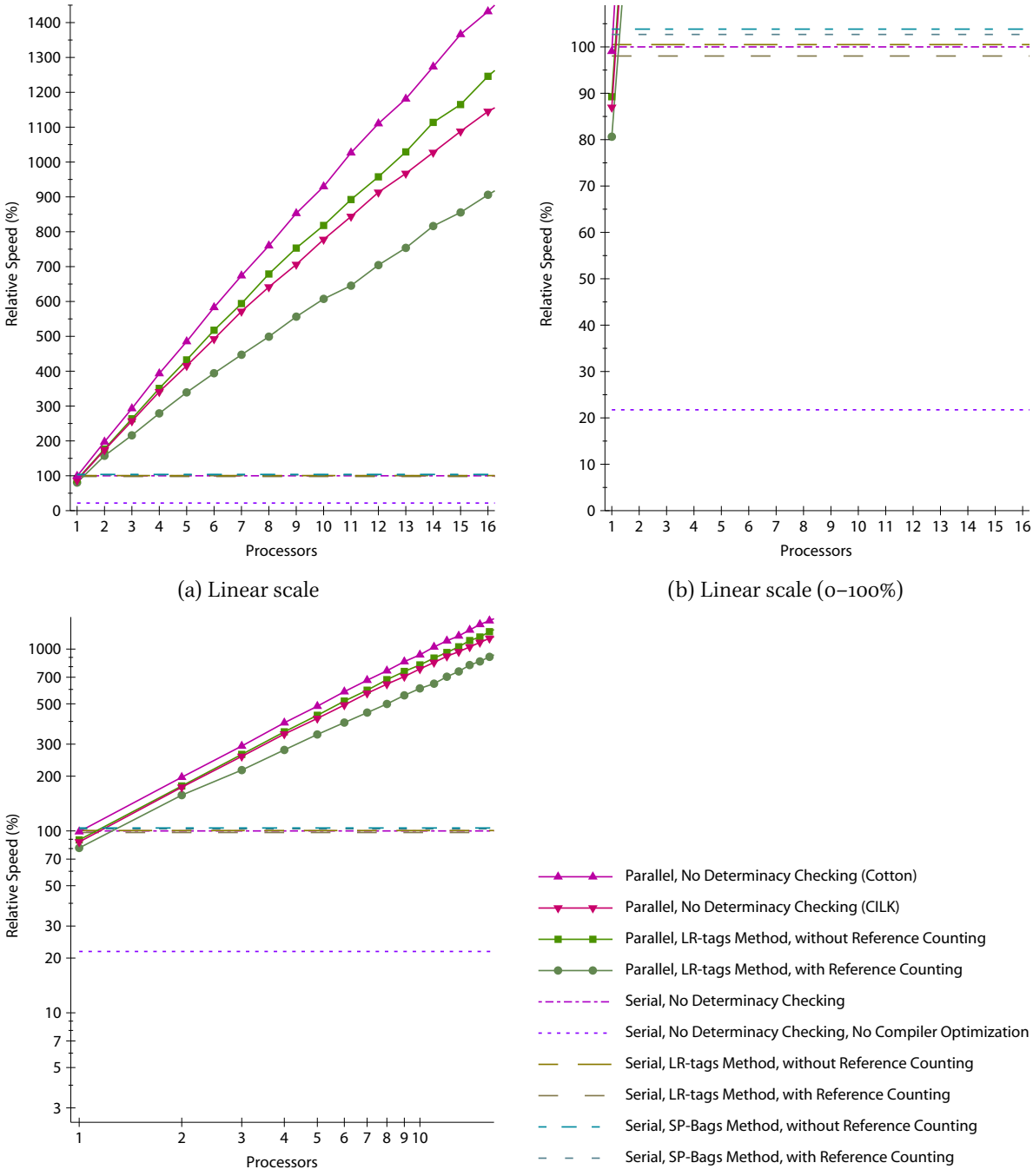
Figure 11.1 shows the performance of the mmult benchmark. This CILK benchmark uses a divide-and-conquer technique to multiply two large (2048×2048) matrices. Internally, the mmult benchmark represents the large matrix as a 128×128 matrix of 16×16 element blocks. The parallel matrix-multiplication algorithm decomposes the multiplication of the matrices into multiplications of these blocks, which are themselves multiplied using a fast serial algorithm.

This test shows that for the right program, determinacy checking can exert a very

2. These observations are not unexpected—in the conclusion of their paper on the Nondeterminator, Feng and Leiserson remark that the released version of the Nondeterminator lacks some of the performance optimizations present in the benchmarking version they describe.

3. Actually, the SP-Bags algorithm *can* be generalized to run in parallel, but the generalization is nonobvious. My first attempt at a determinacy-checking technique, discussed in Appendix F, admits Feng and Leiserson's technique as a specialization.

4. My thanks to Emerich Winkler of Sun Microsystems Canada, who provided me with access to this powerful and expensive machine.



(a) Linear scale

(b) Linear scale (0–100%)

(c) Logarithmic scale

key

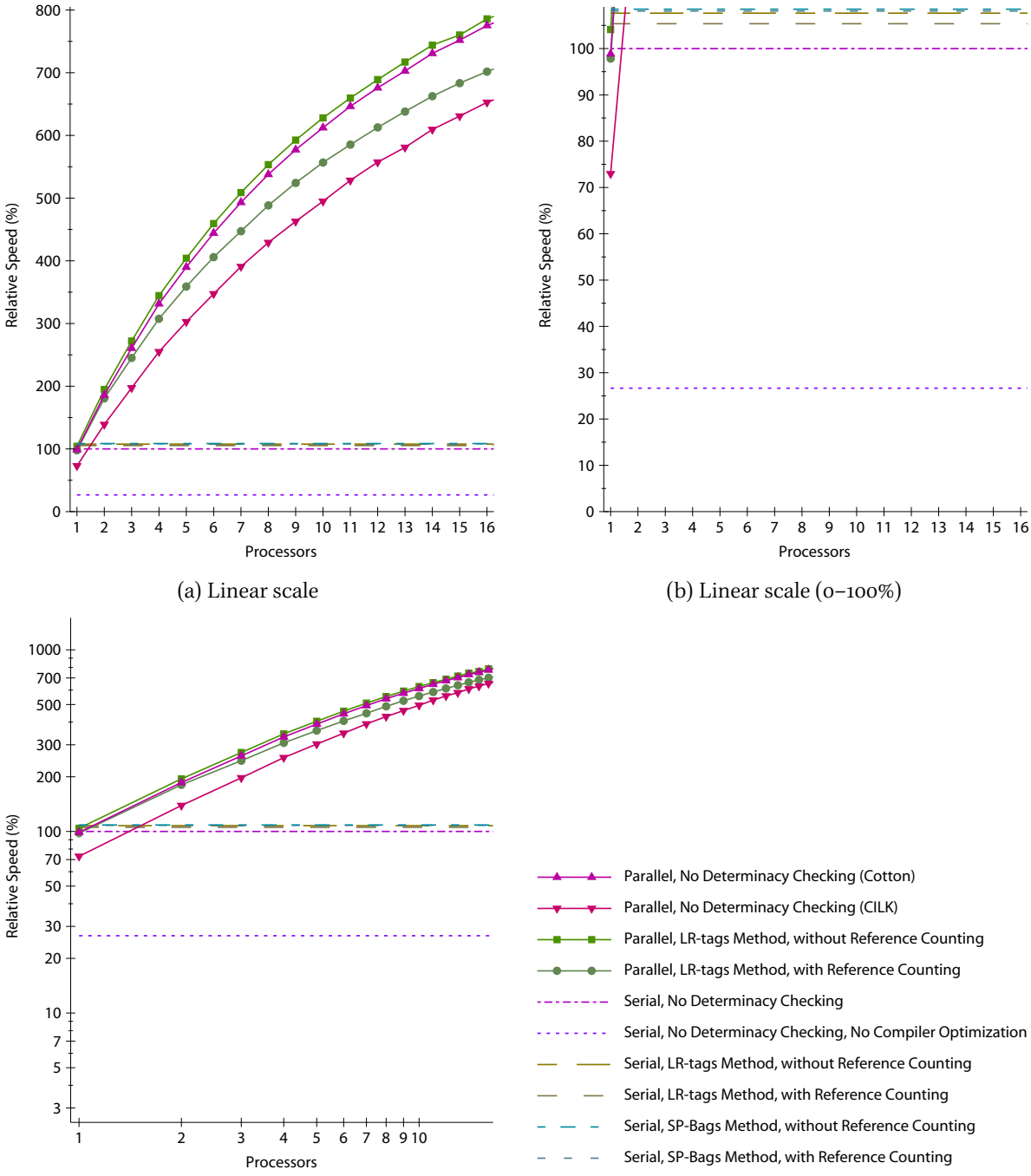
Figure 11.1: Benchmark results for mmult.

low impact. For this program, the parallel implementation of the LR-tags algorithm determinacy checker exerts an overhead of less than 5%, and both serial determinacy checkers exert so little overhead that it is virtually undetectable (in fact, it is sometimes *negative*, a matter we will discuss shortly). For serial execution, the SP-Bags and LR-tags methods turn in equivalent performance. Even on only two processors, parallel execution of this benchmark with determinacy checking enabled is faster than serial execution with no determinacy checking. Clearly, this is an application where it may be perfectly acceptable to leave determinacy checking turned on permanently if it allows us to feel more confident about the results of parallel execution.

The LU-decomposition benchmark, lu, is similar to the mmult benchmark both in terms of its algorithmic structure and its parallel structure: Both algorithms operate on matrices of 16 × 16 element blocks, and use $O(n \log n)$ tasks to decompose a matrix of n elements. As with the mmult benchmark, the overheads of determinacy checking are very low. In fact some of the test results show a *negative* overhead—for serial determinacy checking, the program runs 7.5% faster with determinacy checking turned on. This anomalous result appears to be due to the cache behaviour of the test platform. I investigated this quirk by adding two words of padding to the front of each 16 × 16 element block, mirroring the two words used to store determinacy-checking information, and found that the program ran almost 10% faster. Thus, although determinacy checking does have a cost, in this particular case (an UltraSparc machine performing LU decomposition of a 2048 × 2048 matrix) that cost is offset by altered cache behavior.

The way in which determinacy checking was applied to these two algorithms played a significant role in their good performance. I employed determinacy checking at the level of the 16 × 16 blocks rather than at the level of individual elements. This choice was a natural one because the parallel parts of the both programs work at the level of these blocks. The alternative would have been to apply determinacy checking at the level of individual matrix elements, but such an approach would incur substantially worse time and space overheads, as we will discuss below.

To use determinacy checking on large objects, such as the matrix blocks used in both mmult and lu, we need to be confident that code that claims to be accessing the interior of a determinacy-checked object does not step beyond the boundaries of that object. *Bounds checking* provides one way to ensure this kind of correctness. For mmult and lu,



(a) Linear scale

(b) Linear scale (0–100%)

(c) Logarithmic scale

key

Figure 11.2: Benchmark results for lu.

I examined the code (which was, fortunately, trivial to reason about) and also double-checked that examination using runtime bounds checking.[5] Runtime bounds checking is not used in the test runs evaluated here because we can be confident about the memory-access behaviour of the serial algorithms. Moreover, adding bounds checking to all versions of the algorithm would inflate the running time of the algorithm, making the additional cost of determinacy checking appear smaller.

Using coarse-grained determinacy checking improves both the time and space performance of the algorithms by a significant constant factor. In my implementation, each determinacy-checked object must store r_f, r_b, and w, a space overhead of less than 1% per 16 × 16 block. Had we been checking each matrix element individually, the space overhead would be an additional 150% per object. Time overheads are also reduced by this block-based strategy. The matrix multiply performed by mmult performs three determinacy checks on the blocks it passes to the serial block-multiplication algorithm (namely a read check for each of the two const operands, and a write check for the result area). The serial algorithm then performs 8192 floating-point operations to multiply the blocks. If we were checking at the level of individual numbers rather than blocks, we would have to perform 24,576 determinacy checks for each block multiply.

Finally, these two benchmarks are also interesting because they show some of the tradeoffs between time and space when it comes to the management of $G_{\swarrow}$ and $G_{\searrow}$. If v is the total number of elements in the matrices, both benchmarks create $O(v \log v)$ tasks in the course of their execution. With reference counting, the number of entries in $G_{\swarrow}$ and $G_{\searrow}$ is bounded at $O(v)$. Without reference counting, the ordered lists exceed the $O(v)$ bound, resulting in a non-constant-factor space overhead for determinacy checking; but, as we can see from the performance graphs, avoiding reference counting does achieve a small but noticeable speed improvement.

5. A *single* run with runtime bounds checking is actually sufficient to check that a block-multiply routine cannot access memory inappropriately. The block size is fixed at 16 × 16, and the structure of the algorithm is independent of the data in the block—thus the algorithm will always behave in the same way. This property of array-bounds checking, whereby checking program behavior for a particular input may allow us to draw conclusions about a whole class of inputs, can also apply to determinacy checking, as mentioned previously in Section 6.4.

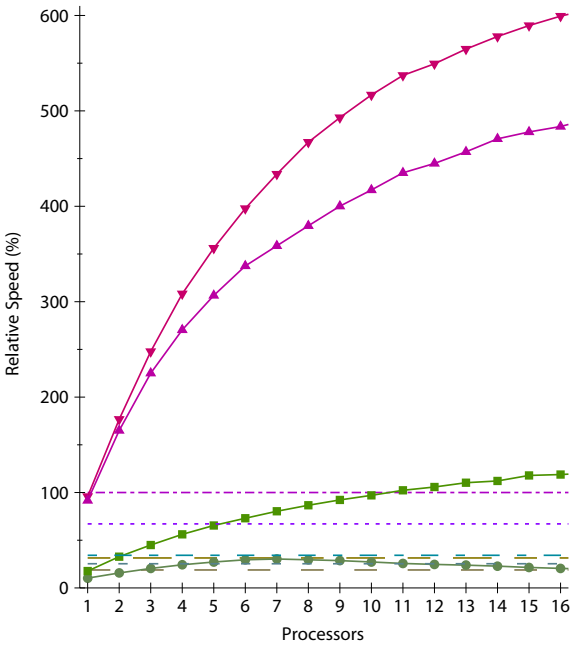
11.4.2 fft, heat, and knapsack

In the remaining benchmarks, the overheads of determinacy checking are higher then they were for mmult and lu. As we saw in Table 11.1, these benchmarks perform determinacy checking on a large number of very small objects. In the fft benchmark, the objects are single-precision complex numbers; in the heat benchmark, the bulk of the determinacy-checked objects are double-precision floating-point numbers (but three integer objects are also checked). In the knapsack benchmark, the checked objects describe items that may be placed into the knapsack: Each object is a weight/value pair.

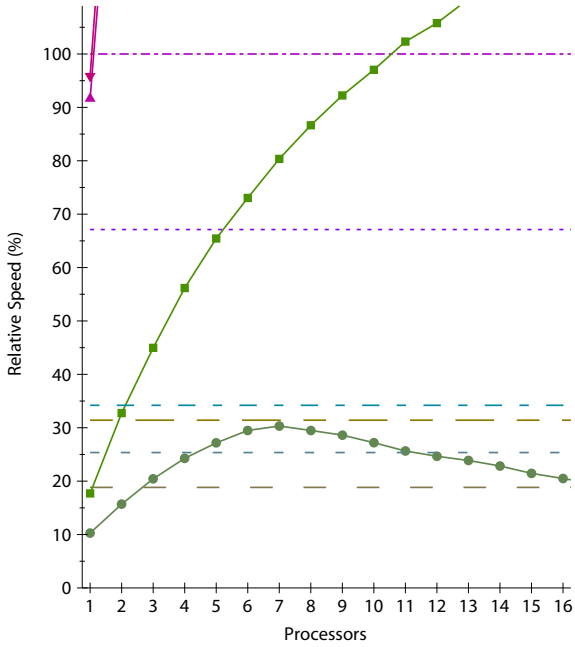
The heat benchmark performs more than a billion determinacy-checked reads and more than 100 million determinacy-checked writes, making it the benchmark that performs the most determinacy checking. Just under a third of those reads are performed on data that represent parameters of the test, defined at the time the benchmark begins. Because these parameters are never modified by parallel code, I declared them to be write-restricted (see Section 10.4), allowing them to be read with minimal overhead.

The knapsack benchmark is perhaps the cruelest benchmark of the set. It is the only benchmark where the number of threads grossly outnumbers the number of determinacy-checked objects. Whereas the other benchmarks consume a few megabytes of additional memory if reference counting is disabled, this benchmark requires a couple of hundred megabytes of memory to run if reference counting is disabled, which seems far less likely to be acceptable in a production environment.

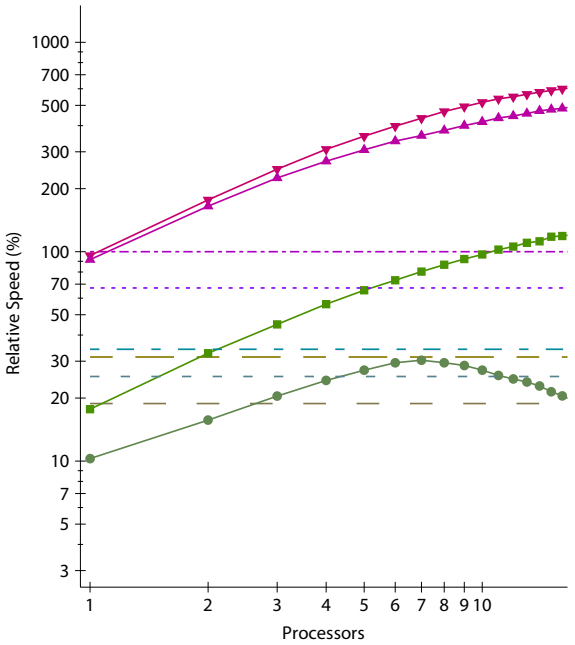
Although the knapsack benchmark has the highest determinacy-checking overhead of the group, we can take comfort in the fact that it does not actually need the full generality of our runtime determinacy checker at all. All the determinacy-checked data in the program is written during the initialization by the initial thread. If the only determinacy-checked data is write-restricted data, the bulk of our general determinacy-checking mechanisms, and their costs, are unnecessary. If we strip out the general determinacy checker and only support determinacy checking for write-restricted data, the performance of this benchmark with determinacy checking becomes virtually indistinguishable from its performance without determinacy checking. In my tests, however, I did not apply this optimization as it would render the benchmark much less useful for gaining insights into determinacy-checking performance.



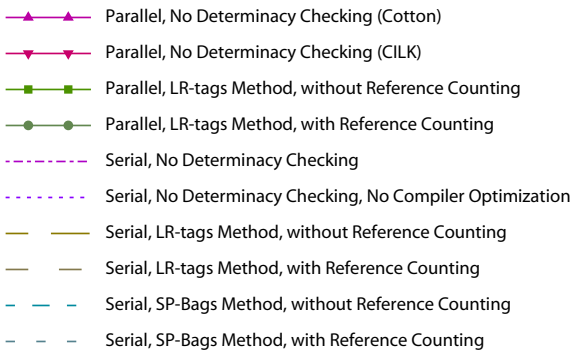
(a) Linear scale



(b) Linear scale (0–100%)

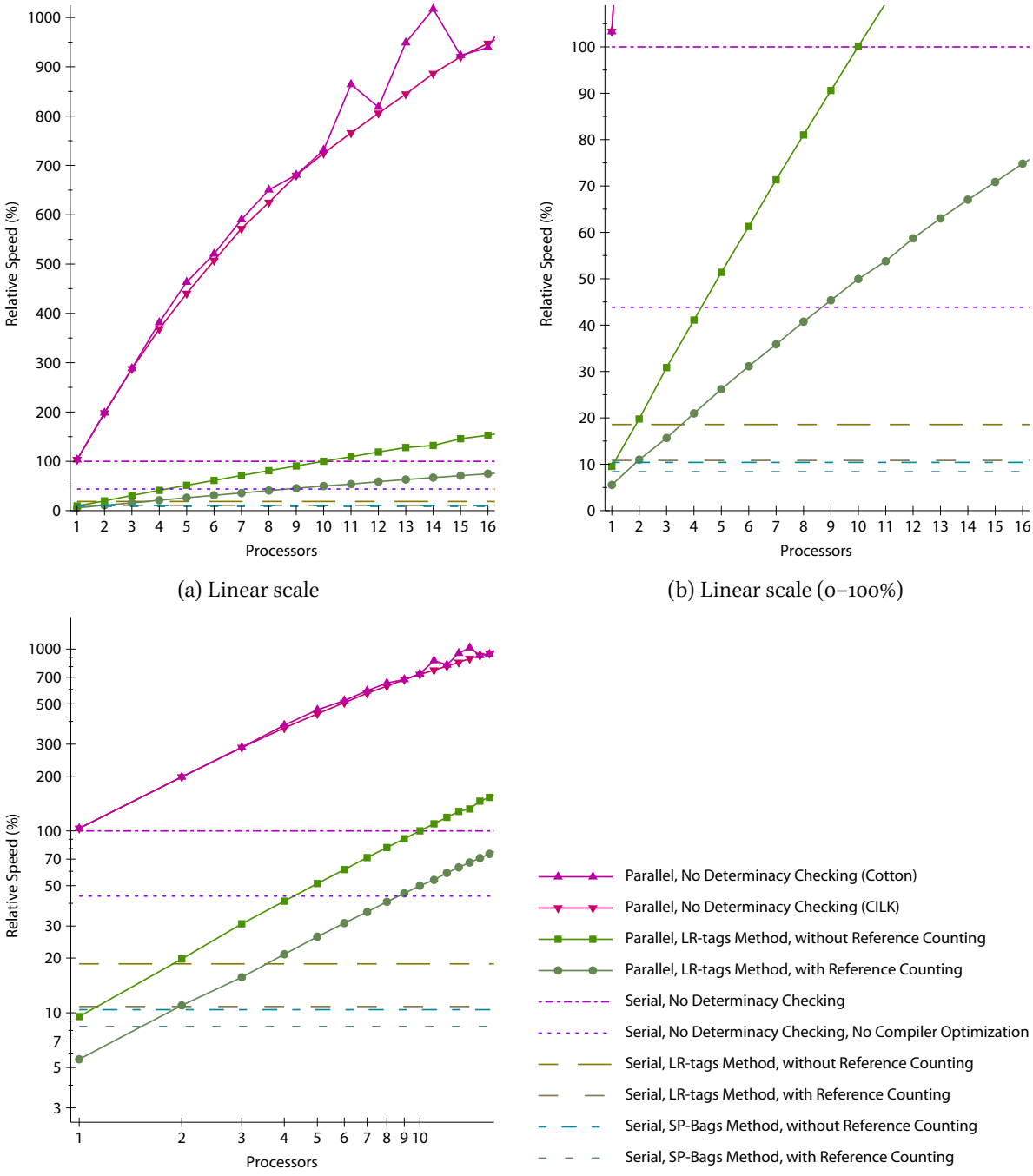


(c) Logarithmic scale



key

Figure 11.3: Benchmark results for fft.



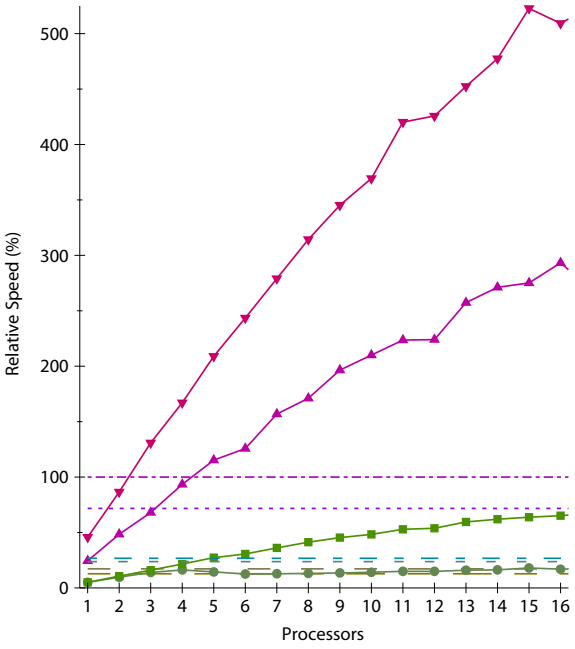
(a) Linear scale

(b) Linear scale (0–100%)

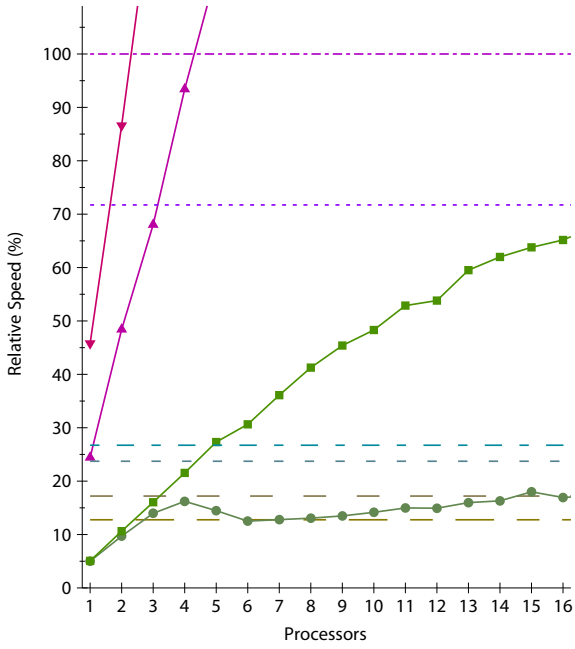
(c) Logarithmic scale

key

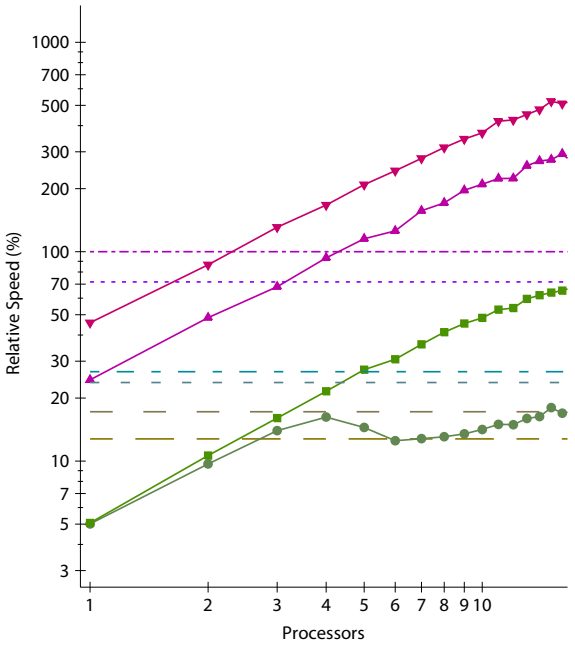
Figure 11.4: Benchmark results for heat.



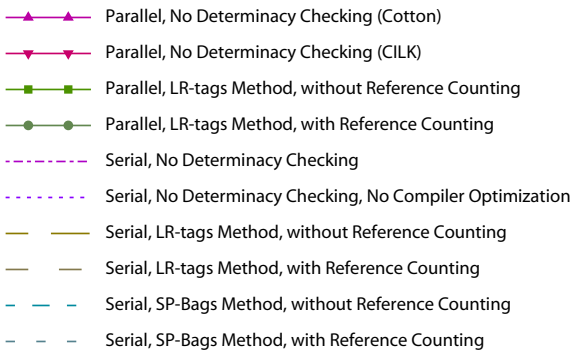
(a) Linear scale



(b) Linear scale (0–100%)



(c) Logarithmic scale



key

Figure 11.5: Benchmark results for knapsack.

11.5 Conclusion

Parallel determinacy checking can be practical, both as a debugging aid and in production code. The tests show the impact of runtime determinacy checking in both a good and a not-so-good light, revealing that the overheads of using runtime determinacy checking depend on the particular application, varying from virtually undetectable to quite noticeable. The number of shared objects, the size of those objects, and the frequency with which they are accessed influence the overheads of runtime checking. Similarly, the number of threads and the frequency with which threads are created also affects the cost that determinacy checking exacts.

Chapter 12

Conclusion

In this chapter, I summarize the results presented in this dissertation. Because the dissertation itself consists of two parts—one discussing functional arrays, the other discussing determinacy checking—this conclusion begins with a similar structure.

12.1 Functional Arrays

In Part I, I developed a functional-array data structure that offers good time and space performance in both theory and practice. From a theoretical standpoint, its asymptotic performance is an improvement over existing techniques for functional arrays. In practice, this new technique can offer better performance than competing tree-based or trailer-based array techniques.

The fat-elements method is not a panacea, however—some applications may improve their time or space performance by using an alternative array mechanism. For example, applications that do not require persistence and are able to use imperative arrays directly can avoid the overheads required by all persistent data structures. Similarly, although a given algorithm may require $O(n \log n)$ time and space when implemented using trees, and only $O(n)$ time and space using fat-element arrays, in practice, the less complex tree-based approach may offer better time or space behaviour if n is sufficiently small.

Given that no known functional-array technique can be best at everything, it is worth remembering that the fat-elements method can integrate well with other techniques. For

example, it can act as a useful adjunct to other techniques for arrays (such as update analysis). In cases where abstract interpretation can determine that an array will only be accessed single-threadedly, a raw imperative array may be used, but when this optimization cannot be applied, a fat-element array can be used. As we saw in Chapter 5, these optimizations need not be all-or-nothing; single-threaded update sequences that form part of a non-single-threaded program can be efficiently applied to a fat-element array.

Like other persistent data structures that use mutation internally, the fat-elements method poses some challenges for traditional garbage collectors. In what may be one of the first discussions of this topic, we have seen how these garbage-collection issues can be addressed.

Although it would seem that no functional-array technique is likely to provide all the answers, the fat-elements method does seem to offer improvements over previous work. In my opinion, it makes the best compromise between time and space performance. It avoids some of the unfortunate worst-case behaviour present in other techniques without making programmers pay for that benefit with heavy constant-factor overheads.

12.2 Determinacy Checking

In Part II, I presented a straightforward runtime technique that can verify that a parallel program satisfies Bernstein's conditions for determinacy. Runtime checks for Bernstein's conditions are useful because, much like array-bounds checking, statically determining whether a general parallel program satisfies Bernstein's conditions is undecidable.[1] Although runtime checks only assure us that a program runs correctly for the given (class of) input, such tests can be invaluable aids for debugging. In addition, the added protection against incorrect results these tests provide can even make the overheads of determinacy checking (which, as we have seen, can be small for some algorithms) worthwhile in production code.

The LR-tags method is orthogonal to other schemes that can enforce determinacy. It is compatible with static determinacy checking, because we can apply runtime checking only to those algorithms and data structures that cannot be statically shown to be

1. See footnote 5 on page 84 for a discussion of the undecidability of static enforcement of Bernstein's conditions.

deterministic. Similarly, it is possible to mix runtime determinacy checks of shared data structures with I-structures (Arvind et al., 1989) and intertask communication through deterministic communication channels in a single program. If some of the shared data in the program is accessed using locks, we may debug the code that uses locks with a lock-discipline checker such as ERASER (Savage et al., 1997), or a serial determinacy checker that supports locks such as ALL-SETS or BRELLY (Cheng et al., 1998). Finally, as with other determinacy-checking techniques that enforce Bernstein's conditions, the LR-tags method can form the core of a system that enforces Steele's more general determinacy conditions.

In addition, the techniques I have devised for determinacy checking are general algorithms that may prove to be applicable elsewhere. At the core of my determinacy checker is a graph representation using two ordered lists that may rapidly satisfy ancestor queries and efficiently maintain node sets against which such queries can be made (i.e., "Are all the nodes in a set ancestors of some other node?"). Similarly, the algorithm for maintaining this graph representation as new leaves are added to the graph appears to be both novel and interesting.

12.3 Future Work

The research undertaken for this dissertation has revealed several possible directions for future work. In this section, I will describe some of these possible avenues.

For the fat-elements method, one question that remains is whether the process of array splitting can be easily combined with garbage collection. It may also be fruitful to consider whether a garbage collector could be made to undo array splits if doing so would reduce space usage without adverse effects.

Some of the issues that arise in garbage collecting functional arrays apply to other persistent data structures. In general, the topic of hard-to-garbage-collect data structures appears to be rarely addressed in the published literature. Developing a framework for understanding what properties may make a data structure difficult to garbage collect, and devising a generalized mechanism for dealing with such structures, is a topic for additional research.

The LR-tags method also provides us with some opportunities for further research. The method was designed to operate on shared-memory machines, but although commodity hardware appears to favour this model, massively parallel machines tend to adopt alternate strategies. Future research could examine techniques for extending the LR-tags method to elegantly support other parallel architectures, such as cache-coherent non-uniform memory architecture, cache-only memory architecture, and distributed virtual shared memory.

Also interesting is the question of whether the LR-tags technique can be extended to efficiently represent programs whose task DAG is not an LR-graph, or programs whose task DAG is an LR-graph but where the DAG's left-to-right node ordering is non-obvious and must be calculated on the fly.

As mentioned in Section 12.2, the graph algorithms used to support determinacy checking appear to be novel and may be applicable to other domains.

Finally, much of the work in this dissertation relies on ordered lists. Although a real-time algorithm for ordered lists has been outlined by Dietz and Sleator (1987), that algorithm is complex and ill-suited to parallel use. A simple real-time algorithm for this problem—especially one that was amenable to parallelization—would strengthen many of the results presented in this dissertation.

12.4 Summary

To summarize, I have presented the following:

- A new method for providing functional arrays, the fat-elements method, which uses periodic array splitting to achieve good time and space bounds
- A proof that the fat-elements method has good time and space complexity
- Performance results that suggest that the fat-elements method has advantages over existing techniques
- Techniques for garbage collecting unused array versions from fat-elements master arrays

- Techniques for optimizing the fat-elements method
- A new method for verifying at runtime whether a program obeys Bernstein's conditions for determinacy: the LR-tags method
- Asymptotic complexity results for the LR-tags method that are better than those for any prior technique
- A new algorithm for performing ancestry queries on nodes or sets of nodes in a dynamic LR-graph
- A proof that the insertion algorithm used to add leaves to an LR-graph is correct
- Techniques for optimizing the LR-tags method
- Performance results that suggest that the LR-tags method has advantages over existing techniques

In addition to this work, there are various topics included in the appendices to this dissertation, which, although secondary to the main results, may be of interest to some readers:

- A discussion of the connections between my work on functional arrays and my work on determinacy checking
- A description of the Cotton environment for parallel programming (which was created to enable me to run Cilk benchmarks under my determinacy checker)
- A description of the Pthieves work-stealing scheduler (also created to enable me to run Cilk benchmarks)
- A description of Dietz and Sleator's (1987) ordered-list data structure
- A sketch of how Dietz and Sleator's (1987) ordered-list data structure may be accessed safely by parallel code with minimal serialization

- A description of an earlier method for parallel determinacy checking that, although less efficient than the technique presented in the body of my dissertation, may be of interest because it admits Feng and Leiserson's (1997) technique as a specialization

- An outline of some of my preliminary ideas for parallel use of functional arrays, which lead me to explore determinacy checking

Finally, although not included in this dissertation, source code for my implementations of the fat-elements and LR-tags methods is available, which may provide answers to various implementation questions not explicitly discussed here (see Appendix H for information on obtaining this material).

Ultimately, I hope that the ideas presented in this dissertation will prove to be *useful* to researchers and practitioners alike.

Appendices

Appendix A

Connecting Functional Arrays and Determinacy Checking

Part I of this dissertation introduced a new technique for implementing functional arrays; Part II introduced a new technique for determinacy checking—can there really be a connection between these two topics? Yes! In this appendix, we will examine some of those connections.

A.1 Shared Memory, Tagged Data, and Hidden Mechanisms

In Part I, we examined a technique that allowed a single shared array to be safely used non–single-threadedly, whereby a read of the array in one context did not conflict with a write to the same array location in another. In Part II, we examined a technique that allowed shared memory to be safely used in multithreaded programs, whereby a read of a memory location in one thread was prohibited from conflicting with a write to that same memory location in another thread.

Since arrays are, in essence, a data abstraction that is analogous to computer memory, we can already see some broad parallels between the work described in Parts I and II.

But the parallels go deeper. In the case of functional arrays, what appear to be distinct arrays created from a single original array may, in fact, all share the same master array. In-

side the master array, element values are tagged so that we can find which array-element values belong to which array version. In the case of determinacy checking, objects in memory are transparently instrumented with tags that allow us to know which parallel threads have read and written those objects. In both cases, the tagging mechanism itself is invisible to programmers—there is no way to access the tags themselves. Programmers deal with a simple external interface; the internal details need not concern them.

A.2 Version Stamps

As we have seen, both techniques tag data with version stamps to keep track of changes made to the data. The LR-tags method uses two sets of version stamps (a single task is represented as a version stamp in both the $G_{\swarrow}$ and $G_{\searrow}$ ordered lists), whereas the fat-elements method uses a single version stamp.

A.3 Referential Transparency versus Determinacy

There is more of a connection between these topics than a mere coincidence of implementation details: There is also a connection between their underlying purposes.

In Chapter 1, we saw that referential transparency is the raison d'être for functional arrays. Were it not for the desire on the part of functional programmers to write programs that are referentially transparent, there would be one less reason for functional arrays. Thus, Part I of this dissertation can be seen as providing an efficient means to enforce referentially transparent access to arrays, whereas Part II provides an efficient means of enforcing deterministic parallel access to shared memory. We have already seen that shared memory and a shared array are analogous—now we will see that determinacy and referential transparency are also analogous.

Referential transparency requires that a function with the same arguments must always yield the same results, whereas determinacy requires that a program with the same input must always yield the same output.[1] It follows that a referentially transpar-

1. This definition is a simplification; there are in fact a variety of definitions for determinacy, each with slightly different requirements but the same ultimate goal: reproducible behaviour.

ent program is deterministic, but not all deterministic programs are referentially transparent.[2] In the same way that referentially transparent programs are easier to reason about than their non–referentially-transparent counterparts, so, too, are deterministic programs easier to reason about than nondeterministic ones.

A.4 Conclusion

While both parts of this dissertation can be read separately, we have seen that there is actually a significant connection between these two areas. This revelation should not come as much of a surprise, because my work on functional arrays would never have lead me to the problem of parallel determinacy checking if there had not been a strong connection between the two.

2. Again, this statement is a simplification. Whether referentially transparent programs can be nondeterministic depends entirely on how you define nondeterminacy.

Appendix B

Cotton: A Parallelism Mechanism for C++

In this appendix, I describe the mechanism I developed to allow parallel code to be run easily under C++. The goal was a straightforward one: Allow the Cilk benchmarks to be used to test my determinacy-checking algorithms.

B.1 Cilk

Cilk (Frigo et al., 1998; Blumofe et al., 1996) is not a mere thread library, but is instead a powerful extension to Ansi C that adds constructs for parallelism to the language and schedules the resulting parallel code under an efficient work-stealing scheduler.

The Cilk system provides a language translator, cilk2c, to convert Cilk programs into machine-readable C code, and a runtime system that supports and schedules the translated code. Cilk is promoted as a *faithful* extension to C, because Cilk programs can be transformed into serial C programs simply by eliding the Cilk keywords from the program's source code (a task that can be performed by the C preprocessor).[1]Cilk has a well-deserved reputation as a fast, efficient, and expressive parallel language. Listing B.1 shows an example Cilk procedure that calculates Fibonacci numbers (using a

1. The C elision of a Cilk program is always a valid program in Gnu C, but may not always be a valid Ansi C program. Note also that the converse operation—adding Cilk keywords to a C program—may not result in parallel execution that is faithful to the C original, or even semantically correct; parallelization remains a thoughtful exercise.

Listing B.1: A procedure to calculate Fibonacci numbers in CILK.

```
cilk void fib (int n, int *result) {
        if (n < 2) {
                *result = n;
        } else {
                int x, y;

                spawn fib (n - 1, &x);
                spawn fib (n - 2, &y);
                sync;
                *result = x + y;
        }
}
```

parallel implementation of the classic, inefficient, algorithm).[2] We will use this example throughout the remainder of our discussion.

Programming in CILK is usually much more pleasant than programming using POSIX threads (IEEE, 1995) or similar thread libraries, but CILK is not the ideal tool for every situation. CILK has only been ported to a small number of UNIX platforms, making its availability limited. Moreover, the CILK language is closely coupled to its threading mechanism—cilk2c can only target the CILK runtime system, rather than POSIX threads or a user-defined threads package. This tight integration translates into runtime efficiency, but makes third-party adaptations and extensions to CILK more difficult. As with MODULA-3 (Nelson, 1991), ADA 95 (ISO, 1995), and JAVA (Joy et al., 2000), the integration of threads into the language ties the language's users to its implementer' thread-support choices. In addition, CILK is tied to the GNU C dialect—there is currently no CILK++ providing CILK extensions for C++.

COTTON is my solution to both the portability and extensibility problems present in CILK and the problems of handling numerous thread libraries for C and C++. COTTON is not a thread library, it is a simple, but elegant, threading interface. It aims to provide

2. Obviously, there are better algorithms for calculating Fibonacci numbers than this one, but this inefficient algorithm provides a useful example of a short parallel program. Also, note that defining fib as a procedure rather than a function was a deliberate choice—we will address implementing fib as a function shortly.

similar expressiveness to that found in CILK and to allow migration in both directions between CILK and more basic thread libraries such as POSIX threads. The design goals of COTTON were to:

- Be modular and extensible, suitable for research into scheduling and parallelism
- Be simple, not replicating the work of the CILK project and thread-library developers
- Allow efficient serial execution of COTTON programs through a trivial translation process from COTTON to C
- Allow efficient parallel execution of COTTON programs under CILK through a trivial translation process from COTTON to CILK
- Allow efficient parallel execution of programs using POSIX threads (and similar libraries) through a trivial translation process from COTTON to C++
- Use the C preprocessor to perform all translations
- Allow development in C++ as well as C

Many of these above goals reflected my needs as a researcher. I needed a simple foundation on which to examine issues relating to parallel program execution. The CILK language suggested a means to faithfully extend the C language for parallelism, provided a simple mechanism to parallelize existing code, and offered many examples of parallel programs. Had cilk2c produced human-readable C code that I could adapt to other thread libraries, I might have used CILK. But because I wanted to be able to experiment with different thread libraries, including those conforming to the POSIX threads specification (IEEE, 1995) as well as those offering the earlier C-Threads interface (Cooper & Draves, 1988), I developed COTTON.

B.1.1 The CILK and COTTON Languages

Listings B.1 and B.2 show a procedure to calculate Fibonacci numbers in CILK and COTTON, respectively. In CILK, spawn indicates a parallel procedure call in which a thread

Listing B.2: The fib procedure in COTTON.

```
PARALLEL void fib (int n, int *result) {
    PARALLEL_BLOCK
    if (n < 2) {
        *result = n;
    } else {
        int x, y;

        SPAWN (fib, n − 1, &x);
        SPAWN (fib, n − 2, &y);
        SYNC;
        *result = x + y;
    }
}
```

is spawned to run the procedure invocation and the calling code continues with the following statements while the spawned thread runs. The sync statement provides a local barrier that pauses execution until all threads spawned by the procedure have terminated. There is an implicit sync when a procedure returns—thus a CILK procedure cannot spawn a detached daemon thread that continues executing after the procedure has returned. Procedures that will be invoked with spawn must be prefixed with the keyword cilk (and, conversely, such cilk procedures must always be called using spawn).

The COTTON version of the program is very similar to the CILK program. Instead of a prefix cilk, procedures that will be invoked in parallel are given the qualifier PARALLEL and include as the first keyword inside their outermost block the keyword PARALLEL_BLOCK. Similarly, spawn is replaced with a macro SPAWN, and sync is replaced with SYNC. To translate a COTTON program into a CILK program, we use the C preprocessor to make the following substitutions:[3]

3. SPAWN takes a variable number of arguments. Macros that take a variable number of arguments were added to the C language definition (ISO, 1999) only recently, but many C preprocessor implementations, including the popular GNU C preprocessor, have allowed macros to take a variable number of arguments for some time. If we must follow earlier ANSI C standards, we must either use a set of macros that include the number of function arguments in their names, such as SPAWN0, SPAWN1, and so forth, or adopt a more verbose means of expression, such as SPAWN f BEGIN_ARGS $a_1, \ldots, a_n$ END_ARGS.

$$\begin{aligned}
\text{PARALLEL} &\mapsto \text{cilk} \\
\text{PARALLEL_BLOCK} &\mapsto \epsilon \\
\text{SPAWN}(f,a_1,\ldots,a_n) &\mapsto \text{spawn } f(a_1,\ldots,a_n) \\
\text{SYNC} &\mapsto \text{sync}
\end{aligned}$$

(In these mappings, ϵ represents an empty string.)

As we stated earlier, CILK programs may be converted to C programs that execute serially simply by deleting the keywords cilk, spawn, and sync from the program source. Thus the serial C translation is obtained by using the following macro substitutions:

$$\begin{aligned}
\text{PARALLEL} &\mapsto \epsilon \\
\text{PARALLEL_BLOCK} &\mapsto \epsilon \\
\text{SPAWN}(f,a_1,\ldots,a_n) &\mapsto f(a_1,\ldots,a_n) \\
\text{SYNC} &\mapsto \epsilon
\end{aligned}$$

The remaining case is the COTTON translation for thread libraries such as POSIX threads, C-Threads, and custom-written thread libraries. We shall detail this translation in the next section.

B.2 Implementing COTTON for Thread Libraries

To understand how the COTTON thread-library translation is implemented, we shall first look at how we would hand-parallelize our code to use a fictitious thread library with two primitives:

$$\begin{aligned}
\text{thread_fork}(*(\textit{any} \rightarrow \textit{any}), \textit{any}) &\rightarrow \textit{thread_id} \\
\text{thread_join}(\textit{thread_id}) &\rightarrow \textit{any}
\end{aligned}$$

where *any* represents an arbitrary pointer and *(any → any) represents a pointer to a function that takes an arbitrary pointer and returns an arbitrary pointer.[4] Both POSIX

4. I have tried to insulate readers from the unpleasantness of C/C++ type specifications in the text of this appendix; in C, any ≡ void * and *(*any* → *any*) ≡ *any* (*)(*any*). The C and C++ source shown in the

threads and C-Threads follow this form. The pointer value returned by the spawned thread is made available to the parent thread by thread_join.

To code fib as a parallel procedure, we must take account of the fact that whereas fib takes two arguments, the threads interface only allows for a single argument.[5] The standard solution to this problem is to allocate a record structure, known as an **argument block**, in memory to hold the function arguments. An example coding of fib under this scheme is shown in Listing B.3. We have declared two new functions, an **invocation function**, spawn_fib, that puts the arguments for fib into the argument block, and a **dispatch function**, run_fib, that pulls the arguments out of the argument block and calls fib. The procedure fib itself has only changed slightly, invoking recursion through calls to spawn_fib rather than fib, and waiting for the two child tasks to finish (the equivalent of sync) using thread_join.

If we have several functions with the same type signature as fib (that is, functions with the same argument types), we can reduce the number of support functions and structure declarations by noting that the structures for functions with identical argument types are identical. Applying this generalization yields the program shown in Listing B.4. In this version of the program, the invocation and dispatch functions have suffixes based on the argument types rather than the names of the functions they support (in this case, the suffix has changed from _fib to _i_pi, standing for an integer and a pointer to an integer argument). The argument block is also updated to reflect the new naming conventions and to store a pointer to the function that the dispatch function should call—this function pointer is provided as an argument to the invocation function.

If we limit ourselves to coding in clean, standard C, the above is about as far as we can go. But if we can use the powerful extensions present in C++, additional possibilities become available. First, we are able to replace all of the function names beginning with spawn_ with a single overloaded function, spawn. But we need not stop at a simple renaming of functions. We can collapse the myriad of functions and structure definitions needed to support spawn for different function arguments into a single structure

listings uses traditional C type specifications.

5. Some readers may realize that a functional version of fib could be called by the Posix threads library with much less trouble. For now, the additional trouble is useful, since it better represents the general case. We will discuss an efficient implementation of fib as a function shortly.

Listing B.3: The fib procedure, hand-converted to use a thread library.

```
typedef struct {
        int n;
        int *result;
} fib_args;

void run_fib (fib_args *args) {
        int n = args->n;
        int *result = args->result;

        free(args);
        fib(n, result);
}

thread_id spawn_fib (int n, int *result) {
        fib_args *args;

        args = malloc (sizeof (fib_args));
        args->n = n;
        args->result = result;
        return thread_fork (&run_fib, args);
}

void fib (int n, int *result) {
        if (n < 2) {
                *result = n;
        } else {
                int x, y;
                thread_id child1, child2;

                child1 = spawn_fib (n - 1, &x);
                child2 = spawn_fib (n - 2, &y);
                thread_join (child2);
                thread_join (child1);
                *result = x + y;
        }
}
```

Listing B.4: The fib procedure generalized for all functions with the same type signature.

```
typedef void (*f_i_pi)(int, int *);

typedef struct {
        f_i_pi fn;
        int arg1;
        int *arg2;
} closure_i_pi;

void run_i_pi (closure_i_pi *closure) {
        f_i_pi fn = closure->fn;
        int arg1 = closure->arg1;
        int *arg2 = closure->arg2;

        free (closure);
        (*fn) (arg1, arg2);
}

thread_id spawn_i_pi (f_i_pi fn, int arg1, int *arg2) {
        closure_i_pi *closure;

        closure = malloc (sizeof (closure_i_pi));
        closure->fn = fn;
        closure->arg1 = arg1;
        closure->arg2 = arg2;
        return thread_fork (&run_i_pi, closure);
}

void fib (int n, int *result) {
        if (n < 2) {
                *result = n;
        } else {
                int x, y;
                thread_id child1, child2;

                child1 = spawn_i_pi (&fib, n - 1, &x);
                child2 = spawn_i_pi (&fib, n - 2, &y);
                thread_join (child2);
                thread_join (child1);
                *result = x + y;
        }
}
```

definition and a spawn function for each supported function arity. This simplification is achieved through the use of *templates*. Listing B.5 shows an implementation using templates.

One subtlety of the template implementation is that the arguments provided to spawn do not have to be the same types as the argument types declared by the function—C++ has a variety of type-conversion behaviours, and by leaving the types unconstrained, we allow any desired type conversion to occur. We do not need to worry about whether a proliferation of spawn-template instances could be created by this added flexibility because spawn is an inline function.

At this point the hard work is done! We can implement sync by using a stack (implemented as a class *ThreadGroup*, see Listing B.6) to hold the *thread_id*s of spawned threads, and providing a method sync proceed down the stack, executing thread_join on each one.

Thus, our C-preprocessor definitions implement the following mappings:

$$
\begin{aligned}
\text{PARALLEL} &\mapsto \epsilon \\
\text{PARALLEL\_BLOCK} &\mapsto \text{ThreadGroup \_\_tg;} \\
\text{SPAWN}(f,a_1,\ldots,a_n) &\mapsto \text{\_\_tg.push(spawn(\&}f,a_1,\ldots,a_n\text{))} \\
\text{SYNC} &\mapsto \text{\_\_tg.sync()}
\end{aligned}
$$

Listing B.7 shows the Cotton fib procedure shown in Listing B.2 translated to use our new definitions.

B.2.1 Extending Cotton to Support Functions

As presented so far, Cotton only supports procedures, but fib is most naturally specified as a function. We can support functions by moving the assignment of the result value out of the invoked function (fib, in this case) and into the dispatch function. Listing B.8 shows our running example converted to use fspawn to spawn functions and assign their results to a supplied destination. The arguments to fspawn are identical to the arguments to spawn, except that an additional argument—a pointer to an area of memory for storing the function's result—is added.

Listing B.5: The fib procedure with thread support through templates.

```
template <class A, class B>
struct TwoArgClosure {
        typedef void (*F)(A,B);
        F fn;
        A arg1;
        B arg2;

        TwoArgClosure(F f, A a, B b) : fn(f), arg1(a), arg2(b) { }

        static void run (TwoArgClosure *closure) {
                F fn = closure->fn;
                A arg1 = closure->arg1;
                B arg2 = closure->arg2;

                delete closure;
                (*fn) (arg1, arg2);
        }
};

template <class A, class B, class X, class Y>
inline thread_id spawn (void (*fn)(A, B), X arg1, Y arg2) {
        TwoArgClosure<A, B> *closure;

        closure = new TwoArgClosure<A, B>(fn, arg1, arg2);
        return thread_fork (&closure->run, closure);
}

void fib (int n, int *result) {
        if (n < 2) {
                *result = n;
        } else {
                int x, y;
                thread_id child1, child2;

                child1 = spawn (&fib, n - 1, &x);
                child2 = spawn (&fib, n - 2, &y);
                thread_join (child2);
                thread_join (child1);
                *result = x + y;
        }
}
```

Listing B.6: The *ThreadGroup* task for managing thread synchronization.

```
class ThreadGroup {
    struct StackElement {
        StackElement *next;
        thread_id tid;

        StackElement(StackElement *n, thread_id t) : next(n), tid(t) { }
    };
    StackElement *threads;

public:
    ThreadGroup() : threads(0) {}

    ~ThreadGroup() {
        sync();
    }

    void sync() {
        StackElement *current = threads;
        while(current) {
            StackElement *next = current->next;
            thread_join(current->tid);
            delete current;
            current = next;
        }
        threads = 0;
    }

    void push(thread_id t) {
        threads = new StackElement(threads, t);;
    }
};
```

Listing B.7: The COTTON fib procedure translated into C++.

```
void fib (int n, int *result) {
    ThreadGroup __tg;

    if (n < 2) {
        *result = n;
    } else {
        int x, y;

        __tg.push(spawn (&fib, n − 1, &x));
        __tg.push(spawn (&fib, n − 2, &y));
        __tg.sync();
        *result = x + y;
    }
}
```

We could define another macro, FSPAWN, that is identical to spawn except for an added argument at the front to specify the location to store the function result, but such a macro looks somewhat ugly. We can, in fact, adopt a slightly more elegant syntax of LET x BECOME(f,a_1,...,a_n). A COTTON implementation of fib as a function, using these macros, is shown in Listing B.9.

For the thread-library translation, we use

$$\begin{aligned} \text{LET} &\mapsto \text{\_\_tg.push(fspawn(\&} \\ \text{BECOME}(f,a_1,\ldots,a_n) &\mapsto \text{, \&}f,a_1,\ldots,a_n)) \end{aligned}$$

For the CILK translation, the macros expand as follows:

$$\begin{aligned} \text{LET} &\mapsto \epsilon \\ \text{BECOME}(f,a_1,\ldots,a_n) &\mapsto \text{= spawn } f(a_1,\ldots,a_n)) \end{aligned}$$

The C translation is identical, except for the removal of the spawn keyword.

This translation style assumes that functions are always called from an assignment statement. When spawned function calls would be part of an expression, the expression

Listing B.8: The fib function, using fspawn.

```
template <class R, class V, class A>
struct OneArgFnClosure {
        typedef V (*F)(A);
        R *rptr;
        F fn;
        A arg;

        OneArgFnClosure(R *r, F f, A a) : rptr(r), fn(f), arg(a) { }

        static void run (OneArgFnClosure *closure) {
                R *rptr = closure->rptr;
                F fn = closure->fn;
                A arg = closure->arg;

                delete closure;
                *rptr = (*fn) (arg);
        }
};

template <class R, class V, class A, class X>
inline thread_id fspawn (R *rptr, V (*fn)(A), X arg) {
        OneArgFnClosure<R, V, A> *closure;

        closure = new OneArgFnClosure<R, V, A>(rptr, fn, arg);
        return thread_fork (&closure->run, closure);
}

int fib (int n) {
        ThreadGroup __tg;

        if (n < 2) {
                return n;
        } else {
                int x, y;
                thread_id child1, child2;

                __tg.push(fspawn (&x, &fib, n - 1));
                __tg.push(fspawn (&y, &fib, n - 2));
                __tg.sync();
                return x + y;
        }
}
```

Listing B.9: The fib function in COTTON.

```
PARALLEL int fib (int n) {
        PARALLEL_BLOCK
        if (n < 2) {
                return n;
        } else {
                int x, y;
                LET x BECOME(fib, n − 1);
                LET y BECOME(fib, n − 2);
                SYNC;
                return x + y;
        }
}
```

can usually be trivially rewritten to use spawned assignment on temporary variables, which are then used in the expression.[6]

B.3 Improving Efficiency

For most real programs, the overheads of Cotton are negligible if heap-memory allocation and deallocation is reasonably efficient. In comparisons using a null thread library (one where thread_fork simply invokes the passed function on its argument and thread_join is a no-op) I found little difference between the performance of the serial C elision of COTTON programs, hand-coded parallelization, and COTTON's parallelization. Nevertheless, there are a few efficiency criticisms that might be leveled at the COTTON implementation and we shall address those criticisms in the remainder of this section.

B.3.1 Avoiding *ThreadGroup*s

C and C++ programmers like to avoid dynamically allocating heap memory where possible, preferring the cheaper option of statically allocating memory on the stack frame. The *ThreadGroup* class must allocate memory dynamically on the heap to store *thread_id*s

6. I experimented with an alternate style, having spawned functions return a *Pending*<τ>, where τ is the return type of the passed function, but limitations in the facilities provided by C++ meant that this approach had much muddier semantics.

because it has no way to tell how many times the push method is going to be invoked. One solution is to provide an alternate version of PARALLEL_BLOCK that takes an argument specifying an upper limit on the number of threads that will be spawned. However, requiring the programmer to state the number of threads that will be spawned may represent an additional maintenance headache and an additional opportunity for program failure if the declared number of threads and the actual number of threads do not match.

An alternate solution involves recognizing a common idiom in parallel programs written for the CILK language. Often, all the spawn statements in a function are present in groups that are immediately followed by a sync statement. For this common case, we implement a variant style where the spawned procedures are placed between COBEGIN and COEND, and where calls to SPAWN are replaced with calls to FAST_SPAWN. Only spawning statements are allowed between COBEGIN and COEND—a compound statement (such as an if or for statement) is unlikely to perform as desired. Similarly, FAST_FSPAWN and LAST_FSPAWN are provided for functions.

In CILK, the last spawn before the sync is, in most respects, redundant, because the parent thread immediately waits for that spawned child—the spawn keyword is only required because of CILK's requirement that parallel procedures (those marked with cilk) always be called via spawn. To allow COTTON to optimize this case for platforms other than CILK, we provide two additional macros, LAST_SPAWN and LAST_FSPAWN, which may optionally be used for final spawn in a COBEGIN/COEND block.

We will shortly see how this variation allows us to generate efficient C++ code, but first let us examine fib written in this style. Listing B.10 shows the fib function implemented using these macros.

For the CILK translation, the macros expand as follows:

$$
\begin{aligned}
\texttt{COBEGIN} &\mapsto \epsilon \\
\texttt{COEND} &\mapsto \texttt{sync;} \\
\texttt{FAST\_SPAWN}(f,a_1,\ldots,a_n) &\mapsto \texttt{spawn } f(a_1,\ldots,a_n) \\
\texttt{LAST\_SPAWN}(f,a_1,\ldots,a_n) &\mapsto \texttt{spawn } f(a_1,\ldots,a_n) \\
\texttt{FAST\_FSPAWN}(x,f,a_1,\ldots,a_n) &\mapsto x = \texttt{spawn } f(a_1,\ldots,a_n) \\
\texttt{LAST\_FSPAWN}(x,f,a_1,\ldots,a_n) &\mapsto x = \texttt{spawn } f(a_1,\ldots,a_n)
\end{aligned}
$$

Listing B.10: An optimization-ready fib function in COTTON.

```
PARALLEL int fib (int n) {
    if (n < 2) {
        return n;
    } else {
        int x, y;
        COBEGIN
            FAST_FSPAWN(x, fib, n − 1);
            LAST_FSPAWN(x, fib, n − 2);
        COEND
        return x + y;
    }
}
```

Notice that for CILK, FAST_SPAWN and LAST_SPAWN are identical to each other and to SPAWN. The FAST_FSPAWN and LAST_FSPAWN macros are provided to support functions. These constructs are perhaps a little uglier than the LET . . . BECOME construct introduced earlier, but this form is necessary because the destination for the function result is used in several places in the macro thread-library translation.

For the COTTON thread-library translation, we avoid using a *ThreadGroup* by declaring an *ActiveThread* object to hold the *thread_id* of each newly spawned thread. The important part of the *ActiveThread* class is its destructor, which waits for the thread to finish. If several *ActiveThread* objects are created in a block, when the block is exited, the program will wait for all the threads managed by the *ActiveThread* objects to terminate—equivalent to a sync operation.

The *ActiveThread* class is parameterized with a type, τ. *ActiveThread*<τ> stores information about a thread that is running a function that returns an object of type τ (procedures use *ActiveThread*<*void*>). This type parameter is only used in certain specializations of the *ActiveThread* class, which we will discuss in Section B.3.4.

Listing B.12 shows how we could adapt our translation of the COTTON fib function to use the *ActiveThread* class in the translation of the code shown in Listing B.10.

Notice that FAST_SPAWN translates to a declaration. In C++, a declaration must associate an identifier with the object declared, even if the declared object will never

Listing B.11: The *ActiveThread* class.

```
template <class R>
struct ActiveThread {
        thread_id active;

        ActiveThread(R* x, thread_id tid) : active(tid) { } // called by functions

        ActiveThread(thread_id tid) : active(tid) { } // called by procedures

        ~ActiveThread() {
                thread_join(active);
        }
};
```

again be explicitly referenced. To avoid saddling the programmer with the chore of providing a suitable identifier, we rely instead on a preprocessor trick that allows us to generate an identifier of the form __fs_*line*, where *line* is the current source line. For this technique to work, we require that every occurrence of FAST_SPAWN must be placed on a separate line. The translation rules for the thread-library translation are as follows:

$$
\begin{aligned}
\text{COBEGIN} &\mapsto \{ \\
\text{COEND} &\mapsto \} \\
\text{FAST\_SPAWN}(f,a_1,\ldots,a_n) &\mapsto \textit{ActiveThread}\langle\textit{void}\rangle \text{ \_\_thr\_}\textit{line}(\text{spawn}(\&f,a_1,\ldots,a_n)) \\
\text{LAST\_SPAWN}(f,a_1,\ldots,a_n) &\mapsto f(a_1,\ldots,a_n) \\
\text{FAST\_FSPAWN}(x,f,a_1,\ldots,a_n) &\mapsto \textit{ActiveThread}\langle\text{typeof}(x)\rangle \\
&\qquad \text{\_\_thr\_}\textit{line}(\&x, \text{fspawn}(\&x,f,a_1,\ldots,a_n)) \\
\text{LAST\_FSPAWN}(x,f,a_1,\ldots,a_n) &\mapsto x = f(a_1,\ldots,a_n)
\end{aligned}
$$

Notice that this code uses typeof, a GNU C++ extension.[7] It is actually possible to use tricks involving sizeof to achieve the same ends, but those tricks are ugly and would obfuscate matters considerably.

7. Why typeof (or a mechanism of similar power) was excluded from the ISO C++ standard is a mystery to me.

Listing B.12: The fib function, using the ActiveThread class.

```
int fib (int n) {
        if (n < 2) {
                return n;
        } else {
                int x, y;
                {
                        ActiveThread<int> __thr(fspawn (&x, &fib, n - 1));
                        y = fib(n - 2);
                }
                return x + y;
        }
}
```

B.3.2 Avoiding Dynamic Allocation in Invocation and Dispatch

*ThreadGroup*s are not the only source of dynamic memory allocation. The invocation and dispatch functions also use new and delete to allocate argument blocks on the heap. But, when the COBEGIN/COEND construct is used, it is safe to allocate the argument blocks in the stack frame. Listing B.13 shows a revised version of the code that avoids any heap memory allocation.

This code also uses typeof to support the DECLARE macro, which allows us to declare a variable without needing to (redundantly) specify its exact type. The DECLARE macro is expanded as

$$\text{DECLARE}(v,e) \quad \mapsto \quad \textbf{typeof}(e)\ v = e$$

For example, "DECLARE(x, strlen(foo))" expands as "typeof(strlen(foo)) x = strlen(foo)", which is semantically equivalent to "*size_t* x = strlen(foo)". As with the earlier use of typeof, DECLARE can be implemented in pure ISO C++ using tricks involving sizeof, but the details are truly disgusting.[8]

8. Okay, if you *must* know, you can find the amount of space required using sizeof and then allocate that much memory on the stack frame (using a *char* array with additional magic to ensure it is properly aligned) and then call a function that uses placement new to install the new object. You can even use similar tricks to ensure that the destructor is called.

Listing B.13: The fib function, with efficient thread support for all single-argument functions.

```
template <class R, class V, class A>
struct FastOneArgFnClosure {
        typedef V (*F)(A);
        R *rptr;
        F fn;
        A arg;

        FastOneArgFnClosure(R *r, F f, A a) : rptr(r), fn(f), arg(a) { }

        static void run (FastOneArgFnClosure *closure) {
                *closure->rptr = (*closure->fn) (closure->arg);
        }

        thread_id invoke () {
                return thread_fork(&run, this);
        }
};

template <class R, class V, class A, class X>
inline FastOneArgFnClosure<R, V, A> fclosure(R *rptr, V (*fn)(A), X arg) {
        return FastOneArgFnClosure<R, V, A>(rptr, fn, arg);
}

#define DECLARE(var, value) typeof(value) var = value

int fib (int n) {
        if (n < 2) {
                return n;
        } else {
                int x, y;
                {
                        DECLARE(__clo, fclosure(&x, &fib, n - 1));
                        ActiveThread<int> __thr( &x, __clo.invoke());

                        y = fib(n - 2);
                }
                return x + y;
        }
}
```

With this additional optimization, the COTTON translation rules for FAST_SPAWN and FAST_FSPAWN become

FAST_SPAWN(f,a_1,...,a_n) $\mapsto$ DECLARE(__clo_*line*, closure(&f,a_1,...,a_n))
ActiveThread<*void*> __thr_*line*(__clo_*line*.invoke())

FAST_FSPAWN(x,f,a_1,...,a_n) $\mapsto$ DECLARE(__clo_*line*, fclosure(&x, &f,a_1,...,a_n))
ActiveThread<typeof(x)>
__thr_*line*(&x, __clo_*line*.invoke())

B.3.3 Creating Fewer Template Instances

Another potential criticism of the thread-library translation for COTTON is that a myriad of dispatch functions can be created if SPAWN is invoked for many different functions with distinct argument types. In COTTON, invocation functions are inlined, but dispatch functions are not, so a separate dispatch function will be created for each combination of argument types, even though the generated code for many of the dispatch functions is likely to be identical (e.g., on many architectures, all pointers are passed to functions identically, yet from COTTON's perspective they are different argument types and demand different dispatch functions).

C++'s template-specialization mechanisms provide a means to avoid this proliferation. C++ allows templates to provide specialized implementations for particular types, or classes of types that are chosen over the general implementation. We can use this technique to provide **structural-surrogate** type substitution, where we substitute a type with a surrogate type that has identical size and calling conventions. For example, on most machines all kinds of pointers can be represented by a single pointer type, and, similarly, on many machines, a machine word can represent an integer, an unsigned integer, or a pointer. From a low-level perspective, anything that is represented as a single machine word is passed into and out of functions in the same way.

For every type τ, we provide a (potentially machine-dependent) structural surrogate, *StructualTraits*<τ>::*Surrogate*. By default (i.e., unless overridden by a template specialization), the surrogate type is identical to the provided type (i.e., *StructualTraits*<τ>::*Surrogate* $\equiv$ τ). Known surrogates are detailed through specializations—Listing B.14 shows some of the possibilities.

Listing B.14: A template defining structural surrogates.

```
template <class T>
struct StructualTraits {
        typedef T Surrogate;
};

template <class T>
struct StructualTraits<T *> {
        typedef void * Surrogate;
};

template <>
struct StructualTraits<int> {
        typedef void * Surrogate;
};
```

B.3.4 Eliminating the Argument Block

Seasoned users of Posix threads may have been feeling that the fib example needlessly allocates a memory block—on many architectures, we can pass an integer argument and return an integer result without needing to resort to allocating an argument block (relying instead on the use of casts to convert between pointers and integers, and thereby passing the integer into and out of the fib function directly).

As in the previous section, we can use template specialization as a route to optimization. In this case, we map single-argument functions (whose argument and return types are generic pointers (i.e., *void **), or types whose structural surrogate is a generic pointer), onto the basic thread primitives, without the need to allocate an argument block.

But supporting such functions requires a departure from our previous implementation of functions. Usually, it is the child thread's responsibility to write the result value into a space provided by the parent thread, but for this special case, we would prefer the child to return its result via the return value of thread_join (usually, the return value of thread_join is unused). To implement these alternate return semantics, we use the template specialization *ActiveThread<void *>*, shown in Listing B.15.

Listing B.15: A specialization of *ActiveThread* for functions that return integers.

```
template <> // specialization
struct ActiveThread<int> {
        thread_id tid;
        int* rptr;

        ActiveThread (int* r, thread_id t) : tid(t), rptr(r) { }

        ~ActiveThread () {
                *rptr = (int) thread_join(tid);
        }
};

template <class V, class A>
struct FastOneArgIntFnClosure {
        typedef V (*F)(A);
        F fn;
        A arg;

        FastOneArgIntFnClosure(F f, A a) : fn(f), arg(a) { }

        thread_id invoke () {
                return thread_fork(fn, arg);
        }
};

template <class V, class A, class X> // specialization
inline FastOneArgIntFnClosure<V, A> fclosure(int *rptr, V (*fn)(A), X arg) {
        return FastOneArgIntFnClosure<V, A>(fn, arg);
}
```

Listing B.16: A hand-coded implementation of fib using a thread library.

```
int fib (int n) {
        if (n < 2) {
                return n;
        } else {
                int x, y;
                thread_id tid;
                tid = thread_fork(&fib, (void *) (n − 1));
                y = fib(n−2);
                x = (int) thread_join(tid);
                return x + y;
        }
}
```

B.3.5 Applying All Three Optimizations

When all three optimizations are applied, an good optimizing C++ compiler will produce output for the program given in Listing B.10 that is exactly equivalent to the code produced for the hand coded C function shown in Listing B.16. Thus these optimizations allow COTTON to produce object code that is exactly equivalent to carefully hand-coded solutions.

B.4 Limitations

COTTON provides a high level abstraction, but it caters to the lowest common denominator of its underlying thread packages. It supports the intersection of functionality between C-Threads, POSIX threads and CILK. For example, whereas CILK and POSIX threads each provide facilities for thread cancellation and thereby make applications like nondeterministic search straightforward to implement, C-Threads does not natively support cancellation, so COTTON does not provide it. Similarly, CILK provides a facility, known as *inlets*, that allow a the result of a spawned function to be asynchronously received and acted upon in the calling procedure; no analogous facility exists in C-Threads or POSIX threads—in fact, the serial elision of a CILK program with inlets is not even valid ANSI C.

These features could conceivably be addressed by COTTON, but there comes a point where the original design simplicity begins to be obscured. Pushing COTTON to support functions was one step away from its simple foundations, adjusting it to provide optimal code was another step, and pushing it still further to support cancellation, mutexes, inlets, thread-stack sizes, daemon threads, or other features could each be seen as a step too far.

B.5 Conclusion

COTTON, in combination with the PTHIEVES work-stealing scheduler, provides the parallel platform on which the LR-tags method was tested. As the performance results in Chapter 11 show, COTTON/PTHIEVES can sometimes outperform CILK, which came as an unexpected surprise. Not too shabby for a C-preprocessor–based C++ hack.

Appendix C

The Pthieves Work-Stealing Scheduler

This appendix provides an overview of Pthieves, a lightweight work-stealing–scheduler front end. Pthieves does not concern itself with the low-level details of task scheduling, instead acting as an intermediary between Posix threads and programs that require fine-grained parallelism. Although the work-stealing algorithm in Pthieves is similar to that in the Cilk system (which has a provably memory-efficient scheduler), Pthieves does not claim to be memory efficient in theory.[1] In practice, however, for the applications I have tested, Pthieves appears to work reasonably well.

C.1 Not Another Scheduler!

I never expected (or intended) to write my own work-stealing scheduler. When I first wrote a prototype of my determinacy-checking system, I wrote the code in Java. Java seemed ideal because it contained built-in constructs for parallelism and provides its own scheduler. Sadly, the overheads of Java's virtual machine and the inefficiencies of its synchronization mechanisms muddied the waters enough for me to decide to re-implement my work in an "efficient" language with excellent parallel scalability.[2] So I

1. I also make no claim that Pthieves is provably inefficient.

2. Others have not been so quick to give up on Java. For example, Doug Lea (2000) has developed an efficient work-stealing scheduler written in pure Java.

looked to Cilk (Frigo et al., 1998; Blumofe et al., 1996; Blumofe & Leiserson, 1999), which seemed to fit the bill admirably. Alas, although Cilk is powerful (and included excellent benchmarks for would-be authors of determinacy checkers), its codebase did not appear to be written with third-party modification in mind.

Because Cilk was not amenable to modification, I developed the Cotton parallelism mechanism to allow me to borrow benchmarks from Cilk and run them under the Posix threads environment (or the simpler C-threads library)—see Appendix B. Cotton worked wonderfully, but the Solaris and Linux implementations of Posix threads did not, and neither did the C-Threads threads library on NextStep. These thread libraries simply were not designed for programs that used the kind of fine-grained parallelism present in the Cilk benchmarks.

I began looking for alternative schedulers. My officemate, David Simpson, was working on a work-stealing scheduler (Simpson & Burton, 1999), but it was intended for scheduling a parallel functional programming language and his prototype implementation was written in Java, the very language I had fled. I considered whether I could use Cilk's powerful and efficient work-stealing scheduler without using the rest of the Cilk system, but Cilk's scheduler is tightly integrated with the cilk2c translation mechanism, which provides a special version of each spawnable function that uses call/cc (Reynolds, 1990; Danvy & Filinski, 1972) rather than the more usual C-calling conventions.

So, after considering my options, I decided to try my hand at writing my own work-stealing scheduler. Interestingly, at around the same time, several other researchers also implemented user-level thread libraries using work-stealing schedulers, including *Hood* (Blumofe & Papadopoulos, 1998), *StackThreads* (Taura et al., 1999), and *Pthreads for Dynamic and Irregular Parallelism* (Narlikar & Blelloch, 1998).

All of the user-level thread libraries listed above were developed after I created Cotton, and some of them were created after I had written Pthieves. Yet, even if these schemes had been developed earlier, they still might not have dissuaded me from developing my own user-level scheduler. None of the above schemes support the simple thread_fork/thread_join api that Cotton expects, and all of them have subtle practical annoyances that, although not insurmountable, would have created more work for me in the long run. Hood, for example, disallows many C-standard library calls under Linux and encodes parallel functions in the form of a C++ class, increasing the porting work

Listing C.1: A function to calculate Fibonacci numbers in CILK.

```
cilk int fib (int n) {
        if (n < 2) {
                return n;
        } else {
                int x, y;

                x = spawn fib (n − 1);
                y = spawn fib (n − 2);
                sync;
                return x + y;
        }
}
```

necessary for the CILK benchmarks I wanted to run. Similarly, StackThreads requires compiler changes (which were only provided for an obsolete version of gcc), and Narlikar and Blelloch's PTHREADS implementation is a modification of the proprietary Solaris PTHREADS library and is, therefore, not available in source form. Finally, all of the above techniques are low-level, making porting them difficult (initially, I was doing my development work under NEXTSTEP, an unsupported platform for all of these tools).

C.2 The PTHIEVES Interface

PTHIEVES provides two functions to user-level code:

$$\begin{aligned} \text{thread\_fork}(*(\textit{any} \longrightarrow \textit{any}), \textit{any}) &\longrightarrow \textit{thread\_id} \\ \text{thread\_join}(\textit{thread\_id}) &\longrightarrow \textit{any}. \end{aligned}$$

This interface is similar to the interface provided by both POSIX threads and C-threads, except that no provision is made for programs to use locks or other synchronization mechanisms—instead, PTHIEVES assumes that the program follows a nested-parallelism model. This requirement means that, in a given task, calls to thread_join should follow the reverse order of calls to thread_fork (the *ThreadGroup* class in Listing B.6 provides facilities to ensure the necessary ordering rules are followed). Listings C.1 and C.2 show

Listing C.2: The fib function using PTHIEVES.

```
int fib (int n) {
        if (n < 2) {
                return n;
        } else {
                int x, y;
                thread_id calc_y, calc_x;
                calc_y = thread_fork(&fib, (void *) (n - 2));
                calc_x = thread_fork(&fib, (void *) (n - 1));
                x = (int) thread_join(calc_x);
                y = (int) thread_join(calc_y);
                return x + y;
        }
}
```

Listing C.3: CILK-like serial semantics for thread_fork and thread_join.

```
typedef void * thread_id;

thread_id thread_fork(void * (fn)(void *), void *arg) {
        return (*fn)(arg);
}

void * thread_join(thread_id tid) {
        return tid;
}
```

a function to calculate Fibonacci numbers written in CILK and ordinary C using the PTHIEVES thread library, respectively.

The semantics of thread_fork and thread_join under PTHIEVES on a uniprocessor machine are slightly different from the semantics of spawn and sync in the serial elision of a CILK program. Listing C.3 shows the semantics that CILK users might expect, whereas the definitions for thread_fork and thread_join shown in Listing C.4 are the actual definitions required to execute the code in exactly the same way that PTHIEVES does on a uniprocessor machine. In essence, CILK follows the left branch of the tree first, whereas COTTON follows the right branch (or, put another way, PTHIEVES pushes function calls

Listing C.4: Actual serial semantics for thread_fork and thread_join in Pthieves.

```
typedef struct thread_id {
        void * (fn)(void *);
        void *arg
};

thread_id thread_fork(void * (fn)(void *), void *arg) {
        thread_id tid = { fn, arg };
        return tid;
}

void * thread_join(thread_id tid) {
        return (*tid.fn)(tid.arg);
}
```

onto its work stack, whereas Cilk pushes continuations). For many algorithms, the difference is either irrelevant, or, as in our fib example, trivial to account for by re-ordering calls to thread_fork to provide the same serial execution order as Cilk.[3]

In the discussion that follows, we will assume that code written for Pthieves is transformed as necessary to execute on a uniprocessor using the same serial execution order as Cilk.

C.3 How Pthieves' Scheduler Differs from Cilk's Scheduler

Cilk's scheduler served as the inspiration for Pthieves' scheduler—both are based on an efficient-work stealing algorithm (Blumofe & Leiserson, 1999), and both use the threading facilities provided by the operating system to provide a virtual-processor abstraction on which tasks are scheduled (in both cases, the number of active virtual processors should be the same as the number of physical processors present in the machine). But

3. The only exception is Cilk code that performs a spawn inside a loop and then performs a sync outside the loop. To achieve the same execution pattern as Cilk, the loop must be rewritten as a recursive function, with each recursive step being invoked with thread_fork.

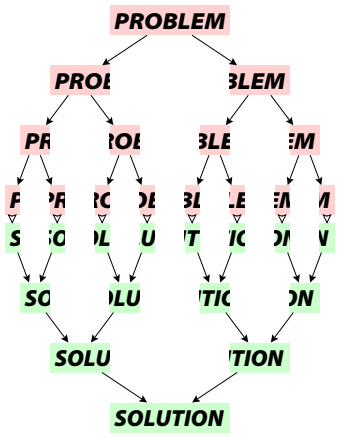


Figure C.1: Divide-and-conquer algorithms use nested parallelism.

PTHIEVES compromises the efficient work-stealing algorithm to the extent necessary to be portable and easy to implement, whereas CILK does not. In this section, I will outline the essential differences between the schedulers provided by PTHIEVES and CILK.

To understand how PTHIEVES operates, we will examine a generic parallelizable divide-and-conquer algorithm, shown in Figure C.1. Figure C.2 shows how the serial execution of such an algorithm uses memory—in this example, we can see that there are never more than four function-activation records on the stack (the figure uses a rather whimsical representation of the memory state of each activation, representing it with a randomly chosen unique bar graph and abstract shape). Careless parallel execution (a breadth-first strategy), on the other hand, could cause up to fifteen activation records in existence concurrently.

Figure C.3 shows how work is stolen in CILK. When work is stolen in CILK, the entire state of the topmost spawned task is stolen (thanks to special continuation-storing operations added to the code by the cilk2c translator). Unfortunately, this operation *cannot* be simulated in a simple user-level thread library such as PTHIEVES without resorting to machine-dependent tricks (because it's difficult to know how much of the processor stack to copy, and the stack may contain pointers that need to be adjusted for their new locations).[4]

4. StackThreads avoids the stack-copying problem by changing function-call behaviour to eliminate the expectation that activation records on the stack are contiguous. PTHIEVES does not have that luxury, however.

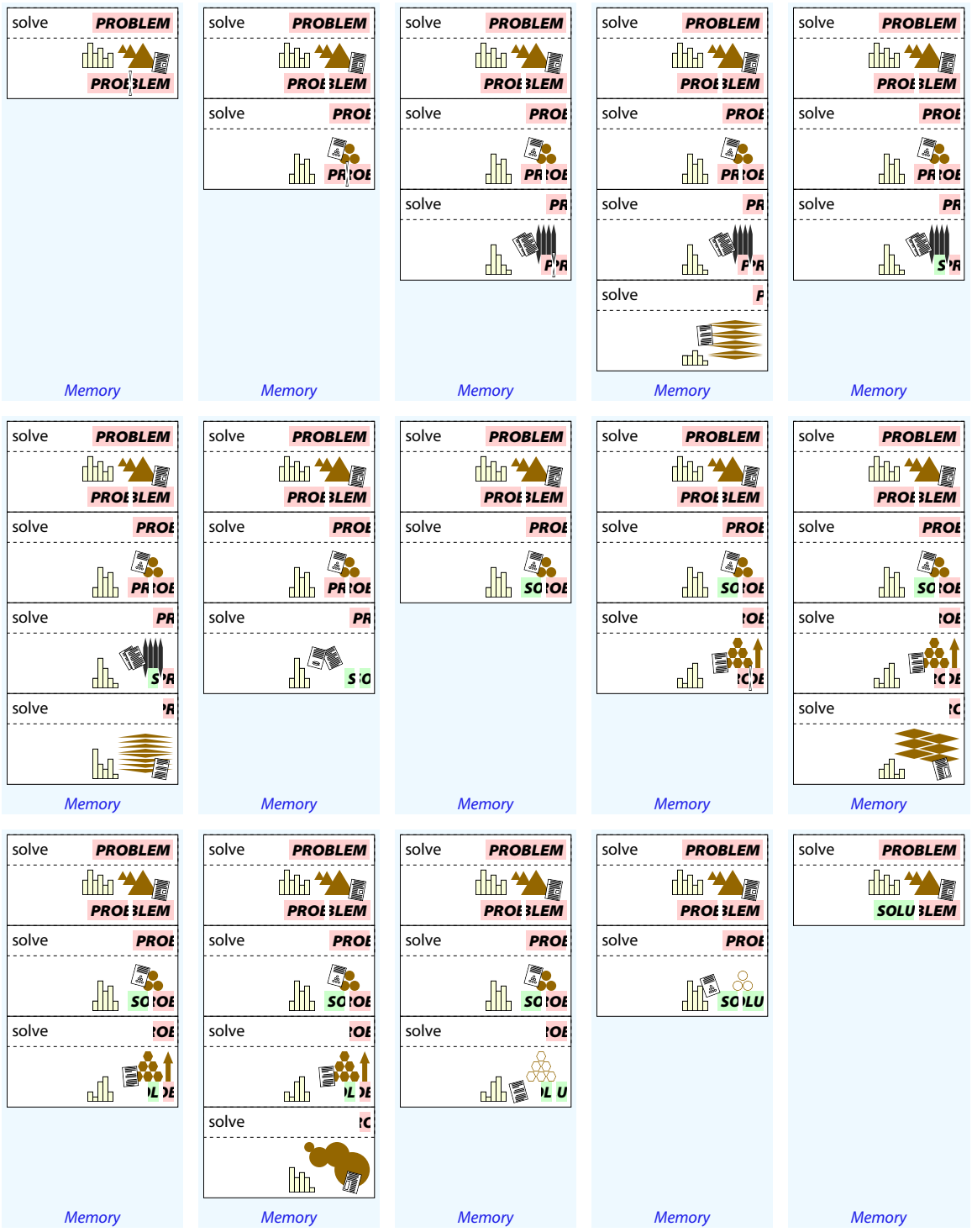


Figure C.2: Serial execution of a divide-and-conquer algorithm. (*continued over...*)

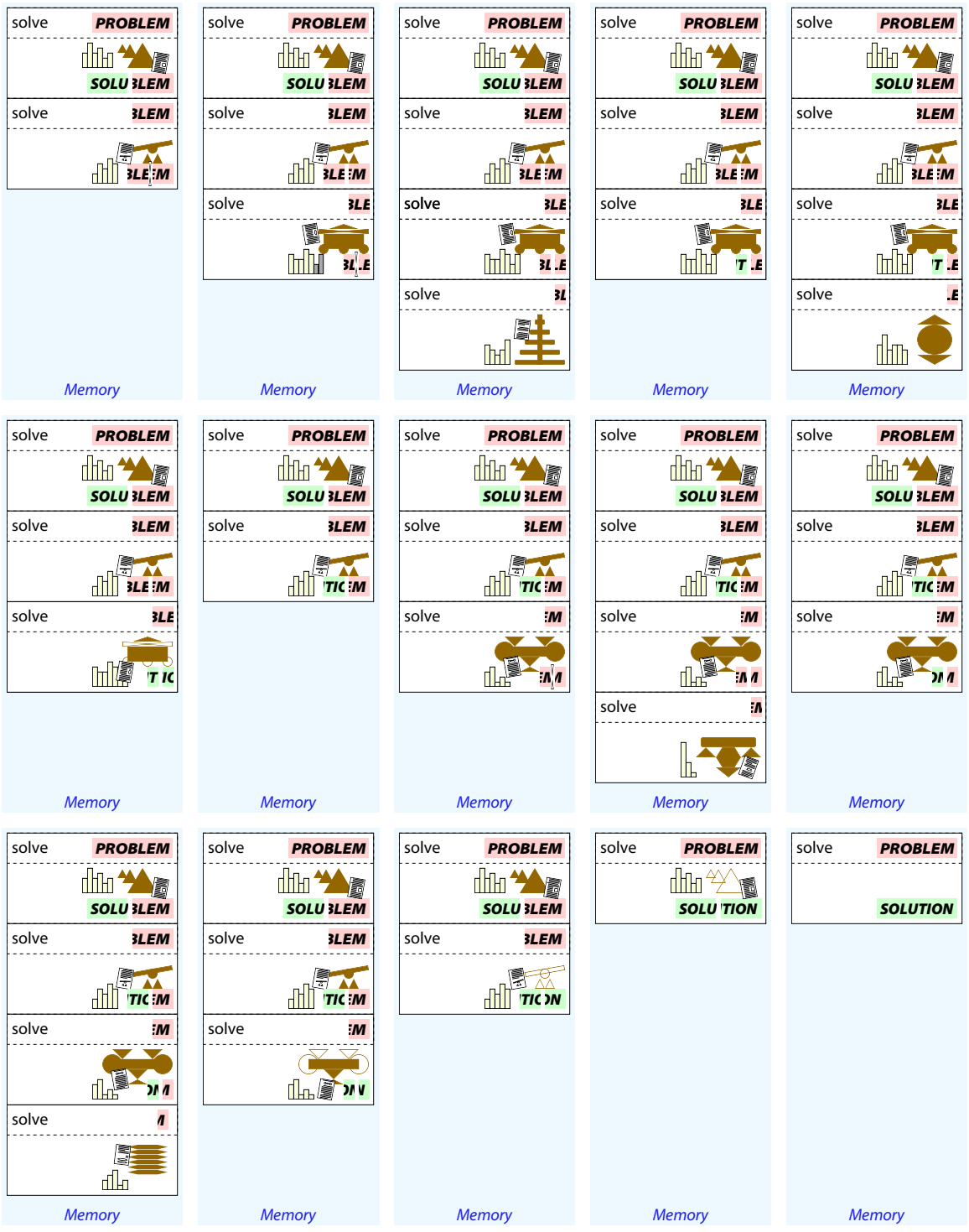
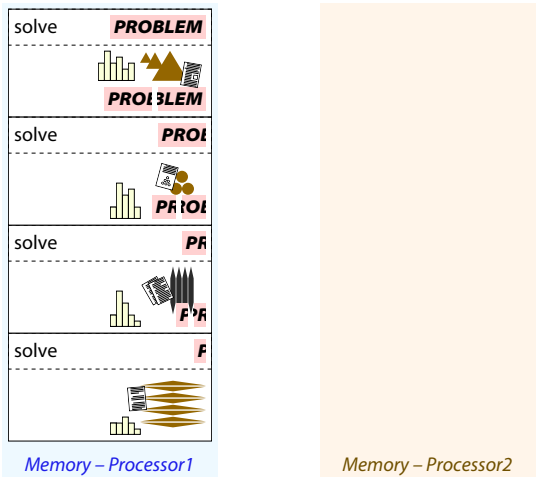
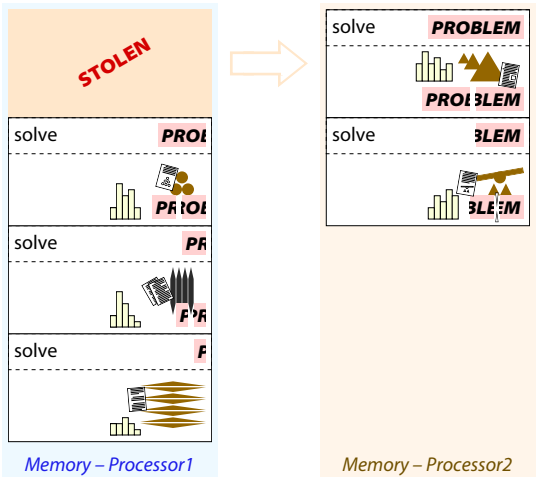


Figure C.2 (*cont.*): Serial execution of a divide-and-conquer algorithm.

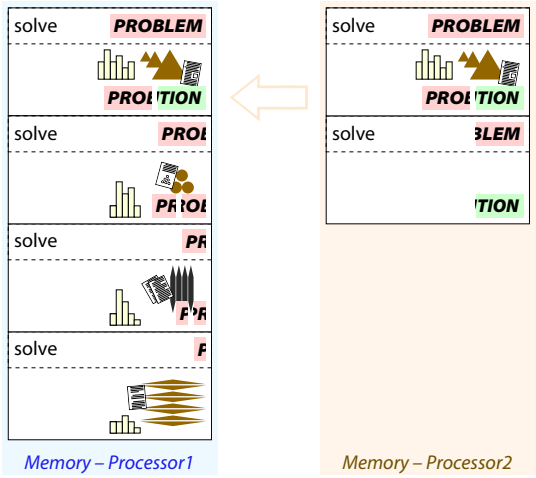


(a) Processor 2 is looking for work.

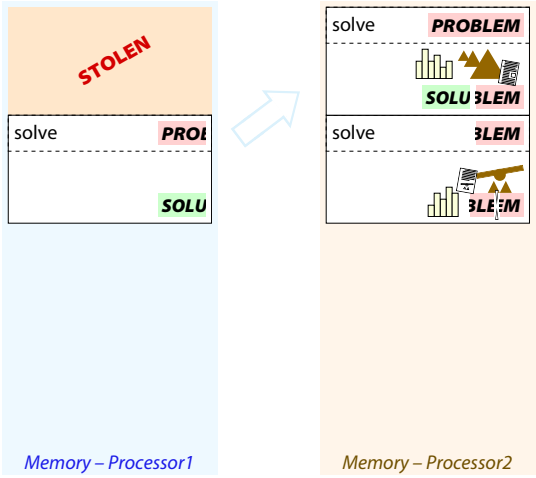


(b) Processor 2 steals work from Processor 1.

Figure C.3: How work-stealing operates in Cilk.



(a) Processor 2 finishes before Processor 1 knows its work has been stolen.



(b) Processor 1 finishes and discovers its work was stolen. Processor 2 will finish the work.

Figure C.4: Stolen work and task completion in CILK.

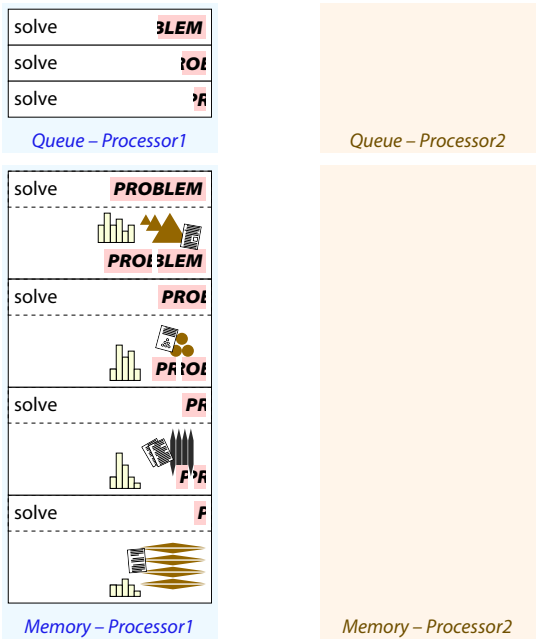
When work is stolen, there are two possibilities: either the thief will finish before the victim needs the result, or the victim will need the result before the thief has finished. Figure C.4 shows how Cilk handles task completion in these two cases. In Cilk, both cases are handled efficiently.

Figure C.5 shows how work stealing is handled by Pthieves. Unlike Cilk, Pthieves cannot steal as much context as Cilk does, leaving the thief a second-class citizen compared to its victim. This difference has important ramifications for the completion of stolen work. If the thief finishes the work before it is needed, all is well. If, however, the victim discovers the theft, it is left in a bind. In general, the victim cannot keep itself busy by stealing work, because doing so could leave it busy with other unrelated stolen work when the thief finishes the work it stole from the victim.[5] Thus, the victim should sleep. But rather than leave a (physical) processor performing no work, just before the victim sleeps, it creates a new, idle, virtual processor that is free to steal work. When the original thief finishes its stolen work and discovers that the work it stole was not completed soon enough, it wakes the sleeping victim and then dies. Thus, the number of active processors always remains constant.

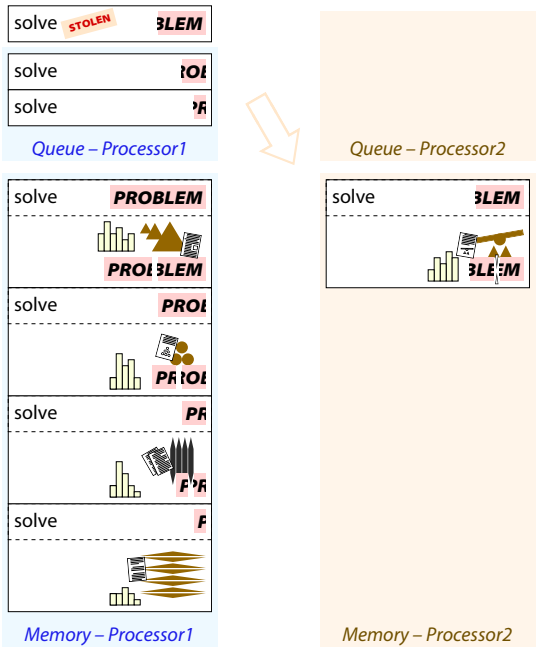
In theory, every piece of stolen work could be completed "too late", and thus cause a proliferation of operating-system–level threads implementing the virtual-processor abstractions (with most of the processors sleeping). But this scenario is no worse than calling pthread_fork instead of thread_fork—in both cases, the largest worry is not how much real memory is used, but how much address space (i.e., not how much stack is used, but how much stack space is allocated).[6] Narlikar and Blelloch (1998) indicate that this issue can be addressed by creating operating-system–level threads with small stack allocations rather than the 1 MB operating-system default. If Pthieves were to follow this model (it does not do so, currently), whenever an existing processor started to run out of stack space, it would simply create a new virtual processor and then sleep,

5. In the specific case of two processors, the victim actually *can* steal from the thief, because doing so always aids progress towards getting the original piece of stolen work completed. This two-processor optimization is supported in Pthieves, as dual-processor hardware is probably the most common multiprocessor configuration.

6. Actually, in some Posix-threads implementations, a greater worry is how foolishly the threads will be scheduled. This worry does not apply to Pthieves, however, because Pthieves always ensures that the number of active Posix threads is equal to the number of processors present in the machine.

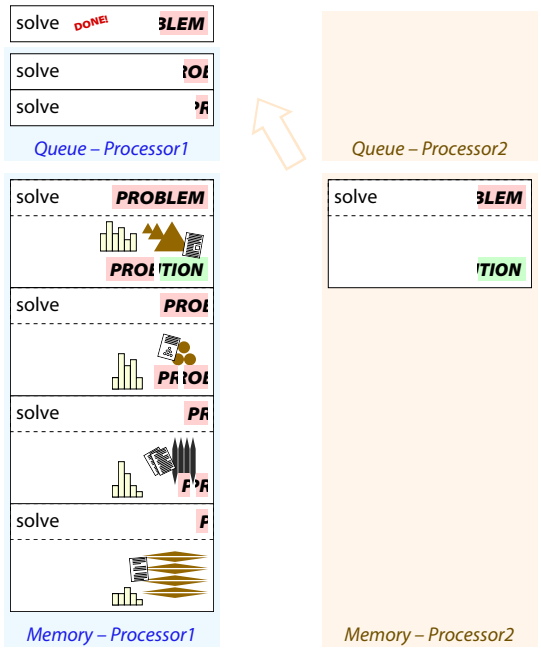


(a) Processor 2 is looking for work.

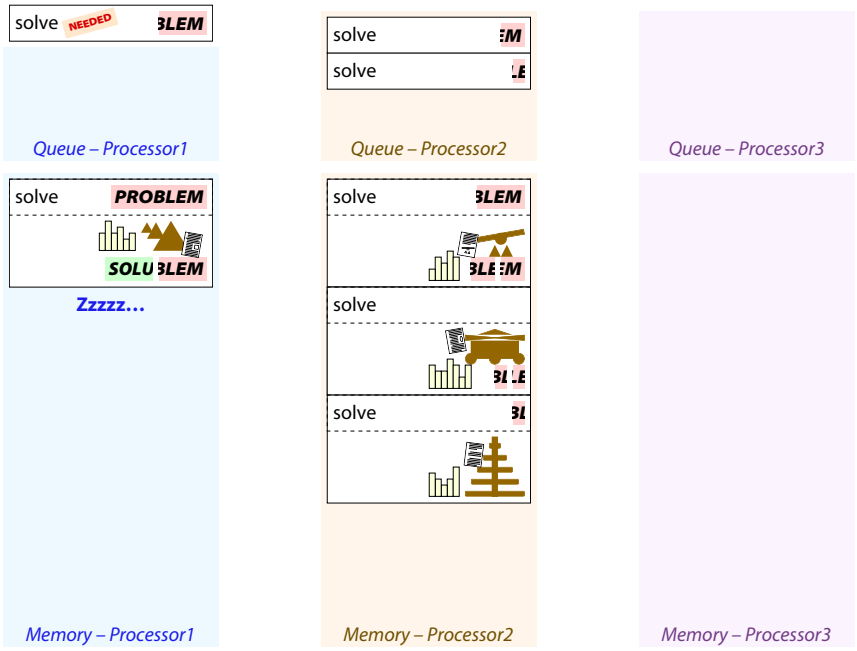


(b) Processor 2 steals work from Processor 1.

Figure C.5: How work-stealing operates in COTTON.



(a) Processor 2 finishes before Processor 1 knows its work has been stolen.



(b) Processor 1 finishes and discovers its work was stolen, so Processor 1 creates a new processor, Processor 3, and sleeps. When Processor 2 finishes its stolen work, it will wake Processor 1 and destroy itself.

Figure C.6: Stolen work and task completion in Cotton.

waiting for that virtual processor to finish its work. In practice, this technique has not proved necessary, because—for the benchmarks I needed to run—stolen work is usually completed before the theft is discovered, and the number of virtual processors created is usually only two or three times the number of physical processors.[7]

C.4 Conclusion

As the performance results in Chapter 11 show, PTHIEVES performs remarkably well, given its simplicity and lack of a rigorous proof of good memory performance.

7. In a work-stealing scheduler, provided that the amount of parallelism significantly exceeds the number of physical processors, we can expect work queues to be fairly long on average. Thus, while thefts are occuring at the far end of the work queue, the victim is busy with work at the near end of the queue.

Appendix D

The List-Order Problem

In this appendix we present an algorithm that addresses the list-order problem.[1] This algorithm has been presented previously by Dietz and Sleator (1987), but we explain it here both because maintaining an ordered list is a fundamental to our method, and to show that this technique can be implemented relatively easily.

We have chosen to present the simplest practical solution to the list-order problem that requires constant amortized time and space for insertion, deletion, and comparison. Other, more complex, solutions to the list-order problem exist, including a constant real-time algorithm (Dietz & Sleator, 1987; Tsakalidis, 1984).

Although the algorithm has been presented before, our presentation of it may be of some interest to those who might encounter it elsewhere, since we present it from a slightly different perspective, and reveal some properties that may not have been obvious in the original presentation.[2] Note, however, that for brevity we omit proofs of the complexity of this algorithm, referring the interested reader to the original paper (Dietz & Sleator, 1987) for such matters.

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

1. This appendix is a minor revision of one that appeared in a prior paper by the present authors (O'Neill & Burton, 1997).

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

D.1 An O(log n) Solution to the List-Order Problem

The algorithm maintains the ordered list as a circularly linked list. Each node in the list has an integer label, which is occasionally revised. For any run of the algorithm, we need to know N, the maximum number of versions that might be created. This upper limit could be decided for each run of the algorithm, or, more typically, be fixed by an implementation. Selection of a value for N should be influenced by the fact that the larger the value of N, the *faster* the algorithm runs (because it operates using an interval subdivision technique), but that real-world considerations[3] will likely preclude the choice of an extremely large value for N. In cases where N is fixed by an implementation, it would probably be the largest value that can fit within a machine word, or perhaps a machine half-word (see below).

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

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

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

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

To compare two elements of the list, x and y, for order, we perform a simple integer comparison of $g(\mathit{base}, x)$ with $g(\mathit{base}, y)$, where *base* is the first element in the list.

3. Such as the fact that arithmetic on arbitrarily huge integers cannot be done in constant time. In fact, if we *could* do arithmetic on arbitrary-sized rationals in constant time, we would not need this algorithm, since we could then use a labeling scheme based on rationals.

Deletion is also a simple matter, we just remove the element from the list. The only remaining issue is that of insertion. Suppose that we wish to place a new element i such that it comes directly after some element e. For most insertions, we can select a new label that lies between $l(e)$ and $l(s(e))$. The label for this new node can be derived as follows:

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

This approach is only successful, however, if the gap between the labels of e and its successor is greater than 1 (i.e., $g(e, s(e)) > 1$), since there must be room for the new label. If this is not the case, we must 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 must 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 a sufficiently wide gap, we can then insert a new node using the procedure we outlined earlier.

D.2 Refining the Algorithm to O(1) Performance

The algorithm, as presented so far, takes $O(\log n)$ amortized time to perform an insertion (Dietz & Sleator, 1987). However, there is a simple extension of the algorithm which allows it to take $O(1)$ amortized time per insertion (Tsakalidis, 1984; Dietz & Sleator, 1987), by using a two-level hierarchy: 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 ordered list element e is now represented by a node in the lower (child) list, $c(e)$, and a node in the upper (parent) list, $p(e)$. Nodes that belong to the same sublist

will share the same node in the upper list; thus,

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

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 ordered sublists are 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. That means that the parent ordered list contains $n_0/\log n_0$ entries.

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

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

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

To insert an element i after e in the overall ordered list, if the sublist that contains $c(e)$ has sufficient space, all that needs to be done is to insert a new sublist element i_c after $c(e)$, and perform the assignments $c(i) \leftarrow n_c$ and $p(i) \leftarrow p(e)$. However, if the sublist contains $2\lceil \log n_0 \rceil$ elements, it may not be possible to make insertions after some of its elements. In that case, we must 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 are given a new parent i_p which is inserted in the upper ordered list immediately after e_p.

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

Since this new scheme only creates $n/\lceil \log n \rceil$ entries in the upper ordered list, M can be slightly lower. Recall that previously we imposed the condition $M > N^2$. Now we

have a slightly smaller M, since it need only satisfy the condition

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

In practice, this condition would mean that if we required up to 2^{32} list entries, we would need an arena size of 2^{54} (instead of 2^{64}). Similarly, if we wished all labels to fit within 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 ordered list at one time (instead of 2^{16} items).

Following this scheme, we can implement efficient ordered lists and by simple derivation, a quick and effective scheme for representing relationships between tasks in a task DAG.

Appendix E

Practicalities of Parallel Ordered Lists

In Appendix D, we examined Dietz and Sleator's (1987) algorithm for providing efficient ordered lists. In this appendix, we will examine a few of the practicalities for implementing these lists, especially the issue of providing parallel access to this data structure. As with any discussion of implementation issues, there is always a question of how deep to go and when to refer the reader to the documented source code of an actual implementation. This appendix merely touches on some of the most salient features of my implementation—many more details can be found in the commented source code.

E.1 Basic Structure

Figure E.1 shows a basic ordered list. This list shows the basic properties that we would expect from the description in Appendix D: a two-level structure with an "upper list" (shown at the top, running left to right) where each upper-list node contains a lower list (shown beneath each upper node, running top to bottom).[1]

This data structure includes a useful optimization over the structure described in Appendix D: The list is singly linked rather than doubly linked. Although this optimization makes some insertion and deletion operations more awkward, this kind of ordered

1. If you are lucky enough to be viewing the figure in colour, the upper nodes are violet and the lower nodes are yellow; otherwise the upper nodes are a slightly darker grey than the lower nodes.

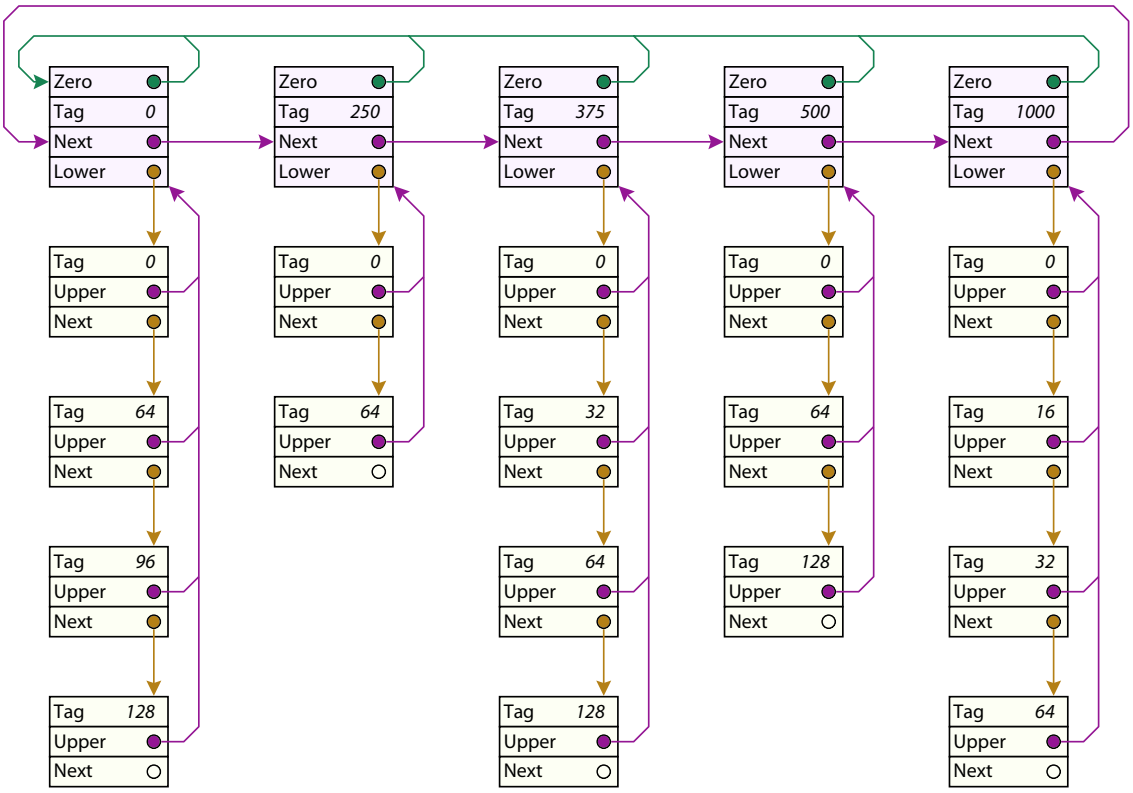


Figure E.1: A basic ordered list.

list is sufficient for many real-world situations. Also note that the lower nodes are missing Contents fields to contain the data stored in the list. This omission reflects the way we use ordered lists when implementing the fat-elements and LR-tags methods—we use the ordered-list entries themselves as version stamps; there is no actual data stored in the list.

In this realization of the ordered-list structure, a reference to an ordered-list entry is in fact a reference to a lower-list node. Each lower-list node contains a reference to the upper-list node to which it belongs. Constructing an algorithm to traverse the entire ordered list is straightforward.[2]

2. See the succ() method in gen-vstamp.h or the succ function in version-stamp.sml—see Appendix H.

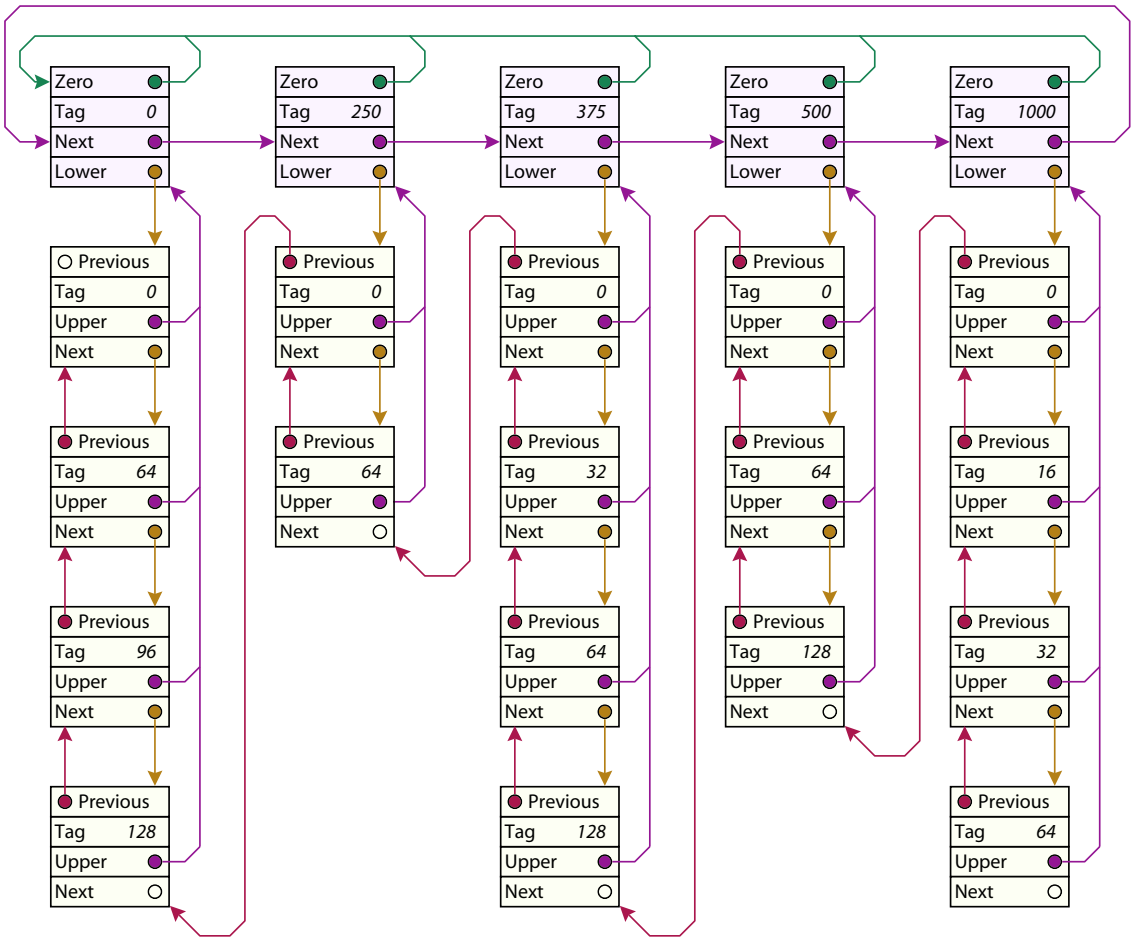


Figure E.2: An ordered list that supports deletion.

E.2 Supporting Deletion

As with ordinary linked lists, sometimes a singly-linked list is incapable of efficiently providing the desired range of operations and a doubly-linked list is required. Figure E.2 shows the ordered-list structure with backward links added to make deletion and certain kinds of insertion easier. Observe that the backward links *do not* mirror the forward links—the backward links trace the flattened ordered list, rather than following the ups and downs of the upper and lower lists. Even though there are no back links in the upper list, we can nevertheless travel backwards in the upper list using the Lower, Previous,

and Upper links (because every upper-list node always has at least one lower-list node associated with it).

E.3 Locking Strategies

If the data structure portrayed in the preceding sections looked somewhat complex, those complexities are nothing compared to the intricacies involved in allowing parallel access to the data structure. Serializing parallel access is a tricky business—it is easy to serialize access correctly, and easy to serialize access quickly, but serializing parallel accesses such that they are both quick *and* correct is a more challenging problem.

For the ordered list, the most common operation in typical use is the order query. If the two nodes being compared are in the same lower list, we use their lower tags, but if they are in different lower lists, they are compared based on the tags of the upper-list nodes to which each lower list belongs.

If the tags given to nodes were immutable, order queries would be straightforward—the insertion or deletion of other nodes in the graph would have no bearing on the tags being compared. But sometimes an insertion may require a certain amount of reshuffling (see Appendix D). One solution to this problem is to use a locking strategy that prevents reshuffles from occurring while order queries are taking place, but I have adopted a "lock-free" strategy for order queries that improves performance by allowing the list to be reorganized even as it is being read. (List insertions and deletions, on the other hand, do require locking.)

To allow a lock-free approach, the code that compares version stamps needs to be able to determine when it has read data that is consistent, and when it has read data that is inconsistent. Consistency information is communicated via a Version counter (each lower list has an associated version counter (held in the lower list's upper node), and the upper list also has a version counter (held in the first upper node)—see Figure E.3). Whenever the version counter is even, the list it guards is in a consistent state, and whenever it is odd, a reorganization is in progress.

A read begins by reading the version counter. If the version number is odd, we must wait until it becomes even. We may then proceed with reading the Tag fields necessary

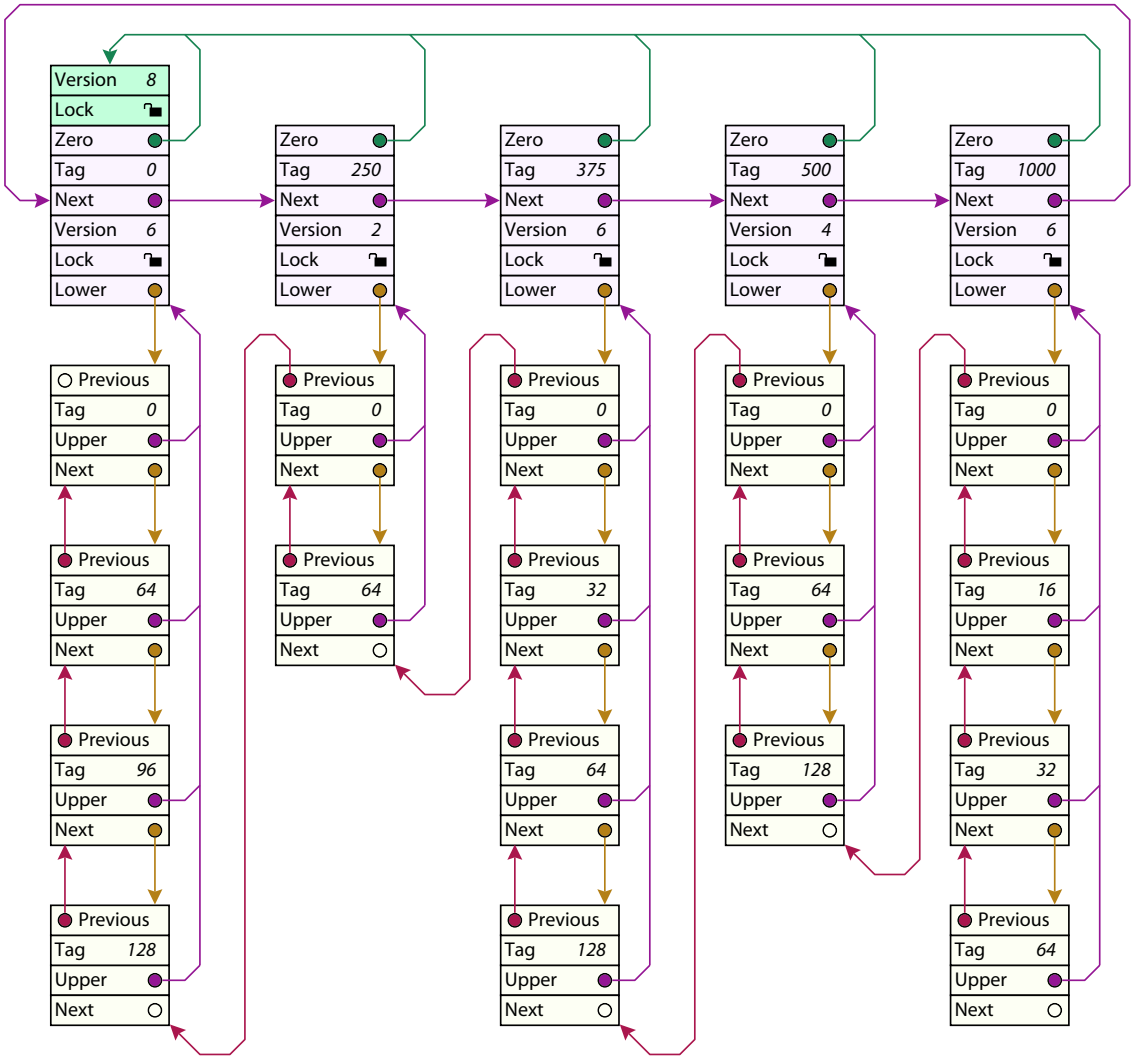


Figure E.3: An ordered list with locking support.

to perform the order query. After we have read the data structure, we check the version counter again. If it holds the same value it held when we began, we have read consistent data. If the counter's value has changed, we must start over.

There are a number of subtleties involved in this approach because the Version field for a lower-list is stored in its associated upper-list node, but reorganizations can move a lower-list node from one upper list node to another. If we allow deletion, the situation is further complicated: Between the moment when the upper node is located (by reading the Upper field of the lower node) and the moment that we read the Version field from that upper node, the lower node could be moved to belong to a new upper node as part of a reshuffle and memory containing the old upper node could have been recycled. (This problem can be solved using machine-specific locking features, or by adding a Version field to each lower node and incrementing the counter when the node's Upper pointer is changed—making otherwise invisible changes to the Upper pointer detectable.).

Insertions, deletions, and reorganizations of elements within a list use a shared/exclusive lock associated with the list to arbitrate access. When performing an insertion, a shared lock is acquired (because multiple insertions can take place concurrently), but when a reorganization is required, an exclusive lock must be acquired and the Version counter incremented. (In practice, the version counter is not incremented immediately; there is a certain amount of preparatory work that is done while the global lock is held that introduces no inconsistencies to the Tag (and Upper) values of nodes).

Deletions always require an exclusive lock on the list they are modifying. I would have preferred to allow deletions to operate concurrently with insertions, but realizing this goal has proved to be tricky in practice. For now, I have settled for a locking strategy that I know is correct, even if it may not necessarily be optimal.

E.4 Conclusion

This appendix has only touched on some of the issues involved in providing parallel access to an ordered list. The most complete documentation of my implementation can be found in the file gen-vstamp.h in the source code for the LR-tags method (see Appendix H). There are, however, several observations I can and should make.

Writing code that uses complex synchronization mechanisms can be very difficult. Tracking down simple mistakes can be arduous, and, even after all known bugs are fixed and the code appears to behave flawlessly, it is very difficult to be certain that there are no opportunities for subtle race conditions. The contrast between writing complex lock-based code with writing algorithms that adhere to Bernstein's conditions (and can be checked with the LR-tags determinacy checker) is a stark one. In my experience, code that can be checked using Bernstein's conditions can be checked and debugged quickly, whereas code that uses locks can be very difficult to debug. Although race detectors exist for programs that use locks (Savage et al., 1997; Cheng et al., 1998), these detectors require fairly primitive uses of locks, and appear to be of little use for data structures that endeavour to use a minimal amount of locking, such as the parallel ordered list we have discussed here.

Appendix F

An Alternative Method for Determinacy Checking

This appendix presents a determinacy-checking technique that was a precursor to the LR-tags method presented in the body of this dissertation. This technique is more limited than the LR-tags method because it can only check Bernstein's conditions for programs that use nested parallelism, whereas the LR-tags method works for any program whose parallel structure can be modeled using an LR-graph. In addition, this technique is merely a "nearly constant" amortized-time algorithm, rather than a constant amortized-time algorithm.

F.1 Basics of Determinacy Checking

As we discussed in Chapter 6, checking determinacy at runtime requires us to check that each read is valid given previous writes, and that each write is valid given previous reads and writes. You will recall that we need to consider prior reads when checking writes because reads might be scheduled in a different order on other runs, and error detection should not be influenced by the order of task execution, even when the tasks are scheduled in a simple serial fashion (see Listing 6.1(a) on page 79).

We can express the necessary validity conditions in terms of the relationships between tasks in the DAG task model. A read is valid if the task that last modified the data

item is an ancestor of, or is itself, the task performing the read. Writes are similar to reads, in that the task that last modified the data item must be an ancestor of the task performing the write, but writes also require that all reads done since the last write are also ancestors of the task performing the write.

As we discussed in Section 7.1, we associate each datum d with both the last writer for that datum, $w(d)$, and the set of tasks that have accessed that datum since it was written, $R(d)$. We may think of $R(d)$ as the "reader set" for d, but $R(d)$ also includes the task that last wrote d. Given these definitions, Bernstein's conditions become:

- *Reads* — A read is valid if the task that last modified the data item is an ancestor of (or is itself) the task performing the read. Expressing this restriction algebraically, a task t may read a datum d if

$$w(d) \trianglelefteq t \tag{Bern-1}$$

  where $w(d)$ is the task that wrote the value in d. When a read is valid, we update $R(d)$ as follows:

$$R(d) := R(d) \cup \{t\}$$

- *Writes* — Writes are similar to reads in that the task that last modified the data item must be an ancestor of the task performing the write, but writes also require that all reads done since the last write must also have been done by ancestors of the task performing the write. Thus, a task t may write a datum d if

$$\forall r \in R(d) : r \trianglelefteq t \tag{Bern-2}$$

  where $R(d)$ is the set of tasks that have accessed d since it was last written (including the task that performed that write). If the write is valid, we update $R(d)$ and $w(d)$ as follows:

$$\begin{aligned} w(d) &:= t \\ R(d) &:= \{t\} \end{aligned}$$

At this point, we will deviate from the determinacy-checking framework set out in

Chapter 7. Observe that we can use a single value to represent all the reads instead of maintaining the set $R(d)$ for each datum, d. Given the function ncd(T), which finds the nearest common descendant in the task DAG for some set of tasks T we can use the equivalence $(\forall x \in T : x \lhd t) \Leftrightarrow (\text{ncd}(T) \trianglelefteq t)$, which is the definition of nearest common descendant, to change the test for writes to $(r(d) \trianglelefteq t) \wedge (w(d) \trianglelefteq t)$ where $r(d) \equiv \text{ncd}(R(d))$. It is a simple matter to keep $r(d)$ updated as reads occur, since $\text{ncd}(\{r(d), t\}) \equiv \text{ncd}(R(d) \cup \{t\})$.

By maintaining and checking values for $r(d)$ and $w(d)$ for each datum d we have a simple runtime check for determinacy that can be applied to any program that can be described using the DAG model. This method does, however, presuppose that we can determine the nearest common descendant of two tasks, which, for a dynamic parallel program, may or may not be practical, depending on how the DAG is generated.

Many parallel algorithms are not hampered by this problem, however, because they do not need the full power of an arbitrary DAG to describe how their tasks relate and can, instead, be described by a series-parallel graph. If we restrict ourselves to this class of parallel algorithms, we find that we can trade the graph model for a dynamic tree, and, in so doing, be able to solve ancestor queries more quickly and avoid solving the nearest common descendant problem in the general case.[1]

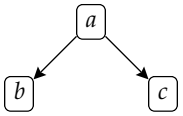
Figure F.1 shows how the tasks that were represented using a static series-parallel graph in Figure 6.1(a) can be represented using a dynamic tree. In this representation, the tree expands as tasks are forked and contracts again when they join. When a join occurs, the parent's children are removed from the tree and all their work becomes attributed to the parent—as if the children had never existed and all the work had been done by the parent.[2]

We have outlined the basis of this determinacy-checking technique, but we must

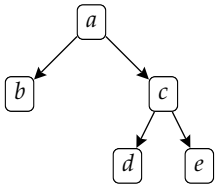
1. The nearest common descendant problem in a DAG becomes the nearest common ancestor problem for a dynamic tree, which we then solve in just-in-time fashion through a useful side effect of other behaviour.

2. The parent and child metaphor fails us here, since it seems wrong that parents should always outlive their children, and strange that they should assume the credit for their children's work. It would probably be better to talk of contractors and subcontractors, since it seems more plausible that a contractor would be around for a longer period of time than its subcontractors, and that it would sleep while its subcontractors work, and then take credit for all the work that the subcontractors had done. But we will stick to parent and child, because these are the terms in common usage.

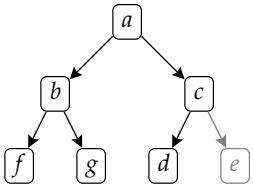
(a) Initially, there is just one task: a.



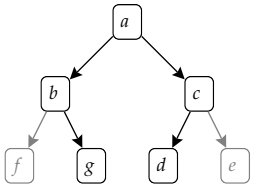
(b) a forks two children, b and c, which begin running. a is suspended until both its children have terminated.



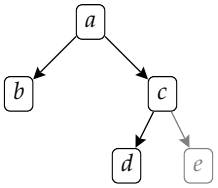
(c) c forks d and e; c is suspended until d and e terminate.



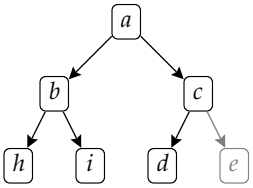
(d) b forks f and g. e dies.



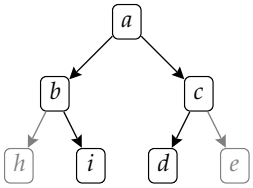
(e) f dies.



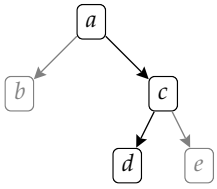
(f) g dies—f and g are removed from the tree; their work becomes attributed to their parent b which resumes (represented as task b' in Figure 6.1(a)).



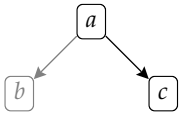
(g) b forks two new children, h and i.



(h) h dies.



(i) i dies; b takes credit for its children's work and resumes executing (task b'' in Figure 6.1(a)); b completes its work and dies.



(j) d dies—d and e are removed from the tree and c takes credit for their work and resumes (the task becomes c' in Figure 6.1(a)).

(k) c dies—b and c are removed from the tree and have all their work attributed to a (a' in Figure 6.1(a)); a completes execution and itself dies.

Figure F.1: Using a dynamic tree instead of a series-parallel graph. Here we represent one possible parallel execution sequence for the series-parallel graph given in Figure 6.1(a).

now show that we can solve ancestor queries quickly for a dynamic tree, how we can attribute the work of child tasks to their parents, and how to fit these pieces together to produce a viable method for determinacy checking.

F.2 Solving Ancestor Queries for a Dynamic Tree

In Figure F.1, we saw how we may represent the current state of tasks using a tree, rather than a full series-parallel graph. This representation makes the problem of solving ancestor queries easier. Although we will eventually need to solve ancestor queries in a dynamic tree, we will begin by examining the existing work on this problem, which has been directed at solving such queries for static trees.

Schubert et al. (1983) developed a straightforward labeling for a static tree that can quickly solve ancestor queries. The label for each node i has two components, which we shall call $l(i)$ and $u(i)$. The first component $l(i)$ is an integer representing the order in which the node is encountered in a preorder traversal of the tree (thereby giving every child an l-label greater than that of its parent or any of its ancestors, and, hence, $l(j) \geq \max\{l(i) \mid i \trianglelefteq j\}$). The second component $u(i)$ is defined to be an integer one greater than the largest l-label in the subtree rooted at i (thus $u(i) > \max\{l(j) \mid i \trianglelefteq j\}$).[3] An example of this labeling is shown in Figure F.2. (Note that in my examples, nodes have either two children or none, but the principles apply equally well to arbitrary numbers of children.)

Using this scheme, it is a simple matter to determine the positional relationship of one node to another. For two nodes, i and j,

$$
\begin{aligned}
i = j &\Leftrightarrow l(i) = l(j) \\
i \lhd j &\Leftrightarrow l(i) < l(j) < u(i) \\
j \lhd i &\Leftrightarrow l(j) < l(i) < u(j) \\
i \wr j &\Leftrightarrow l(i) < l(j) \geq u(i) \\
j \wr i &\Leftrightarrow l(j) < l(i) \geq u(j).
\end{aligned}
$$

3. Schubert et al.'s value for $u(k)$ is actually one *less* than the value presented here, but the principles are the same.

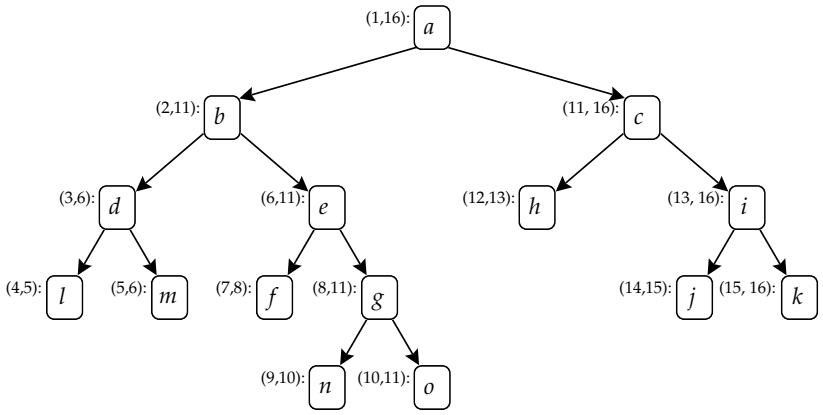


Figure F.2: Labeling a static tree to solve ancestor queries.

Schubert's approach is designed for labeling a static tree, but if we use rational numbers for labels rather than integers, we can easily extend Schubert's method to support dynamic trees. Unfortunately, using rational labels is inefficient, but by examining this revision we lay the groundwork for our final solution. The underlying theme is that we may insert new labels in the space between old ones.

Figure F.3 shows a tree labeled with rationals rather than integers. The root is labeled such that $l(r) = 0$ and $u(r) = 1$ (although any values such that $l(r) < u(r)$ would have sufficed). As before, $i \trianglelefteq j \Leftrightarrow l(i) \leq l(j) < u(i)$ and $u(i) > \max\{l(j) \mid i \trianglelefteq j\}$.

To add children, $c_1, \ldots, c_n$, to a leaf i (making it a parent node) we subdivide the interval $(u(i) - l(i))$ between the two labels of the parent node. We label the kth child c_k such that

$$l(c_k) = l(i) + k \times \frac{u(i) - l(i)}{n + 1}$$

and

$$u(c_k) = l(i) + (k + 1) \times \frac{u(i) - l(i)}{n + 1}.$$

This revised labeling scheme supports dynamic trees that grow at the leaves.[4] But using rationals as labels is problematic because their numerators and denominators grow

4. This method also easily accommodates tree growth from the addition of leftward children because the interval between $l(t)$ and $l(c_1)$, where t is a parent and c_1 is a leftmost child, is never filled by subsequent growth at the leaves. Other growth patterns and wholesale dynamic reorganizations might be more difficult to handle, but these limitations are of no concern to us in this discussion.

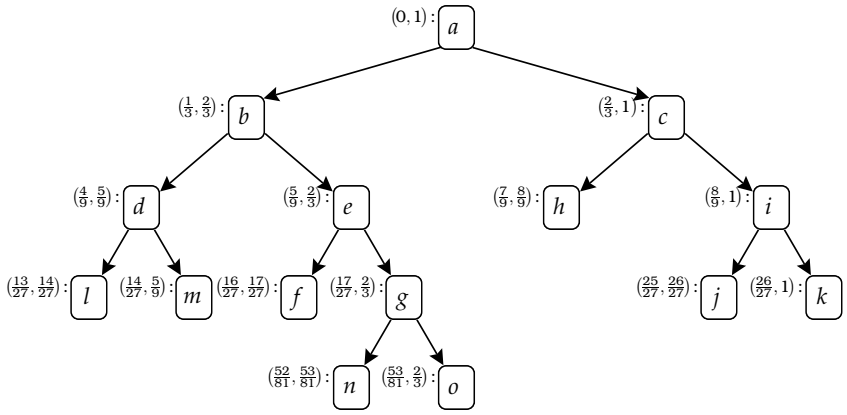
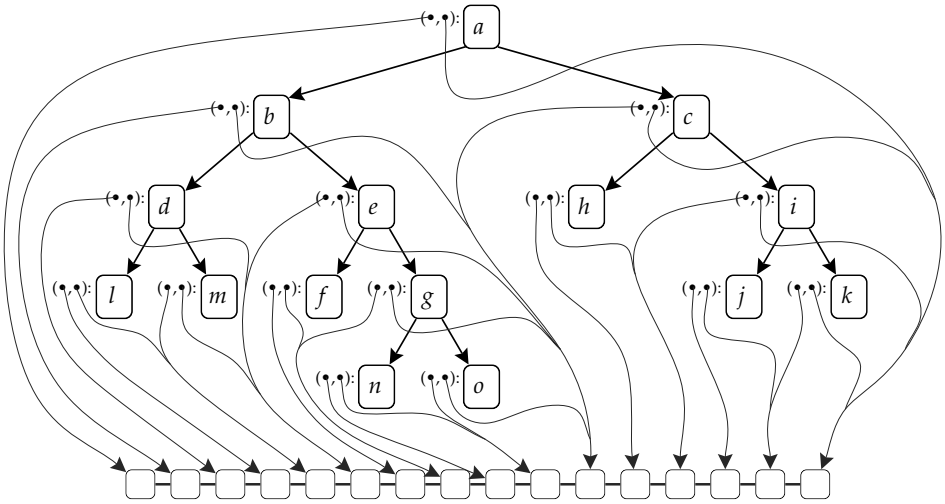


Figure F.3: Labeling a dynamic tree to solve ancestor queries.

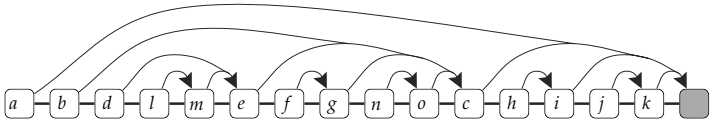
exponentially with the level of the tree. The rationals method shows that a labeling scheme that can insert labels between existing labels and can compare labels for order is all that is required to support ancestor queries in a dynamic tree, but rationals themselves, although conceptually simple, would not have the constant-factor time and space overheads we require, so we must look to other labeling methods with similar insertion and comparison properties.

Figure F.4(a) shows how we can replace the rationals we used in Figure F.3 with references to the elements of an ordered list—an ordered list being a data structure that supports the insert, delete, and successor operations of a linked list, but can also swiftly compare two list items to see which comes first in the list (Dietz & Sleator, 1987; Tsakalidis, 1984). Instead of being rationals, $l(k)$ and $u(k)$ are references to elements in in such an ordered list.

As in the previous scheme, we add children by generating new node labels that lie between two existing labels, but rather than dividing a numerical interval, we achieve this objective by adding new ordered-list elements that lie between existing ordered-list elements. To add n children, $c_1, \ldots, c_n$, to a leaf i making i a parent, we add n items, $e_1, \ldots, e_n$, to the list between $l(i)$ and $u(i)$, setting $l(c_1) = e_1, \ldots, l(c_n) = e_n$, and $u(c_1) = e_2, \ldots, u(c_{n-1}) = e_n, u(c_n) = u(i)$. Thus, prior to spawning its children (when i was a leaf), $l(i)$ and $u(i)$ were contiguous, but afterwards the entries for the children lie between $l(t)$ and $u(t)$. In the previous method we compared labels for order using their



(a) Using pointers to entries in an ordered list as node labels.



(b) Clarifying the diagram by not depicting the task tree. This diagram is complex because it focuses on labeling a tree. But we do not need to represent the tree at all. By removing it, $l(k)$ implicitly represents the position of k in the ordered list, with the arrows now representing $u(k)$.

Figure F.4: Using an ordered list to solve ancestor queries.

numerical value, but now we compare labels based on their position in the ordered list; thus, $x < y$ if the element that x references comes before the element that y references in the ordered list.

All that remains is to clean up the representation, which looks ugly as presented so far. We have never actually needed to explicitly represent the task tree itself (we need only store $l(k)$ and $u(k)$ for each task, k), so it can be removed. Also, the mapping from k to $l(k)$ is one-to-one, so we can make $l(k)$ and k synonymous and store any information k stored (including $u(k)$) in the ordered-list node at $l(k)$—previously the ordered list contributed nothing but its ordering, but now it also stores information. This yields our final data structure, shown in Figure F.4(b).

Solutions to the ordered-list problem (Dietz & Sleator, 1987; Tsakalidis, 1984) can

perform insertions, deletions, and order queries in constant time.[5] By using such an ordered list, we can perform ancestor queries on a task tree in constant time and extend the tree in constant time and space per child added.[6]

F.3 Solving Ancestor Queries for Parallel Tasks

In the preceding section, we presented a method for solving ancestor queries in a dynamic tree, yet our problem is to solve ancestor queries at runtime for a set of tasks in a run that can be described by a series-parallel graph. We have already seen (in Figure F.1) that we can use a dynamic tree to represent the current state of parallel tasks that fork and join. The complication is that at a join, when execution resumes at the parent, we must not only delete the children from the tree, but attribute all the work done by the children to their parent.

Indirection nodes (Turner, 1979; Ehrig, 1978; Peyton Jones, 1987), a technique long used in the graph-rewriting community, provide us with the means we need to attribute the work of children to their parents. When all the children of a node have terminated, we take their tags out of the ordered list, and overwrite them with an indirection node that points to their parent (see Figure F.5). Anything that had been tagged with the child's tag is now effectively tagged with the parent's tag.

Over time, indirections can form chains as tasks die, so we need a technique that will short-circuit indirection chains as they are followed and ensure that they are as short as possible. One option is to use a variation of Tarjan's disjoint-set union algorithm (1975). Tarjan's algorithm is not quite amortized constant time per operation, but is close enough to be the technique of choice for a real-world implementation.

Theoretically, however, we can develop a true amortized linear-time algorithm using a variant of Gabow and Tarjan's linear-time discrete-set union algorithm (1985). Gabow and Tarjan's algorithm requires that we know what task any given task will be indirected to, and we do: its parent. Gabow and Tarjan's algorithm also requires us to know the

5. As in the rest of this dissertation, Dietz and Sleator's simpler constant–amortized-time algorithm is usually the preferable choice.

6. Our complexity results assume that tasks are running under a serial scheduler; under a parallel scheduler some serialization is required, potentially resulting in some loss of parallelism.

number of tasks that will be created ahead of time. Because we cannot know this information in advance, we must use the same technique that is used to build dynamic arrays from static arrays—*successive size doubling*.

Successive size doubling works as follows: You begin with a data structure that can hold k items. When that data structure is full, you copy all the items into a new data structure that can hold twice as many items. Successive doubling even supports item deletion by not truly deleting items, but simply not copying them when we relocate data from one data structure to the next.

Adding successive doubling makes Gabow and Tarjan's algorithm take constant amortized time, rather than constant worst-case time, but amortized time is sufficient to make the overheads of our technique constant.

This technique applies not only to our method, but, according to Feng,[7] can be adapted for Feng and Leiserson's SP-Bags determinacy algorithm as well.

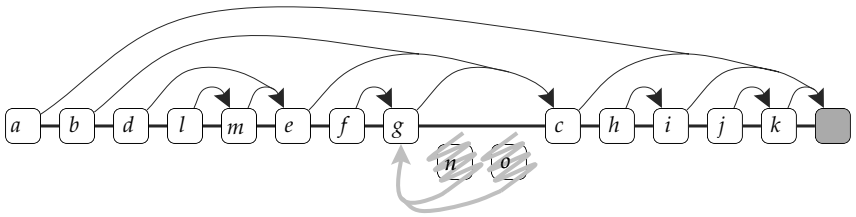
F.4 Checking Determinacy

So far, we have developed a method for generating task tags that support ancestor queries (and the initiation and termination of child tasks) in nearly-constant time; now we simply need to show how these task tags can be used to tag data to enforce determinacy.

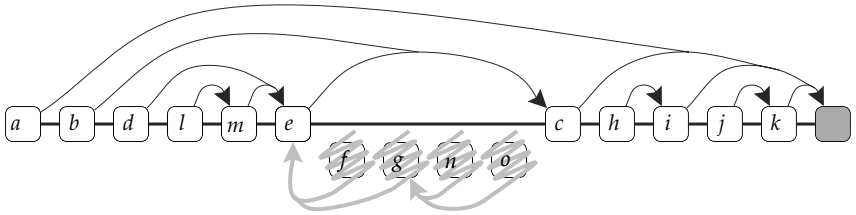
As we saw in Section F.1, to support reads we need to keep, for each datum d a record of the tag of the task that last wrote the data, $w(d)$, and compare that tag to the tag of the current task, t. If $w(d) \trianglelefteq t$, the read is valid.

For writes, the situation is a little more complex. We have the same condition as reads, $w(d) \trianglelefteq t$, but we also have to ensure that all reads since the last write were performed by ancestors of this write's task; thus, if $R(d)$ is this set of reads, $\forall r \in R(d) : r \trianglelefteq t$. We do not, however, need to store all tasks that have performed reads to check this condition. If two tasks, i and j, are to be added to the read-set, and $i \trianglelefteq j$, we need only store j, because for some task, k, $j \trianglelefteq k \Rightarrow i \trianglelefteq k$ (the ancestor relation is transitive). We also need to correctly maintain $R(d)$ as tasks terminate.

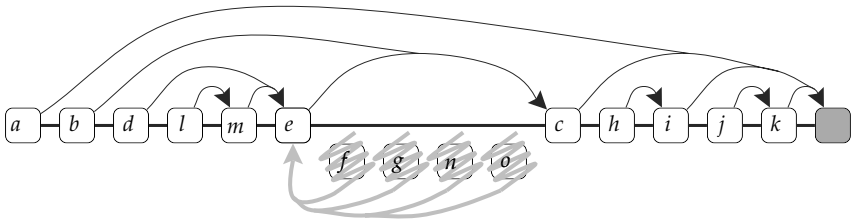
7. Personal communication via email, 16 December 1998.



(a) The ordered list after n and o have both terminated and execution resumes at their parent, g. Notice that n and o are removed from the ordered list and overwritten with indirections to g.



(b) The same ordered list after f and g have both terminated and control resumes at their parent, e.



(c) If the tags for n or o are accessed, the indirection chains are shortened so that they point directly to e (which is the standard behaviour for indirection nodes).

Figure F.5: Using indirection nodes to attribute the work of terminated children to their parents.

In the previous section, we outlined how indirections are used when children terminate. When all the children of a task have terminated, references to the children become synonymous (though indirection) to references to the parent. For example, if $R = \{b, f, g\}$, where f and g are children of c, c does not need to be in R while f and g are running. However, after both f and g have terminated, they are synonymous with c; thus, $R = \{b, c, c\} \equiv \{b, c\}$.

We can optimize $R(d)$ to ensure that $|R(d)| \leq 2$. For any set R where $|R| \geq 2$, $\exists r_l \in R, r_g \in R : \text{ncd}(\{r_l, r_g\}) = \text{ncd}(R)$, provided that the task DAG is a series–parallel graph. One possible choice for r_l and r_g is the least and greatest of R's members (as ordered in the ordered-list representation). Thus, r_l and r_g are the leftmost-uppermost and rightmost-lowermost nodes in the task DAG.[8] Let us also define $r(d)$ as the value held by $r_l(d)$ and $r_g(d)$ when $r_l(d) = r_g(d)$ (when $r_l(d) \neq r_g(d)$, $r(d)$ is undefined), with updates to $r(d)$ updating both $r_l(d)$ and $r_g(d)$.

The full rules for reads and writes in a task t on datum d are as follows:

- *Reads* — A read is valid if $w(d) \trianglelefteq t$. For a valid read, if $r(d)$ is defined and $r(d) \trianglelefteq t$, we update $r(d)$ to be t; otherwise, we update $r_l(d) := \min(r_l(d), t)$, and $r_g(d) := \max(r_g(d), t)$.

- *Writes* — A write is valid if $r(d)$ is defined and $r(d) \trianglelefteq t$. For a valid write, we update both $w(d)$ and $r(d)$ to be t.

The ordering of tasks for $\min$ and $\max$ is the same as the ordering of task l-labels: Tasks that come earlier in the ordered list are considered to be less than those that come later.

Notice that we never need to calculate the least-common ancestor of $r_l(d)$ and $r_g(d)$, because when execution resumes in their least-common ancestor, the use of indirection to attribute the work of children to their parents will mean that $r_l(d) = r_g(d)$, and writes will once more be allowed.

F.5 Alternate Strategies

We chose r_l and r_g as two distinguished members of R that could act as surrogates for the entire set, but r_l and r_g are not the only possible choices. In this section, we

8. This choice of r_l and r_g only works when the task DAG is a series–parallel DAG.

will examine a variant of the previous method that uses the leftmost-lowermost and rightmost-lowermost nodes as the surrogates for R.

Let $s_t(x)$ be the **situation** of x with respect to t (where t is the current task, and x is some task that is either executing or has completed execution), defined as follows:

$$s_t(x) = \begin{cases} \mathsf{UP} & \text{if } x \trianglelefteq t \\ \mathsf{LHS} & \text{if } x \wr t \\ \mathsf{RHS} & \text{if } t \wr x \\ \mathsf{DOWN} & \text{if } t \triangleleft x \end{cases}$$

Intuitively, UP means that x lies on the path upwards from t to the root of the tree; LHS means x lies to the left-hand side of t in the tree; and RHS means that x lies to the right-hand side of t in the tree. During actual execution, DOWN will never be returned (because it would mean that t could see one of its children while it was executing—remember that t sleeps while its children execute, and after they are complete, the children are indirected to become indistinguishable from the parent), but we do not depend on this actuality.

We define the frontmost member of R, r_f, and the backmost member r_b to be the unique tasks such that

$$\forall t \in R : (t = r_f) \vee (s_t(r_f) = \mathsf{LHS})$$
$$\forall t \in R : (t = r_b) \vee (s_t(r_f) = \mathsf{RHS})$$

(Note that this definition assumes that R does not contain any elements related by $\triangleleft$—as have already seen, R can adequately represent the read set without these superfluous elements.)

With these definitions, we only need to store $r_f(d)$ and $r_b(d)$ for each datum d rather than the full read set, $R(d)$. The full rules for reads and writes in a task t on datum d are as follows:

- *Reads* — A read is valid if $w(d) \trianglelefteq t$. For a valid read, if $s_t(r_f(d)) \neq \mathsf{LHS}$, we perform $r_f(d) := t$. Similarly, if $s_t(r_b(d)) \neq \mathsf{RHS}$, we perform $r_b(d) := t$.

- *Writes* — A write is valid if $s_t(r_f(d)) = \mathsf{UP} \wedge s_t(r_f(d)) = \mathsf{UP}$. For a valid write, we update $w(d)$, $r_f(d)$, and $r_b(d)$ to be t.

This alternate technique requires more memory writes than its predecessor, but has some advantages for optimization as we will see in Sections F.7 and F.6. Moreover, it has the advantage that r_f and r_b can be surrogates for a read-set in more classes of graph than r_l and r_g as previously defined; in fact, r_f and r_b are exactly the same nodes that would be chosen by the techniques given in Section 7.2.

F.6 Time and Space Complexity

The ordered-list data structure requires only constant amortized time for every access, and space proportional to the number of items in the list. Current tasks are represented in the order list, and terminated tasks become indirections to current tasks. We should, perhaps, consider whether the list could be flooded with indirections to deceased tasks, but there can be at most $O(\min(i, n))$ indirections at any given time, where i is the number of indirections created (and thus is proportional to the number of children that have been reaped), and n is the number of checked data items. Those indirections which are not in use can be garbage collected[9] (via reference counting, or any other garbage-collection method that adds constant-factor overhead). Thus, the space overhead is $O(n)$ for n checked data items.

The indirections themselves require amortized constant space and amortized constant time for access.[10]

For serial execution of a parallel program, we have constant amortized-space overhead and constant amortized-time overhead for determinacy checking, but we also need to consider the issue of contention for the ordered list during parallel execution. Ideally, the ordered list would support concurrent access, but existing efficient algorithms for ordered lists require some degree of serialization. Thus, best-case performance would provide constant overheads, but more typical performance would incur some slowdown due to loss of parallelism. The extent of this slowdown will vary from application to application.

9. If we are convinced that i will be small, we can refrain from garbage collecting and save the expense of maintaining reference counts.

10. Real-world implementations will probably settle for the constant space and nearly constant time provided by Tarjan's classic algorithm (1975).

F.7 Speeding Parallel Performance

The above technique works for parallel programs, but requires repeated accesses to the ordered-list data structure to answer situation queries. In any parallel system, contention for a shared structure presents a potential bottleneck, so we should consider whether it is possible to avoid that bottleneck.

In the naïve determinacy-checking scheme we discussed in Section 7.1, I mentioned that it was possible to perform a query once, and then cache the result. The same technique can be used to reduce the number of queries performed by the directory technique discussed in Section F.5. In any task t the value of $s_t(x)$ will remain the same during t's life—even if x dies and is indirected to its parent. Thus, tasks can cache the answers to situation queries. Parents can even copy their caches to their children.

If we employ **cache preloading**, we can avoid querying the ordered-list data structure for close relatives altogether, where the close relatives of a task are its ancestors and the direct children of those ancestors. We can preload the caches because the parent knows how its children relate to each other and itself. If a parent p creates two children, a and b, the parent can preload a's cache with $s_a(p) = \mathsf{UP}$, $s_a(a) = \mathsf{UP}$, and $s_a(b) = \mathsf{RHS}$; and preload b's cache with $s_b(p) = \mathsf{UP}$, $s_b(a) = \mathsf{LHS}$, and $s_b(b) = \mathsf{UP}$.

Adding a cache raises some design questions, such as determining an appropriate cache size and replacement policy. Different applications will benefit from different cache sizes. At one extreme we might have no caching at all, whereas at the other extreme we might have a cache that grows dynamically, never forgetting any cached information.

Another question is whether we should cache results before or after following indirections. Caching before following indirections eliminates the time spent following those indirections, but caching after following indirections may reduce the number of cached entries and also allows the parent to use data cached by its terminated children (but only after all the children have terminated and have been redirected to point to the parent).

These questions do not have obvious theoretical answers, but I have found through experience that a small cache can improve the parallel execution performance of this determinacy-checking method when contention for the ordered list is high.

F.8 Speeding Serial Performance

If we are executing our program serially and use the caching techniques above, with cache preloading, a dynamic cache that does not discard information, and caching after following indirections, we discover something interesting: The ordered-list data structure is never queried. Every situation query is serviced from the cache.

We can, therefore, abandon our ordered-list data structure entirely, and call our cache the **map**: m. Each task s has an entry $m(s) \in \{\mathsf{LHS}, \mathsf{UP}, \mathsf{RHS}\}$, which defines its relationship to the currently executing task, t. As we move from one task to the next in a depth-first execution, we adjust the map so that it remains correct (altering at most two map entries).

If we execute child tasks from left to right, the map will never hold the value RHS for any task, since the task we are executing will always be the rightmost-lowermost task.[11] Not only is the map now only storing one bit of information per task, but the $r_b(d)$ entry for each datum d is redundant and can be eliminated.

Our final, optimized algorithm for the serial case has been discovered before, by Feng and Leiserson (1997). A task t in an S-Bag in Feng and Leiserson's algorithm exactly corresponds to $m(t) = \mathsf{UP}$; similarly, membership in a P-Bag exactly corresponds to $m(t) = \mathsf{LHS}$. (Interestingly, I did not realize that my optimizations resulted in Feng and Leiserson's algorithm until I implemented their algorithm to compare it against my own, and found that it was operationally identical to my serial-case optimization.)

F.9 Conclusion

Although the technique I have described in this appendix is less efficient in practice than the LR-tags technique discussed in the body of the dissertation, it did serve as a useful stepping-stone to developing that slightly-faster method. Along the way, we have shown that Feng and Leiserson's algorithm can be seen as a specialization of this more-general algorithm, and that both algorithms could, at least in theory, run in amortized-constant time, rather than nearly constant time.

11. Children may have been lower or further right, but for our task to be running, those children must have terminated and been replaced with an indirection to their parent.

Appendix G

Mergeable Parallel Fat-Element Arrays

This appendix examines techniques for rapidly merging several fat-elements–array versions into a single array version. I developed these techniques with an eye towards developing an interesting parallel functional-array data structure. Although this work was instrumental in inspiring me to develop my LR-tags determinacy-checking technique, the array-merge mechanism I developed seemed to have little application. In a sense, what I describe here has been a research dead end for me, but I shall still outline some of the techniques I developed, both because they are interesting historically (this work lead me into the study of determinacy checking), and also in the hope that some future reader may see some applications for these techniques that I do not.

G.1 Simple Parallelism

Functional programming languages are often cited as good candidates for parallel execution. The absence of side-effects and mutable global state mean that function arguments can not only be executed in arbitrary order, but that they can also be evaluated in parallel.

Fat-element arrays are amenable to parallel access with a few caveats: The version list for each master array requires some access arbitration to serialize the creation of version stamps and to prevent two threads from attempting to split an array at the same time. Similarly, the fat-elements master array may require a per-element lock to prevent corruption if two different tasks simultaneously attempt to modify the tree

representing a fat element. If the number of array elements is large compared to the number of processors, it seems reasonable to assume that fat-element arrays could be used in parallel code without lock contention causing severe loss of parallelism.

Note that other functional-array techniques may not be as amenable to parallel access. The trailers technique, for example, suffers poor time performance if the array is accessed non–single-threadedly, and a multithreaded program is, by definition, not single threaded. On the other hand, tree-based arrays offer lower synchronization overheads because the immutable nature of the tree representation eliminates any need for locking.

Although fat-element arrays may be accessed in parallel, every update performed in each parallel thread will generate a distinct new array version. There is no way for several threads to cooperate in building a single array result—if we desire such a result, each thread must create its own array version and we must then merge the results after all those threads have terminated.

Merging several versions of an array of size n appears to require $\Theta(n)$ work, which may be acceptable if some of the threads performed $\Omega(n)$ array accesses or there were $\Omega(n)$ threads, but if each thread performed only a few accesses, the merge becomes a bottleneck. In the rest of this appendix, we will examine some techniques to remove this bottleneck for a particular kind of array merge.

G.2 Mergeable Fat-Element Arrays

There are several possible strategies for merging the array values produced by independent threads that modify the same initial array, including

1. Using a function to combine the element values. For example, we could produce a merged array in which the ith element of the merged array is the sum of the ith element of each of the provided array versions.

2. Using a function to select the element values. For example, we could produce a merged array in which the ith element of the merged array is the maximum of all the ith elements of each of the provided array versions.

3. Assuming that no two threads access the same element, and creating a merged array by applying the writes performed in each thread to the original array.

4. Creating a merged array by applying the writes performed in each thread to the original array in deterministic order (thus, if two array versions change the same array location, we can always know which array version's value will be preferred).

If we adopt the first merge strategy, it is reasonable to assume that a general array merge mechanism will require $\Theta(pn)$ time, where p is the number of arrays that must be merged, and n is the size of the array. This merge time can be improved to $\Theta(n)$ if the merge is performed in parallel on p processors. (Obviously, in certain specialized circumstances, where more is known about the array contents, a faster merge may be possible.)

The remaining strategies are selection strategies. A selection strategy may require less execution time because the selection function does not need to be applied to elements that are unchanged in all versions of the array. The second strategy is a generalized selection strategy, and appears to require $\Omega(c)$ time, where c is the total number of changes made by all threads.[1]

The third strategy can allow very fast merges, essentially requiring no time at all for the merge. In fact, if intrathread accesses are single-threaded, we can simply use an imperative array—tasks are prohibited from seeing the changes made by other threads. Ensuring that tasks play by the rules is a different matter, however, and it is exactly this question—how to ensure that tasks play by the rules—that is addressed by Part II of this dissertation.

The fourth strategy is interesting because it can be implemented efficiently using the fat-elements method; we will consider an instance of this strategy in the remainder of this appendix.

G.3 Fat-Elements Merge

Figure G.1 shows three threads, P, Q, and R, accessing a fat-element array. Each thread performs some element reads and then writes some elements. Thread P writes 97 into

1. Achieving a generalized selection-based merge in time proportional to the number of changes seems to be quite difficult, especially if multiple merges are allowed to occur in parallel and $c < n$.

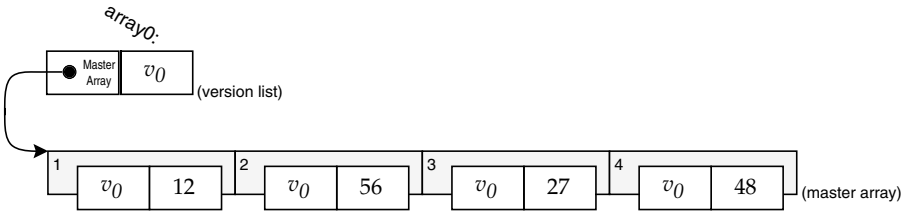
element 1; thread Q writes 131 and 6 into elements 2 and 4, respectively; and thread R writes 16 and 24 into elements 3 and 4, respectively. (None of the threads see the writes performed by the other threads, following the usual rules for fat-element arrays.) Let us suppose that we wish to merge the results of these threads, arrayP1, arrayQ1, and arrayR1, according to the rule that whenever more than one thread writes to the same location, the merged array should prefer a value written by thread P over thread Q, and prefer one written by Q over R.

One possible execution sequence is shown in Figure G.1, with P writing to the array before Q and Q making its first write before R. The pair of writes performed by Q share the same version stamp because they were single-threaded writes (see Chapter 5), as do the pair of writes performed by R.

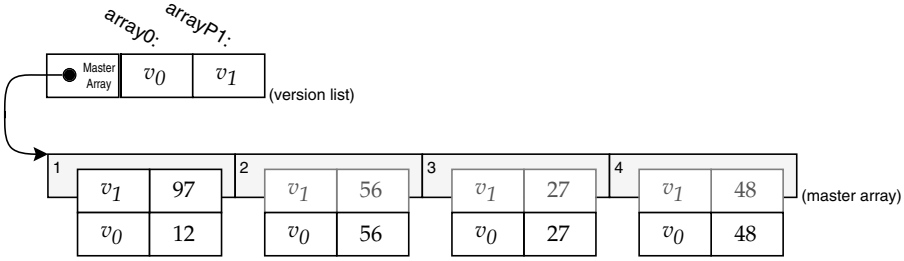
Figure G.2 shows two ways to achieve our desired merge result. Figure G.2(a) adds fat-element entries to create the merged master array, whereas the process illustrated by Figure G.2(b) deletes entries to achieve the same result. (Note that deleting entries is only acceptable if we can be sure that arrayP1, arrayQ1, and arrayR1 are no longer referenced and can therefore be destructively modified—for now we will make this assumption.)

The fat-element entries that need to be removed are exactly those elements that were added to avoid incorrect values being inferred in the data structure (see the discussion of fully persistent updates on page 32). With a small change to the initial setup, these entries are easy to identify, as shown in Figure G.3. Instead of updating the same initial array, each thread is given its own initial array. Although arrayR0 is equivalent to array0, the other two initial arrays, arrayQ0 and arrayP0, contain no changes—their only purpose is to act as separators between the master-array additions and hide changes made by other threads with earlier version stamps. After the threads have terminated, we only have to remove the values associated with these "separator" array versions and we have merged the array.

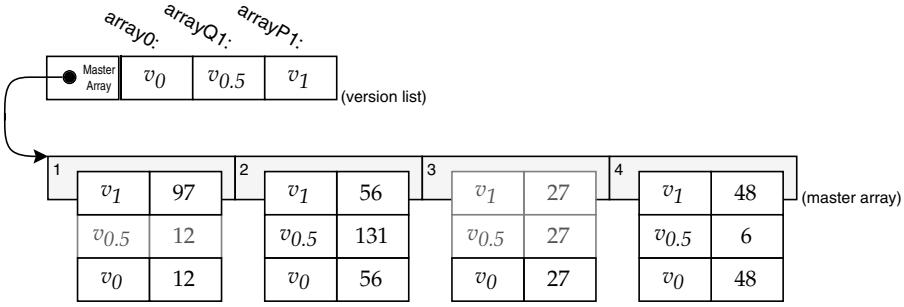
Traversing the array to find versions to delete is expensive (requiring $\Theta(n)$) time, so we do not actually delete these versions immediately. Instead, we simply mark the version stamps for separator versions as corresponding to deleted nodes. Then we perform deletion lazily as "deleted" nodes are encountered.



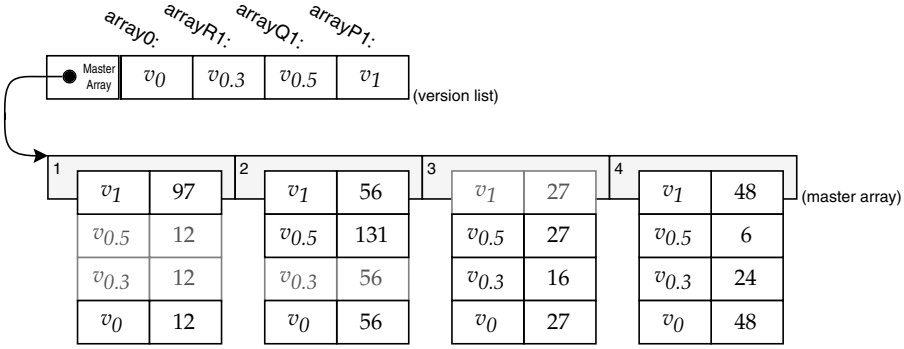
(a) An initial array, array0.



(b) Thread P updates one element of array0, creating arrayP1.

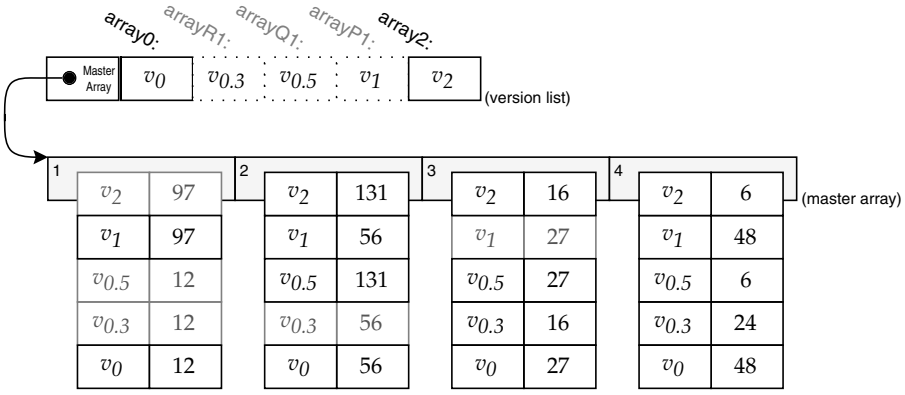


(c) Thread Q updates two elements of array0, creating arrayQ1.

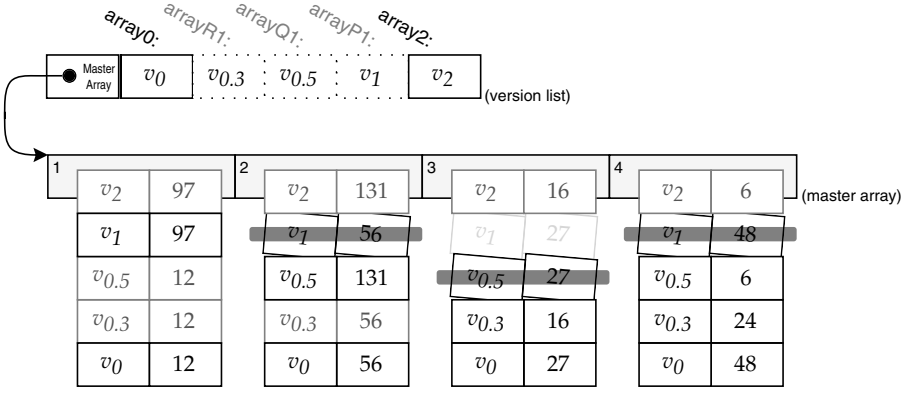


(d) Thread R updates two elements of array0, creating arrayR1.

Figure G.1: Three threads create three array versions.



(a) Using insertion to create the merged array, array2.



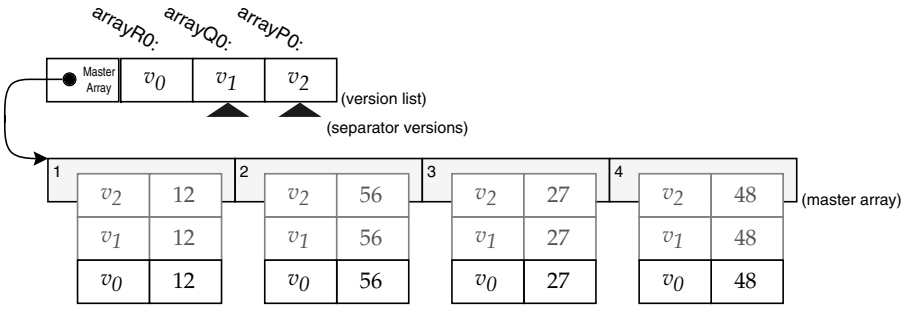
(b) Using deletion to create the merged array, array2.

Figure G.2: Creating a merged array.

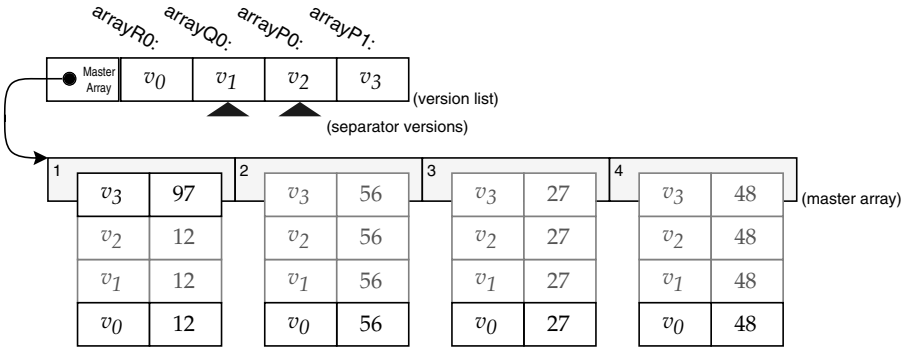
G.4 Issues and Limitations

Earlier, I stated that we would assume that the merge operation was free to destroy the array versions it was merging. But what should we do when this assumption does not hold? Sadly, I do not have a good answer to this question. None of the techniques I have investigated can provide both completely functional semantics and good performance.

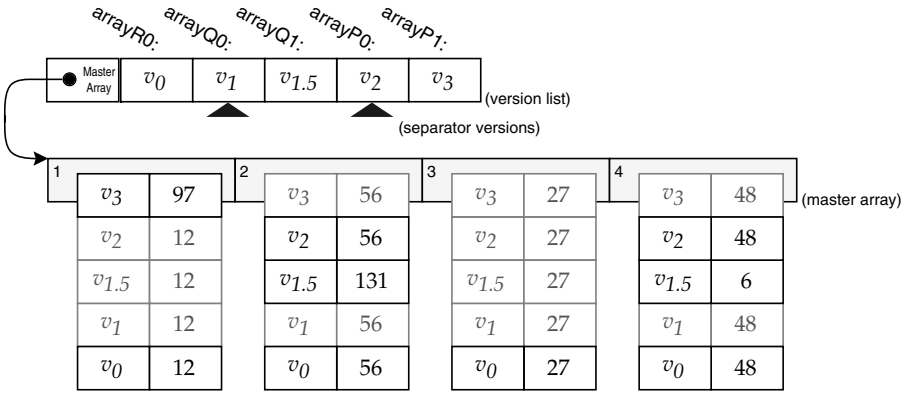
Although this strategy for merging arrays seems to be a dead end, the LR-tags method provides a mechanism that can be used to allow parallel tasks to share an array (or memory).



(a) Each thread has an initial array.

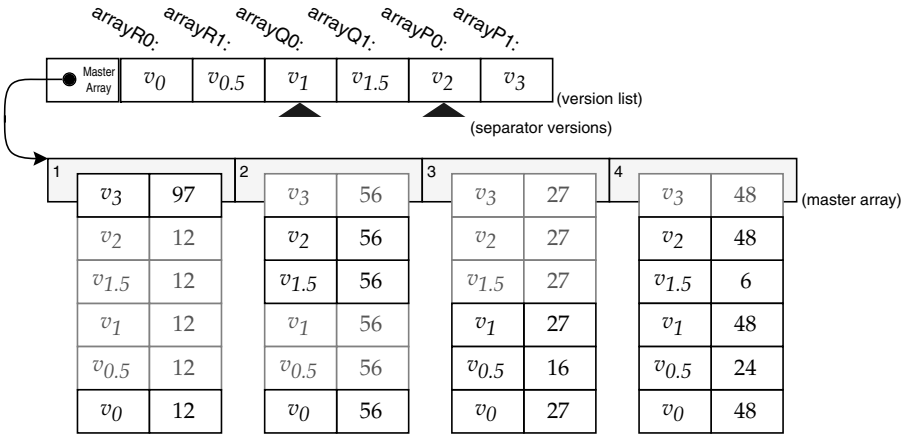


(b) Thread P updates one element of arrayP0, creating arrayP1.



(c) Thread Q updates two elements of arrayQ0, creating arrayQ1.

Figure G.3: Three threads each create separate array versions. (*continued over...*)



(d) Thread R updates two elements of arrayR0, creating arrayR1.

Figure G.3 (*cont.*): Three threads each create three separated array versions.

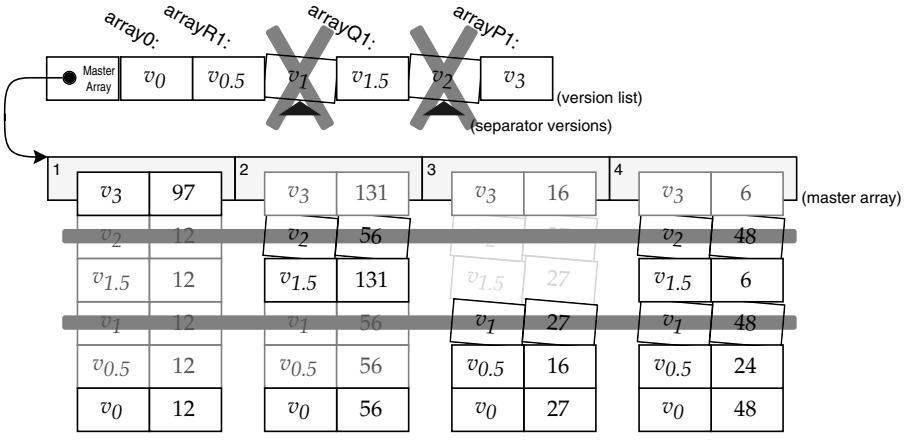


Figure G.4: Creating a merged array from separate array versions.

Appendix H

Obtaining Source Code for the Fat-Elements and LR-tags Methods

While writing this dissertation, I was faced with a dilemma: How should I provide readers with access to the source code for my implementations of the fat-elements and LR-tags methods. I could, perhaps, have typeset all the source code and included it as an appendix, but at 75,000 lines of C/C++ code for the LR-tags method, and 7000 lines of Standard ML code for the fat-elements method, adding such an appendix would have made this dissertation a weighty volume indeed. Even if I were to exclude library code written by others on which my work depends, there would still be more than 10,000 lines of code to present.[1]

Therefore, I am making my source code available electronically. At the time of writing, you can download the code (which is licensed under the GNU General Public License) using the following URL:

http://schoenfinkel.cs.sfu.ca/~oneill/thesis/

Unfortunately, whereas books are relatively persistent, the "World Wide Web" is an ephemeral structure,. Thus, if the URL above does not work, or URLs themselves have become an

1. About 48,000 lines of code in the LR-tags method come from Hans Boehm's garbage collector; and a further 10,000 are due to Doug Lea's malloc and its parallel derivative. Additionally, almost 7000 lines of code come from ported CILK benchmarks. The LR-tags algorithm itself consists of about 4800 lines of code; in addition, the SP-Bags algorithm is implemented in about 1000 lines and the COTTON/PTHIEVES parallel platform is a mere 2000 lines. Similarly, the Standard ML code includes about 400 lines of freely available library code.

anachronism, I can only suggest that you use whatever means at your disposal to search the information infrastructure for my name or the term "LR-tags method" in hopes of finding an archive of the source.

Sadly, my source code is never as thoroughly commented as I would like. If you have questions, feel free to contact me.

Bibliography

Annika Aasa, Sören Holmström, and Christina Nilsson, 1988. An Efficiency Comparison of Some Representations of Purely Functional Arrays. *BIT*, 28(3):490–503. [13, 21, 46, 55]

Peter M. Achten, John H. G. van Groningen, and [Rinus] M. J. Plasmeijer, 1993. High-Level Specification of I/O in Functional Languages. In *Proceedings of the Glasgow Workshop on Functional Programming*, edited by John Launchbury and Patrick M. Sansom, pp. 1–17. Springer Verlag. [9]

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali, 1989. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632. [85, 181]

Babak Bagheri, 1994. *Parallel Programming With Guarded Objects*. Ph.D. dissertation, The Pennsylvania State University. [88]

Henry G. Baker Jr., 1978. Shallow Binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569. [12, 21]

Henry G. Baker Jr., 1991. Shallow Binding Makes Functional Arrays Fast. *ACM SIGPLAN Notices*, 26(8):145–147. [13, 21, 23]

Vasanth Balasundaram and Ken Kennedy, 1989. Compile-Time Detection of Race Conditions in a Parallel Program. In *Proceedings of the Fourth International Conference on Supercomputing*, pp. 175–185. [88]

Adam Beguelin, 1990. *Deterministic Parallel Programming in Phred*. Ph.D. dissertation, University of Colorado at Boulder. [88]

Adam Beguelin and Gary Nutt, 1994. Visual Parallel Programming and Determinacy: A Language Specification, an Analysis Technique, and a Programming Tool. *Journal of Parallel and Distributed Computing*, 22(2):235–250. [88]

Arthur J. Bernstein, 1966. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763. [78, 85]

Garrett Birkhoff, 1967. *Lattice Theory*, vol. 25 of *Colloquium Publications*. American Mathematical Society, New York, third ed. [95]

Guy E. Blelloch and Perry Cheng, 1999. On Bounding Time and Space for Multiprocessor Garbage Collection. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 34(5):104–117, ACM Press. [162]

Adrienne Bloss, 1989. Update Analysis and the Efficient Implementation of Functional Aggregates. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM Press. [11, 16]

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, 1996. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69. [191, 216]

Robert D. Blumofe and Charles E. Leiserson, 1999. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748. [216, 219]

Robert D. Blumofe and Dionisios Papadopoulos, 1998. Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors. Tech. rep., Department of Computer Science, University of Texas at Austin. [216]

Hans-Juergen Boehm and Mark Weiser, 1988. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, 18(9):807–820. [162]

Per Brinch Hansen, 1994a. Interference Control in SuperPascal – A Block-Structured Parallel Language. *The Computer Journal*, 37(5):399–406. [87, 88]

Per Brinch Hansen, 1994b. SuperPascal—A Publication Language for Parallel Scientific Computing. *Concurrency: Practice and Experience*, 6(5):461–483. [87]

Gerth Stølting Brodal and Chris Okasaki, 1996. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6(6):839–857. [15]

F. Warren Burton, 1982. An Efficient Functional Implementation of FIFO Queues. *Information Processing Letters*, 14(5):205–206. [15]

David Callahan and Jaspal Subhlok, 1989. Static Analysis of Low-Level Synchronization. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*. ACM SIGPLAN Notices, 24(1):100–111, ACM Press. [88]

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark, 1998. Detecting Data Races in Cilk Programs that Use Locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pp. 298–309. ACM Press. [181, 241]

Jong-Deok Choi and Sang Lyul Min, 1991. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 26(7), pp. 145–154. ACM Press. [89]

Tyng-Ruey Chuang, 1992. Fully Persistent Arrays for Efficient Incremental Updates and Voluminous Reads. In *ESOP '92: 4th European Symposium on Programming*, edited by Bernd Krieg–Brückner, 582, pp. 110–129. Springer Verlag. [13, 22, 23, 49, 51, 55]

Tyng-Ruey Chuang, 1994. A Randomized Implementation of Multiple Functional Arrays. In *Conference Record of the 1994 ACM Conference on LISP and Functional Programming*, pp. 173–184. [13, 22, 49, 51, 55]

Tyng-Ruey Chuang and Benjamin Goldberg, 1993. Real-Time Deques, Multihead Turing Machines, and Purely Functional Programming. In *Proceedings of the Conference on*

Functional Programming Languages and Computer Architecture, pp. 289–298. ACM Press. [15]

Shimon Cohen, 1984. Multi-Version Structures in Prolog. In *International Conference on Fifth Generation Computer Systems*, pp. 265–274. [23, 51, 55]

Eric C. Cooper and Richard P. Draves, 1988. C Threads. Tech. Rep. CMU-CS-88-154, School of Computer Science, Carnegie Mellon University. [193]

Patrick Cousot and Radhia Cousot, 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 238–252. ACM Press. [11]

Patrick Cousot and Radhia Cousot, 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 269–282. ACM Press. [11]

Olivier Danvy and Andrzej Filinski, 1972. Abstracting Control. In *Conference on Lisp and Functional Programming*, pp. 717–740. [216]

Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis, 1999. *Graph Drawing: Algorithms for the Visualization of Graphs*, chap. 4. Prentice Hall. [ix, 95, 97, 155]

Paul F. Dietz, 1989. Fully Persistent Arrays (Extended Abstract). In *Proceedings of the Workshop on Algorithms and Data Structures (WADS '89)*, edited by Frank Dehne, Joerg-Ruediger Sack, and Nicola Santoro, 382, pp. 67–74. Springer Verlag. [22, 51, 55]

Paul F. Dietz and Daniel D. K. Sleator, 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. A revised version of this paper is available as a University of Rochester technical report. [27, 103, 150, 182, 183, 229, 231, 235, 249, 250]

Edsger Wybe Dijkstra, 1968. Co-operating Sequential Processes. In *Programming Languages*, edited by F. Genuys, pp. 43–112. Academic Press. [77]

Anne Dinning and Edith Schonberg, 1990. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pp. 1–10. ACM Press. [78, 89]

Anne Dinning and Edith Schonberg, 1991. Detecting Access Anomalies in Programs with Critical Sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM SIGPLAN Notices, 26(12):85–96. [89]

James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert Endre Tarjan, 1989. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124. [4, 14, 23, 25, 51]

Richard J. Duffin, 1965. Topology of Series-Parallel Networks. *Journal of Mathematical Analysis and Applications*, 10:303–318. [84]

Hartmut Ehrig, 1978. Introduction to the Algebraic Theory of Graph Grammars. In *Graph Grammars and their Application to Computer Science and Biology: International Workshop*, edited by Volker Claus, Hartmut Ehrig, and Gregor Rozenberg, 73, pp. 1–69. Springer Verlag. [251]

Perry A. Emrath and Davis A. Padua, 1988. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pp. 89–99. [79, 88, 89]

Toshio Endo, Kenjiro Taura, and Akinori Yonezawa, 1997. A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. In *Proceedings of the 1997 ACM/IEEE Supercomputing Conference: High Performance Networking and Computing (SC '97)*, edited by ACM. ACM Press and IEEE Computer Society Press. [162]

Lars-Henrik Eriksson and Manny Rayner, 1984. Incorporating Mutable Arrays into Logic Programming. In *Proceedings of the Second International Conference on Logic Programming*, edited by Sten-Åke Tärnlund, pp. 101–114. Ord & Form. [21]

Mingdong Feng and Charles E. Leiserson, 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on*

Parallel Algorithms and Architectures (SPAA '97), pp. 1–11. ACM Press, New York. [xxix, 79, 89, 90, 93, 166, 184, 258]

Michael J. Flynn, 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960. [80]

Matteo Frigo and Steven G. Johnson, 1997. The Fastest Fourier Transform in the West. Technical Report MIT/LCS/TR-728, Massachusetts Institute of Technology. [166]

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, 33(5), pp. 212–223. [166, 191, 216]

Harold N. Gabow and Robert Endre Tarjan, 1985. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221. [251]

David Gries, 1981. *The Science of Programming*. Texts and Monographs in Computer Science, Springer Verlag. [15]

Charles Antony Richard Hoare, 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557. [77]

Charles Antony Richard Hoare, 1985. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, Prentice Hall International. [85]

Sören Holmström, 1983. How to Handle Large Data Structures in Functional Languages. In *Proceedings of the SERC Chalmers Workshop on Declarative Programming Languages*. [21]

Robert T. Hood, 1982. The Efficient Implementation of Very-High-Level Programming Language Constructs. Technical Report TR82-503, Computer Science Department, Cornell University. [15]

Robert T. Hood and R. Melville, 1981. Real-Time Queue Operations in Pure Lisp. *Information Processing Letters*, 13(2):50–54. [15]

Robert R. Hoogerwoord, 1992a. A Logarithmic Implementation of Flexible Arrays. In *Proceedings of the 2nd International Conference on Mathematics of Program Construction*, 669, pp. 191–207. Springer Verlag. [20, 46]

Robert R. Hoogerwoord, 1992b. A Symmetric Set of Efficient List Operations. *Journal of Functional Programming*, 2(4):505–513. [15]

Paul R. Hudak, 1992a. Continuation-Based Mutable Abstract Datatypes. Tech. Rep. YALEU/DCS/RR-914, Department of Computer Science, Yale University. [8]

Paul R. Hudak, 1992b. Mutable Abstract Datatypes – or – How to Have Your State and Munge it Too. Tech. rep., Yale University. [8]

Paul R. Hudak and Adrienne Bloss, 1985. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*. ACM Press. [11]

Paul R. Hudak, Simon L. Peyton Jones, Philip L. Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur S. Nikhil, William D. Partain, and John Peterson, 1992. Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5). [8, 16]

John Hughes, 1985. An Efficient Implementation of Purely Functional Arrays. Tech. rep., Department of Computer Sciences, Chalmers University of Technology. [21, 55]

H. S. Huitema and [Rinus] M. J. Plasmeijer, 1992. The Concurrent Clean System Users Manual, Version 0.8. Tech. Rep. 92-19, University of Nijmegen. [9]

IEEE, 1995. *IEEE 1003.1c-1995: Information Technology – Portable Operating System Interface (Posix) – System Application Program Interface (API) Amendment 2: Threads Extension (C Language).* IEEE Computer Society Press. [192, 193]

ISO, 1995. *ISO/IEC 8652:1995: Information Technology – Programming Languages – Ada.* International Organization for Standardization. [192]

ISO, 1999. *ISO/IEC 9899:1999: Programming Languages – C*. International Organization for Standardization. [194]

Richard Jones, 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley & Sons. [51]

Simon B. Jones and Daniel Le Métayer, 1989. Compile-Time Garbage Collection by Sharing Analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 54–74. ACM Press. [11]

Bill Joy, Guy L. Steele, Jr, James Gosling, and Gilad Bracha, 2000. *The Java Language Specification*. The Java Series, Addison-Wesley, second ed. [192]

Tiko Kameda, 1975. On the Vector Representation of the Reachability in Planar Directed Graphs. *Information Processing Letters*, 3(3):75–77. [95]

Haim Kaplan and Robert Endre Tarjan, 1996. Purely Functional Representations of Catenable Sorted Lists. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pp. 202–211. [15]

Gregory Maxwell Kelly and Ivan Rival, 1975. Planar Lattices. *Canadian Journal of Mathematics*, 27(3):636–665. [95, 97]

David J. King, 1994. Functional Binomial Queues. In *Glasgow Functional Programming Workshop*, edited by Kevin Hammond, David N. Turner, and Patrick M. Sansom. Springer Verlag. [15]

Joachim Lambek and Philip J. Scott, 1986. *Introduction to Higher Order Categorical Logic*. Cambridge University Press. [8]

John Launchbury and Simon L. Peyton Jones, 1995. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341. [8, 9]

Doug Lea, 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Java Grande Conference*. ACM Press. [215]

Saunders MacLane, 1971. *Categories for the Working Mathematician*. Springer Verlag. [8]

Percy Alexander MacMahon, 1892. The Combination of Resistances. *The Electrician*. [84]

John McCarthy, 1963. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg, pp. 33–70. North-Holland. [79]

John Mellor-Crummey, 1991. On-the-Fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing '91*, pp. 24–33. IEEE Computer Society Press. [78, 89]

Barton P. Miller and Jong-Deok Choi, 1989. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1), pp. 141–150. ACM Press. [89]

Robin Milner, 1989. *Communication and Concurrency*. Prentice Hall International Series in Computer Science, Prentice Hall International. [85]

Robin Milner, Mads Tofte, and Robert Harper, 1990. *The Definition of Standard ML*. MIT Press. [10]

Sang Lyul Min and Jong-Deok Choi, 1991. An Efficient Cache-Based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 235–244. [89]

Markus Mohnen, 1995. Efficient Compile-Time Garbage Collection for Arbitrary Data Structures. In *Proceedings of the Seventh International Symposium on Programming Language Implementation and Logic Programming (PLILP '95)*, edited by M. Hermenegildo and S.D. Swierstra, pp. 241–258. Springer Verlag. [11]

Graeme E. Moss, 1996. Purely Functional Data Structures. Ph.D. qualifying dissertation, York University, UK. [15]

Eugene W. Myers, 1984. Efficient Applicative Data Types. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, edited by Ken Kennedy, pp. 66–75. ACM Press. [20, 35]

Girija J. Narlikar and Guy E. Blelloch, 1998. Pthreads for Dynamic and Irregular Parallelism. In *Proceedings of Supercomputing '98 (CD-ROM)*. ACM SIGARCH and IEEE. Carnegie Mellon University. [216, 225]

Greg Nelson (ed.), 1991. *Systems Programming with Modula-3*. Prentice Hall. [192]

Robert H. B. Netzer and Barton P. Miller, 1992. What Are Race Conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88. [78]

Itzhak Nudler and Larry Rudolph, 1986. Tools for the Efficient Development of Efficient Parallel Programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*. [78, 89]

Chris Okasaki, 1995. Simple and Efficient Purely Functional Queues and Deques. *Journal of Functional Programming*, 5(4):583–592. [15]

Chris Okasaki, 1996. *Purely Functional Data Structures*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University. [15]

Chris Okasaki, 1997. Catenable Double-Ended Queues. In *International Conference on Functional Programming*. [15]

Chris Okasaki and Gaurav S. Kc, 2000. Wide Trees: A Simple Implementation of Persistent Arrays. Working Paper. Department of Computer Science, Columbia University. [20, 23]

Melissa E. O'Neill, 1994. *A Data Structure for More Efficient Runtime Support of Truly Functional Arrays*. M.Sc. thesis, School of Computing Science, Simon Fraser University. [3, 23, 46]

Melissa E. O'Neill and F. Warren Burton, 1997. A New Method for Functional Arrays. *Journal of Functional Programming*, 7(5):487–514. [xxvii, 3, 229]

Mark H. Overmars, 1981. Searching in the Past II: General Transforms. Tech. Rep. RUU–CS–81–9, Department of Computer Science, University of Utrecht. [51]

Mark H. Overmars, 1983. *The Design of Dynamic Data Structures*, vol. 156 of *Lecture Notes in Computer Science*, pp. 153–157. Springer Verlag. [12, 51]

Lawrence C Paulson, 1991. *ML for the Working Programmer*. Cambridge University Press. [46]

Josip E. Pecaric, Frank Proschan, and Yung Liang Tong, 1993. *Convex Functions, Partial Orderings, and Statistical Applications*, vol. 187 of *Mathematics in Science and Engineering*. Academic Press. [42]

Simon L. Peyton Jones, 1987. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science, Prentice Hall. [251]

Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, William D. Partain, and Philip L. Wadler, 1993. The Glasgow Haskell Compiler: A Technical Overview. In *Proceedings of the UK Joint Framework for Information Technology Technical Conference*, pp. 249–257. [10]

Simon L. Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, F. Warren Burton, Joseph H. Fasel, Kevin Hammond, Ralf Hinze, Paul R. Hudak, Thomas Johnsson, Mark P. Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip L. Wadler, 1999. Haskell 98: A Non-Strict, Purely Functional Language. Tech. rep., Yale University. [8]

Willard Van Orman Quine, 1960. *Word and Object*. Harvard University Press. [10]

John C. Reynolds, 1990. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the 25th ACM National Conference*, pp. 717–740. ACM. [216]

John Riordan and Claude Elwood Shannon, 1942. The Number of Two-Terminal Series-Parallel Networks. *Journal of Mathematics and Physics*, 21:83–93. [84]

Neil Sarnak and Robert Endre Tarjan, 1986. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29(7):669–679. [6]

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411. [181, 241]

David A. Schmidt, 1985. Detecting Global Variables in Denotational Specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310. [4]

Edith Schonberg, 1989. On-the-Fly Detection of Access Anomalies. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 24(7):285–297. [89]

Lenhart K. Schubert, Mary Angela Papalskaris, and Jay Taugher, 1983. Determining Type, Part, Color, and Time Relationships. *IEEE Computer*, pp. 53–60. [247]

David J. Simpson and F. Warren Burton, 1999. Space-Efficient Execution of Deterministic Parallel Programs. *IEEE Transactions on Software Engineering*, 25(6):870–882. [216]

Daniel D. K. Sleator and Robert Endre Tarjan, 1985. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686. [29, 39]

Harald Søndergaard and Peter Sestoft, 1990. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517. [10]

Guy L. Steele, Jr, 1990. Making Asynchronous Parallelism Safe for the World. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pp. 218–231. ACM Press. [78, 79, 84, 86, 89]

William R. Stoye, Thomas J. W. Clarke, and Arthur C. Norman, 1984. Some Practical Methods for Rapid Combinator Reduction. In *ACM Symposium on LISP and Functional Programming*, pp. 159–166. ACM Press. [12]

Robert Endre Tarjan, 1975. Efficiency of a Good but Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225. [87, 90, 251, 256]

Robert Endre Tarjan, 1979. Applications of Path Compression on Balanced Trees. *Journal of the ACM*, 26(4):690–715. [90]

Robert Endre Tarjan, 1983. *Data Structures and Network Algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM. [20]

Robert Endre Tarjan, 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318. [37]

Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa, 1999. StackThreads/MP: Integrating Futures into Calling Standards. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99)*, edited by A. Andrew Chien and Marc Snir. ACM SIGPLAN Notices, 34(8):60–71, ACM Press. [216]

Richard N. Taylor, 1983. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376. [88]

Athanasios K. Tsakalidis, 1984. Maintaining Order in a Generalized Linked List. *Acta Informatica*, 21(1):101–112. [27, 103, 229, 231, 249, 250]

David A. Turner, 1979. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49. [251]

David A. Turner, 1986. An Overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166. [20]

Jacobo Valdes, 1978. *Parsing Flowcharts and Series-Parallel Graphs*. Ph.D. dissertation, Stanford University. [80]

Philip L. Wadler, 1990a. Comprehending Monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. ACM Press. [8]

Philip L. Wadler, 1990b. Linear Types Can Change the World! In *Programming Concepts and Methods*, edited by M. Broy and C. Jones. North-Holland. [9]

Philip L. Wadler, 1992. The Essence of Functional Programming. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*. ACM Press. [8]

Paul R. Wilson, 1994. Uniprocessor Garbage Collection Techniques. Tech. rep., University of Texas. [51]

David S. Wise and Daniel P. Friedman, 1977. The One-Bit Reference Count. *BIT*, 17(3):351–359. [12]

Zhiwei Xu and Kai Hwang, 1992. Language Constructs for Structured Parallel Programming. In *Proceedings of the Sixth International Parallel Processing Symposium*, edited by V.K. Prasanna and L.H. Canter, pp. 454–461. IEEE Computer Society Press. [88]

Colophon

This dissertation was typeset with the TEX typesetting system and the LATEX 2_ε and $\mathcal{AMS}$-LATEX macros, with a slightly modified copy of sfu-thesis.cls and various custom macros. The text is set in Adobe Kepler MM and Adobe Myriad MM. Both typefaces are multiple-master fonts, allowing selected attributes of the typeface to be varied continuously—for example, 13 distinct weight/width/optical-size settings are employed by the 33 Kepler fonts used in the text. The font metrics for Kepler and Myriad were automatically generated by FontKit, a custom TEX–font-metrics tool I wrote (in Perl), driving Eddie Kohler's mmafm tool to perform the necessary font-metrics interpolation. Because of the vagaries of configuring math fonts for use with TEX, mathematical content uses Computer Modern and Euler.

All of the diagrams in this dissertation were created using Lighthouse Design's Diagram. The LR-graphs in Chapter 8 were generated with the aid of a custom-written Perl program, ddraw, which created files for Diagram (thanks to the open file-format specification provided by Lighthouse Design). Performance graphs were created using Rüdiger Brühl's Abscissa. Both applications run under NextStep.

The text was edited on aldrington, a NextStation running NextStep 3.3. This machine is older than the dissertation itself by several years, yet even as technology moves forward, it is worth remembering that what matters most is not how new a technology is, but how useful it is.