Purely Functional Data Structures

Chris Okasaki

September 1996
CMU-CS-96-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the Advanced Research Projects Agency (ARPA) under Contract No. F19628-95-C-0050.

Keywords: functional programming, data structures, lazy evaluation, amortization

For Maria

Abstract

When a C programmer needs an efficient data structure for a particular problem, he or she can often simply look one up in any of a number of good textbooks or handbooks. Unfortunately, programmers in functional languages such as Standard ML or Haskell do not have this luxury. Although some data structures designed for imperative languages such as C can be quite easily adapted to a functional setting, most cannot, usually because they depend in crucial ways on assignments, which are disallowed, or at least discouraged, in functional languages. To address this imbalance, we describe several techniques for designing functional data structures, and numerous original data structures based on these techniques, including multiple variations of lists, queues, double-ended queues, and heaps, many supporting more exotic features such as random access or efficient catenation.

In addition, we expose the fundamental role of lazy evaluation in *amortized* functional data structures. Traditional methods of amortization break down when old versions of a data structure, not just the most recent, are available for further processing. This property is known as *persistence*, and is taken for granted in functional languages. On the surface, persistence and amortization appear to be incompatible, but we show how lazy evaluation can be used to resolve this conflict, yielding amortized data structures that are efficient even when used persistently. Turning this relationship between lazy evaluation and amortization around, the notion of amortization also provides the first practical techniques for analyzing the time requirements of non-trivial lazy programs.

Finally, our data structures offer numerous hints to programming language designers, illustrating the utility of combining strict and lazy evaluation in a single language, and providing non-trivial examples using polymorphic recursion and higher-order, recursive modules.

Acknowledgments

Without the faith and support of my advisor, Peter Lee, I probably wouldn't even be a graduate student, much less a graduate student on the eve of finishing. Thanks for giving me the freedom to turn my hobby into a thesis.

I am continually amazed by the global nature of modern research and how e-mail allows me to interact as easily with colleagues in Aarhus, Denmark and York, England as with fellow students down the hall. In the case of one such colleague, Gerth Brodal, we have co-authored a paper without ever having met. In fact, sorry Gerth, but I don't even know how to pronounce your name!

I was extremely fortunate to have had excellent English teachers in high school. Lori Huenink deserves special recognition; her writing and public speaking classes are undoubtedly the most valuable classes I have ever taken, in any subject. In the same vein, I was lucky enough to read my wife's copy of Lyn Dupré's *BUGS in Writing* just as I was starting my thesis. If your career involves writing in any form, you owe it to yourself to buy a copy of this book.

Thanks to Maria and Colin for always reminding me that there is more to life than grad school. And to Amy and Mark, for uncountable dinners and other outings. We'll miss you. Special thanks to Amy for reading a draft of this thesis.

And to my parents: who would have thought on that first day of kindergarten that I'd still be in school 24 years later?

Contents

Chapter 1

Introduction

Efficient data structures have been studied extensively for over thirty years, resulting in a vast literature from which the knowledgeable programmer can extract efficient solutions to a stunning variety of problems. Much of this literature purports to be language-independent, but unfortunately it is language-independent only in the sense of Henry Ford: Programmers can use any language they want, as long as it's imperative.[1] Only a small fraction of existing data structures are suitable for implementation in functional languages, such as Standard ML or Haskell. This thesis addresses this imbalance by specifically considering the design and analysis of functional data structures.

1.1 Functional vs. Imperative Data Structures

The methodological benefits of functional languages are well known [Bac78, Hug89, HJ94], but still the vast majority of programs are written in imperative languages such as C. This apparent contradiction is easily explained by the fact that functional languages have historically been slower than their more traditional cousins, but this gap is narrowing. Impressive advances have been made across a wide front, from basic compiler technology to sophisticated analyses and optimizations. However, there is one aspect of functional programming that no amount of cleverness on the part of the compiler writer is likely to mitigate — the use of inferior or inappropriate data structures. Unfortunately, the existing literature has relatively little advice to offer on this subject.

Why should functional data structures be any more difficult to design and implement than imperative ones? There are two basic problems. First, from the point of view of designing and implementing efficient data structures, functional programming's stricture against destructive

[1]Henry Ford once said of the available colors for his Model T automobile, "[Customers] can have any color they want, as long as it's black."

updates (assignments) is a staggering handicap, tantamount to confiscating a master chef's knives. Like knives, destructive updates can be dangerous when misused, but tremendously effective when used properly. Imperative data structures often rely on assignments in crucial ways, and so different solutions must be found for functional programs.

The second difficulty is that functional data structures are expected to be more flexible than their imperative counterparts. In particular, when we update an imperative data structure we typically accept that the old version of the data structure will no longer be available, but, when we update a functional data structure, we expect that both the old and new versions of the data structure will be available for further processing. A data structure that supports multiple versions is called *persistent* while a data structure that allows only a single version at a time is called *ephemeral* [DSST89]. Functional programming languages have the curious property that *all* data structures are automatically persistent. Imperative data structures are typically ephemeral, but when a persistent data structure is required, imperative programmers are not surprised if the persistent data structure is more complicated and perhaps even asymptotically less efficient than an equivalent ephemeral data structure.

Furthermore, theoreticians have established lower bounds suggesting that functional programming languages may be fundamentally less efficient than imperative languages in some situations [BAG92, Pip96]. In spite of all these points, this thesis shows that it is often possible to devise functional data structures that are asymptotically as efficient as the best imperative solutions.

1.2 Strict vs. Lazy Evaluation

Most (sequential) functional programming languages can be classified as either *strict* or *lazy*, according to their order of evaluation. Which is superior is a topic debated with religious fervor by functional programmers. The difference between the two evaluation orders is most apparent in their treatment of arguments to functions. In strict languages, the arguments to a function are evaluated before the body of the function. In lazy languages, arguments are evaluated in a demand-driven fashion; they are initially passed in unevaluated form and are evaluated only when (and if!) the computation needs the results to continue. Furthermore, once a given argument is evaluated, the value of that argument is cached so that if it is ever needed again, it can be looked up rather than recomputed. This caching is known as *memoization* [Mic68].

Each evaluation order has its advantages and disadvantages, but strict evaluation is clearly superior in at least one area: ease of reasoning about asymptotic complexity. In strict languages, exactly which subexpressions will be evaluated, and when, is for the most part syntactically apparent. Thus, reasoning about the running time of a given program is relatively straightforward. However, in lazy languages, even experts frequently have difficulty predicting when, or even if, a given subexpression will be evaluated. Programmers in such languages

Name	Running Times of Supported Functions	Page
banker's queues	$snoc/head/tail$: $O(1)$	26
physicist's queues	$snoc/head/tail$: $O(1)$	31
real-time queues	$snoc/head/tail$: $O(1)^\dagger$	43
bootstrapped queues	$head$: $O(1)^\dagger$, $snoc/tail$: $O(\log^* n)$	89
implicit queues	$snoc/head/tail$: $O(1)$	113
banker's deques	$cons/head/tail/snoc/last/init$: $O(1)$	56
real-time deques	$cons/head/tail/snoc/last/init$: $O(1)^\dagger$	59
implicit deques	$cons/head/tail/snoc/last/init$: $O(1)$	116
catenable lists	$cons/snoc/head/tail/+\!\!+$: $O(1)$	97
simple catenable deques	$cons/head/tail/snoc/last/init$: $O(1)$, $+\!\!+$: $O(\log n)$	119
catenable deques	$cons/head/tail/snoc/last/init/+\!\!+$: $O(1)$	122
skew-binary random-access lists	$cons/head/tail$: $O(1)^\dagger$, $lookup/update$: $O(\log n)^\dagger$	79
skew binomial heaps	$insert$: $O(1)^\dagger$, $merge/findMin/deleteMin$: $O(\log n)^\dagger$	83
bootstrapped heaps	$insert/merge/findMin$: $O(1)^\dagger$, $deleteMin$: $O(\log n)^\dagger$	102
sortable collections	add: $O(\log n)$, $sort$: $O(n)$	35
scheduled sortable collections	add: $O(\log n)^\dagger$, $sort$: $O(n)^\dagger$	47

Worst-case running times marked with †. All other running times are amortized.

Table 1.1: Summary of Implementations

are often reduced to pretending the language is actually strict to make even gross estimates of running time!

Both evaluation orders have implications for the design and analysis of data structures. As we will see in Chapters 3 and 4, strict languages can describe worst-case data structures, but not amortized ones, and lazy languages can describe amortized data structures, but not worst-case ones. To be able to describe both kinds of data structures, we need a programming language that supports both evaluation orders. Fortunately, combining strict and lazy evaluation in a single language is not difficult. Chapter 2 describes **$**-notation — a convenient way of adding lazy evaluation to an otherwise strict language (in this case, Standard ML).

1.3 Contributions

This thesis makes contributions in three major areas:

- *Functional programming.* Besides developing a suite of efficient data structures that are useful in their own right (see Table 1.1), we also describe general approaches to designing and analyzing functional data structures, including powerful new techniques for reasoning about the running time of lazy programs.

- *Persistent data structures.* Until this research, it was widely believed that amortization was incompatible with persistence [DST94, Ram92]. However, we show that memoization, in the form of lazy evaluation, is the key to reconciling the two. Furthermore, as noted by Kaplan and Tarjan [KT96b], functional programming is a convenient medium for developing new persistent data structures, even when the data structure will eventually be implemented in an imperative language. The data structures and techniques in this thesis can easily be adapted to imperative languages for those situations when an imperative programmer needs a persistent data structure.

- *Programming language design.* Functional programmers have long debated the relative merits of strict and lazy evaluation. This thesis shows that both are algorithmically important and suggests that the ideal functional language should seamlessly integrate both. As a modest step in this direction, we propose **$**-notation, which allows the use of lazy evaluation in a strict language with a minimum of syntactic overhead.

1.4 Source Language

All source code will be presented in Standard ML [MTH90], extended with primitives for lazy evaluation. However, the algorithms can all easily be translated into any other functional language supporting both strict and lazy evaluation. Programmers in functional languages that are either entirely strict or entirely lazy will be able to use some, but not all, of the data structures in this thesis.

In Chapters 7 and 8, we will encounter several recursive data structures that are difficult to describe cleanly in Standard ML because of the language's restrictions against certain sophisticated and difficult-to-implement forms of recursion, such as polymorphic recursion and recursive modules. When this occurs, we will first sacrifice executability for clarity and describe the data structures using ML-like pseudo-code incorporating the desired forms of recursion. Then, we will show how to convert the given implementations to legal Standard ML. These examples should be regarded as challenges to the language design community to provide a programming language capable of economically describing the appropriate abstractions.

1.5 Terminology

Any discussion of data structures is fraught with the potential for confusion, because the term *data structure* has at least four distinct, but related, meanings.

- *An abstract data type (that is, a type and a collection of functions on that type)*. We will refer to this as an *abstraction*.
- *A concrete realization of an abstract data type*. We will refer to this as an *implementation,* but note that an implementation need not be actualized as code — a concrete design is sufficient.
- *An instance of a data type, such as a particular list or tree*. We will refer to such an instance generically as an *object* or a *version*. However, particular data types typically have their own nomenclature. For example, we will refer to stack or queue objects simply as stacks or queues.
- *A unique identity that is invariant under updates*. For example, in a stack-based interpreter, we often speak informally about "the stack" as if there were only one stack, rather than different versions at different times. We will refer to this identity as a *persistent identity*. This issue mainly arises in the context of persistent data structures; when we speak of different versions of the same data structure, we mean that the different versions share a common persistent identity.

Roughly speaking, abstractions correspond to signatures in Standard ML, implementations to structures or functors, and objects or versions to values. There is no good analogue for persistent identities in Standard ML.[2]

The term *operation* is similarly overloaded, meaning both the functions supplied by an abstract data type and applications of those functions. We reserve the term *operation* for the latter meaning, and use the terms *operator* or *function* for the former.

1.6 Overview

This thesis is structured in two parts. The first part (Chapters 2–4) concerns algorithmic aspects of lazy evaluation. Chapter 2 sets the stage by briefly reviewing the basic concepts of lazy evaluation and introducing **$**-notation.

Chapter 3 is the foundation upon which the rest of the thesis is built. It describes the mediating role lazy evaluation plays in combining amortization and persistence, and gives two methods for analyzing the amortized cost of data structures implemented with lazy evaluation.

Chapter 4 illustrates the power of combining strict and lazy evaluation in a single language. It describes how one can often derive a worst-case data structure from an amortized data structure by systematically scheduling the premature execution of lazy components.

[2]The persistent identity of an ephemeral data structure can be reified as a reference cell, but this is insufficient for modelling the persistent identity of a persistent data structure.

The second part of the thesis (Chapters 5–8) concerns the design of functional data structures. Rather than cataloguing efficient data structures for every purpose (a hopeless task!), we instead concentrate on a handful of general techniques for designing efficient functional data structures and illustrate each technique with one or more implementations of fundamental abstractions such as priority queues, random-access structures, and various flavors of sequences.

Chapter 5 describes *lazy rebuilding*, a lazy variant of *global rebuilding* [Ove83]. Lazy rebuilding is significantly simpler than global rebuilding, but yields amortized rather than worst-case bounds. By combining lazy rebuilding with the scheduling techniques of Chapter 4, the worst-case bounds can be recovered.

Chapter 6 explores *numerical representations*, implementations designed in analogy to representations of numbers (typically binary numbers). In this model, designing efficient insertion and deletion routines corresponds to choosing variants of binary numbers in which adding or subtracting one take constant time.

Chapter 7 examines *data-structural bootstrapping* [Buc93]. Data-structural bootstrapping comes in two flavors: *structural decomposition*, in which unbounded solutions are bootstrapped from bounded solutions, and *structural abstraction*, in which efficient solutions are bootstrapped from inefficient solutions.

Chapter 8 describes *implicit recursive slowdown*, a lazy variant of the *recursive-slowdown* technique of Kaplan and Tarjan [KT95]. As with lazy rebuilding, implicit recursive slowdown is significantly simpler than recursive slowdown, but yields amortized rather than worst-case bounds. Again, we can recover the worst-case bounds using scheduling.

Finally, Chapter 9 concludes by summarizing the implications of this work on functional programming, on persistent data structures, and on programming language design, and by describing some of the open problems related to this thesis.

Chapter 2

Lazy Evaluation and $-Notation

Lazy evaluation is an evaluation strategy employed by many purely functional programming languages, such as Haskell [H$^+$92]. This strategy has two essential properties. First, the evaluation of a given expression is delayed, or *suspended*, until its result is needed. Second, the first time a suspended expression is evaluated, the result is *memoized* (i.e., cached) so that the next time it is needed, it can be looked up rather than recomputed.

Supporting lazy evaluation in a strict language such as Standard ML requires two primitives: one to suspend the evaluation of an expression and one to resume the evaluation of a suspended expression (and memoize the result). These primitives are often called $delay$ and $force$. For example, Standard ML of New Jersey offers the following primitives for lazy evaluation:

```
type α susp
val delay : (unit → α) → α susp
val force : α susp → α
```

These primitives are sufficient to encode all the algorithms in this thesis. However, programming with these primitives can be rather inconvenient. For instance, to suspend the evaluation of some expression e, one writes $delay$ (fn () $\Rightarrow$ e). Depending on the use of whitespace, this introduces an overhead of 13–17 characters! Although acceptable when only a few expressions are to be suspended, this overhead quickly becomes intolerable when many expressions must be delayed.

To make suspending an expression as syntactically lightweight as possible, we instead use **$**-notation — to suspend the evaluation of some expression e, we simply write **$**$e$. **$**e is called a *suspension* and has type τ $susp$, where τ is the type of e. The scope of the **$** operator extends as far to the right as possible. Thus, for example, **$**$f\ x$ parses as **$**($f\ x$) rather than (**$**f) x and **$**$x$+$y$ parses as **$**($x$+$y$) rather than (**$**x)+y. Note that **$**$e$ is itself an expression and can be suspended by writing **$$**e, yielding a nested suspension of type τ $susp$ $susp$.

If s is a suspension of type $\tau\ susp$, then $force\ s$ evaluates and memoizes the contents of s and returns the resulting value of type τ. However, explicitly forcing a suspension with a $force$ operation can also be inconvenient. In particular, it often interacts poorly with pattern matching, requiring a single $case$ expression to be broken into two or more nested $case$ expressions, interspersed with $force$ operations. To avoid this problem, we integrate **$**-notation with pattern matching. Matching a suspension against a pattern of the form **$**$p$ first forces the suspension and then matches the result against p. At times, an explicit $force$ operator is still useful. However, it can now be defined in terms of **$** patterns.

```
fun force ($x) = x
```

To compare the two notations, consider the standard $take$ function, which extracts the first n elements of a stream. Streams are defined as follows:

```
datatype α StreamCell = Nil | Cons of α × α Stream
withtype α Stream = α StreamCell susp
```

Using $delay$ and $force$, $take$ would be written

```
fun take (n, s) =
      delay (fn () ⇒ case n of
                        0 ⇒ Nil
                      | _ ⇒ case force s of
                              Nil ⇒ Nil
                            | Cons (x, s′) ⇒ Cons (x, take (n−1, s′)))
```

In contrast, using **$**-notation, $take$ can be written more concisely as

```
fun take (n, s) = $case (n, s) of
                       (0, _) ⇒ Nil
                     | (_, $Nil) ⇒ Nil
                     | (_, $Cons (x, s′)) ⇒ Cons (x, take (n−1, s′))
```

In fact, it is tempting to write $take$ even more concisely as

```
fun take (0, _) = $Nil
  | take (_, $Nil) = $Nil
  | take (n, $Cons (x, s)) = $Cons (x, take (n−1, s))
```

However, this third implementation is not equivalent to the first two. In particular, it forces its second argument when $take$ is applied, rather than when the resulting stream is forced.

The syntax and semantics of **$**-notation are formally defined in Appendix A.

2.1 Streams

As an extended example of lazy evaluation and **$**-notation in Standard ML, we next develop a small streams package. These streams will also be used by several of the data structures in subsequent chapters.

Streams (also known as lazy lists) are very similar to ordinary lists, except that every cell is systematically suspended. The type of streams is

```
datatype α StreamCell = Nil | Cons of α × α Stream
withtype α Stream = α StreamCell susp
```

A simple stream containing the elements 1, 2, and 3 could be written

```
$Cons (1, $Cons (2, $Cons (3, $Nil)))
```

It is illuminating to contrast streams with simple suspended lists of type $\alpha\ list\ susp$. The computations represented by the latter type are inherently *monolithic* — once begun by forcing the suspended list, they run to completion. The computations represented by streams, on the other hand, are often *incremental* — forcing a stream executes only enough of the computation to produce the outermost cell and suspends the rest. This behavior is common among datatypes such as streams that contain nested suspensions.

To see this difference in behavior more clearly, consider the append function, written $s +\!\!+ t$. On suspended lists, this function might be written

```
fun s ++ t = $(force s @ force t)
```

Once begun, this function forces both its arguments and then appends the two lists, producing the entire result. Hence, this function is monolithic. On streams, the append function is written

```
fun s ++ t = $case s of
                 $Nil ⇒ force t
               | $Cons (x, s') ⇒ Cons (x, s' ++ t)
```

Once begun, this function forces the first cell of s (by matching against a **$** pattern). If this cell is Nil, then the first cell of the result is the first cell of t, so the function forces t. Otherwise, the function constructs the first cell of the result from the first element of s and — this is the key point — the suspension that will eventually calculate the rest of the appended list. Hence, this function is incremental. The $take$ function described earlier is similarly incremental.

However, consider the function to drop the first n elements of a stream.

```
fun drop (n, s) = let fun drop' (0, s') = force s'
                        | drop' (n, $Nil) = Nil
```

```
                              | drop′ (n, $Cons (x, s′)) = drop′ (n−1, s′)
                        in $drop′ (n, s) end
```

This function is monolithic because the recursive calls to $drop'$ are never delayed — calculating the first cell of the result requires executing the entire drop function. Another common monolithic stream function is $reverse$.

```
fun reverse s = let fun reverse′ ($Nil, r) = r
                          | reverse′ ($Cons (x, s), r) = reverse′ (s, Cons (x, $r))
                in $reverse′ (s, Nil) end
```

Here the recursive calls to $reverse'$ are never delayed, but note that each recursive call creates a new suspension of the form $\$r$. It might seem then that $reverse$ does not in fact do all of its work at once. However, suspensions such as these, whose bodies are manifestly values (i.e., composed entirely of constructors and variables, with no function applications), are called *trivial*. A good compiler would create these suspensions in already-memoized form, but even if the compiler does not perform this optimization, trivial suspensions always evaluate in $O(1)$ time.

Although monolithic stream functions such as $drop$ and $reverse$ are common, incremental functions such as $+\!\!+$ and $take$ are the *raison d'être* of streams. Each suspension carries a small but significant overhead, so for maximum efficiency laziness should be used only when there is a good reason to do so. If the only uses of lazy lists in a given application are monolithic, then that application should use simple suspended lists rather than streams.

Figure 2.1 summarizes these stream functions as a Standard ML module. Note that the type of streams is defined using Standard ML's $withtype$ construction, but that older versions of Standard ML do not allow $withtype$ declarations in signatures. This feature will be supported in future versions of Standard ML, but if your compiler does not allow it, then a simple workaround is to delete the $Stream$ type and replace every occurrence of $\tau\ Stream$ with $\tau\ StreamCell\ susp$. By including the $StreamCell$ datatype in the signature, we have deliberately chosen to expose the internal representation in order to support pattern matching on streams.

2.2 Historical Notes

Lazy Evaluation Wadsworth [Wad71] first proposed lazy evaluation as an optimization of normal-order reduction in the lambda calculus. Vuillemin [Vui74] later showed that, under certain restricted conditions, lazy evaluation is an optimal evaluation strategy. The formal semantics of lazy evaluation has been studied extensively [Jos89, Lau93, OLT94, AFM$^+$95].

```
signature STREAM =
sig
  datatype α StreamCell = Nil | Cons of α × α Stream
  withtype α Stream = α StreamCell susp

  val ++      : α Stream × α Stream → α Stream      (* stream append *)
  val take    : int × α Stream → α Stream
  val drop    : int × α Stream → α Stream
  val reverse : α Stream → α Stream
end

structure Stream : STREAM =
sig
  datatype α StreamCell = Nil | Cons of α × α Stream
  withtype α Stream = α StreamCell susp

  fun s ++ t = $case s of
                  $Nil ⇒ force t
                | $Cons (x, s′) ⇒ Cons (x, s′ ++ t)
  fun take (n, s) = $case (n, s) of
                       (0, _) ⇒ Nil
                     | (_, $Nil) ⇒ Nil
                     | (_, $Cons (x, s′)) ⇒ Cons (x, take (n−1, s′))
  fun drop (n, s) = let fun drop′ (0, $c) = c
                          | drop′ (n, $Nil) = Nil
                          | drop′ (n, $Cons (x, s′)) = drop′ (n−1, s′)
                    in $drop′ (n, s) end
  fun reverse s = let fun reverse′ ($Nil, r) = r
                        | reverse′ ($Cons (x, s), r) = reverse′ (s, Cons (x, $r))
                  in $reverse′ (s, Nil) end
end
```

Figure 2.1: A small streams package.

Streams Landin introduced streams in [Lan65], but without memoization. Friedman and Wise [FW76] and Henderson and Morris [HM76] extended Landin's streams with memoization.

Memoization Michie [Mic68] coined the term memoization to denote the augmentation of functions with a cache of argument-result pairs. (The argument field is dropped when memoizing suspensions by regarding suspensions as nullary functions.) Hughes [Hug85] later applied memoization, in the original sense of Michie, to functional programs.

Algorithmics Both components of lazy evaluation — delaying computations and memoizing the results — have a long history in algorithm design, although not always in combination. The idea of delaying the execution of potentially expensive computations (often deletions) is used to good effect in hash tables [WV86], priority queues [ST86b, FT87], and search trees [DSST89]. Memoization, on the other hand, is the basic principle of such techniques as dynamic programming [Bel57] and path compression [HU73, TvL84].

Syntax for Lazy Evaluation Early versions of CAML [W^+90], a close cousin of Standard ML, offered support for lazy evaluation similar to the **$**-notation proposed here. Rather than providing a single lazy constructor, however, CAML allowed any data constructor to be tagged as lazy, after which all applications of the constructor would be evaluated lazily. Although this is more flexible than **$**-notation, it also leads to programs that are significantly harder to read. With **$**-notation, it is syntactically apparent which subexpressions are to be evaluated strictly and which are to be evaluated lazily, but in CAML, this information can only be determined by referring back to the type declarations.

Chapter 3

Amortization and Persistence via Lazy Evaluation

Over the past fifteen years, amortization has become a powerful tool in the design and analysis of data structures. Implementations with good amortized bounds are often simpler and faster than implementations with equivalent worst-case bounds. Unfortunately, standard techniques for amortization apply only to ephemeral data structures, and so are unsuitable for designing or analyzing functional data structures, which are automatically persistent.

In this chapter, we review the two traditional techniques for analyzing amortized data structures — the *banker's method* and the *physicist's method* — and show where they break down for persistent data structures. Then, we demonstrate how lazy evaluation can mediate the conflict between amortization and persistence. Finally, we adapt the banker's and physicist's methods to analyze lazy amortized data structures.

The resulting techniques are both the first techniques for designing and analyzing persistent amortized data structures and the first practical techniques for analyzing non-trivial lazy programs.

3.1 Traditional Amortization

The notion of amortization arises from the following observation. Given a sequence of operations, we may wish to know the running time of the entire sequence, but not care about the running time of any individual operation. For instance, given a sequence of n operations, we may wish to bound the total running time of the sequence by $O(n)$ without insisting that each individual operation run in $O(1)$ time. We might be satisfied if a few operations run in $O(\log n)$ or even $O(n)$ time, provided the total cost of the sequence is only $O(n)$. This freedom opens up a wide design space of possible solutions, and often yields new solutions that are simpler and faster than worst-case solutions with equivalent bounds. In fact, for some problems, such as the union-find problem [TvL84], there are amortized solutions that are asymptotically faster than any possible worst-case solution (assuming certain modest restrictions) [Blu86].

To prove an amortized bound, one defines the amortized cost of each operation and then proves that, for any sequence of operations, the total amortized cost of the operations is an upper bound on the total actual cost, i.e.,

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

where a_i is the amortized cost of operation i, t_i is the actual cost of operation i, and m is the total number of operations. Usually, in fact, one proves a slightly stronger result: that at any intermediate stage in a sequence of operations, the accumulated amortized cost is an upper bound on the accumulated actual cost, i.e.,

$$\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i$$

for any j. The difference between the accumulated amortized costs and the accumulated actual costs is called the *accumulated savings*. Thus, the accumulated amortized costs are an upper bound on the accumulated actual costs whenever the accumulated savings is non-negative.

Amortization allows for occasional operations to have actual costs that exceed their amortized costs. Such operations are called *expensive*. Operations whose actual costs are less than their amortized costs are called *cheap*. Expensive operations decrease the accumulated savings and cheap operations increase it. The key to proving amortized bounds is to show that expensive operations occur only when the accumulated savings are sufficient to cover the cost, since otherwise the accumulated savings would become negative.

Tarjan [Tar85] describes two techniques for analyzing ephemeral amortized data structures: the *banker's method* and the *physicist's method*. In the banker's method, the accumulated savings are represented as *credits* that are associated with individual locations in the data structure. These credits are used to pay for future accesses to these locations. The amortized cost of any operation is defined to be the actual cost of the operation plus the credits allocated by the operation minus the credits spent by the operation, i.e.,

$$a_i = t_i + c_i - \overline{c}_i$$

where c_i is the number of credits allocated by operation i, and $\overline{c}_i$ is the number of credits spent by operation i. Every credit must be allocated before it is spent, and no credit may be spent more than once. Therefore, $\sum c_i \geq \sum \overline{c}_i$, which in turn guarantees that $\sum a_i \geq \sum t_i$, as desired. Proofs using the banker's method typically define a *credit invariant* that regulates the distribution of credits in such a way that, whenever an expensive operation might occur, sufficient credits have been allocated in the right locations to cover its cost.

In the physicist's method, one describes a function Φ that maps each object d to a real number called the *potential* of d. The function Φ is typically chosen so that the potential is initially zero and is always non-negative. Then, the potential represents a lower bound on the accumulated savings.

Let d_i be the output of operation i and the input of operation $i+1$. Then, the amortized cost of operation i is defined to be the actual cost plus the change in potential between d_{i-1} and d_i, i.e.,

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

The accumulated actual costs of the sequence of operations are

$$\begin{array}{rcl} \sum_{i=1}^{j} t_i & = & \sum_{i=1}^{j}(a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ & = & \sum_{i=1}^{j} a_i + \sum_{i=1}^{j}(\Phi(d_{i-1}) - \Phi(d_i)) \\ & = & \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j) \end{array}$$

Sums such as $\sum(\Phi(d_{i-1}) - \Phi(d_i))$, where alternating positive and negative terms cancel each other out, are called *telescoping series*. Provided Φ is chosen in such a way that $\Phi(d_0)$ is zero and $\Phi(d_j)$ is non-negative, then $\Phi(d_j) \geq \Phi(d_0)$ and $\sum a_i \geq \sum t_i$, so the accumulated amortized costs are an upper bound on the accumulated actual costs, as desired.

Remark: This is a somewhat simplified view of the physicist's method. In real analyses, one often encounters situations that are difficult to fit into the framework as described. For example, what about functions that take or return more than one object? However, this simplified view suffices to illustrate the relevant issues. $\diamond$

Clearly, the two methods are very similar. We can convert the banker's method to the physicist's method by ignoring locations and taking the potential to be the total number of credits in the object, as indicated by the credit invariant. Similarly, we can convert the physicist's method to the banker's method by converting potential to credits, and placing all credits on the root. It is perhaps surprising that the knowledge of locations in the banker's method offers no extra power, but the two methods are in fact equivalent [Tar85, Sch92]. The physicist's method is usually simpler, but it is occasionally convenient to take locations into account.

Note that both credits and potential are analysis tools only; neither actually appears in the program text (except maybe in comments).

3.1.1 Example: Queues

We next illustrate the banker's and physicist's methods by analyzing a simple functional implementation of the queue abstraction, as specified by the signature in Figure 3.1.

```
signature QUEUE =
sig
  type α Queue

  exception EMPTY

  val empty   : α Queue
  val isEmpty : α Queue → bool

  val snoc    : α Queue × α → α Queue
  val head    : α Queue → α            (* raises EMPTY if queue is empty *)
  val tail    : α Queue → α Queue      (* raises EMPTY if queue is empty *)
end
```

Figure 3.1: Signature for queues.

(Etymological note: $snoc$ is $cons$ spelled backward and means “cons on the right”.)

A common representation for purely functional queues [Gri81, HM81, Bur82] is as a pair of lists, F and R, where F contains the front elements of the queue in the correct order and R contains the rear elements of the queue in reverse order. For example, a queue containing the integers 1. . . 6 might be represented by the lists F = [1,2,3] and R = [6,5,4]. This representation is described by the following datatype:

```
datatype α Queue = Queue of {F : α list, R : α list}
```

In this representation, the head of the queue is the first element of F, so $head$ and $tail$ return and remove this element, respectively.

```
fun head (Queue {F = x :: f, R = r}) = x
fun tail (Queue {F = x :: f, R = r}) = Queue {F = f, R = r}
```

Remark: To avoid distracting the reader with minor details, we will commonly ignore error cases when presenting code fragments. For example, the above code fragments do not describe the behavior of $head$ or $tail$ on empty queues. We will always include the error cases when presenting complete implementations. ◇

Now, the last element of the queue is the first element of R, so $snoc$ simply adds a new element at the head of R.

```
fun snoc (Queue {F = f, R = r}, x) = Queue {F = f, R = x :: r}
```

Elements are added to R and removed from F, so they must somehow migrate from one list to the other. This is accomplished by reversing R and installing the result as the new F whenever F would otherwise become empty, simultaneously setting the new R to [ ]. The goal is to maintain the invariant that F is empty only if R is also empty (i.e., the entire queue is empty). Note that if F were empty when R was not, then the first element of the queue would be the last element of R, which would take $O(n)$ time to access. By maintaining this invariant, we guarantee that $head$ can always find the first element in $O(1)$ time.

$snoc$ and $tail$ must now detect those cases that would otherwise result in a violation of the invariant, and change their behavior accordingly.

```
fun snoc (Queue {F = [ ], ...}, x) = Queue {F = [x], R = [ ]}
  | snoc (Queue {F = f, R = r}, x) = Queue {F = f, R = x :: r}
fun tail (Queue {F = [x], R = r}) = Queue {F = rev r, R = [ ]}
  | tail (Queue {F = x :: f, R = r}) = Queue {F = f, R = r}
```

Note the use of the record wildcard (...) in the first clause of $snoc$. This is Standard ML pattern-matching notation meaning “the remaining fields of this record are irrelevant”. In this case, the R field is irrelevant because we know by the invariant that if F is [ ], then so is R.

A cleaner way to write these functions is to consolidate the invariant-maintenance duties of $snoc$ and $tail$ into a single *pseudo-constructor*. Pseudo-constructors, sometimes called *smart constructors* [Ada93], are functions that replace ordinary constructors in the construction of data, but that check and enforce an invariant. In this case, the pseudo-constructor $queue$ replaces the ordinary constructor $Queue$, but guarantees that F is empty only if R is also empty.

```
fun queue {F = [ ], R = r} = Queue {F = rev r, R = [ ]}
  | queue {F = f, R = r} = Queue {F = f, R = r}
fun snoc (Queue {F = f, R = r}, x) = queue {F = f, R = x :: r}
fun tail (Queue {F = x :: f, R = r}) = queue {F = f, R = r}
```

The complete code for this implementation is shown in Figure 3.2. Every function except $tail$ takes $O(1)$ worst-case time, but $tail$ takes $O(n)$ worst-case time. However, we can show that $snoc$ and $tail$ each take only $O(1)$ amortized time using either the banker's method or the physicist's method.

Using the banker's method, we maintain a credit invariant that the rear list always contains a number of credits equal to its length. Every $snoc$ into a non-empty queue takes one actual step and allocates a credit to the new element of the rear list, for an amortized cost of two. Every $tail$ that does not reverse the rear list takes one actual step and neither allocates nor spends any credits, for an amortized cost of one. Finally, every $tail$ that does reverse the rear list takes $m + 1$ actual steps, where m is the length of the rear list, and spends the m credits contained by that list, for an amortized cost of $m + 1 - m = 1$.

```
structure BatchedQueue : QUEUE =
struct
  datatype $\alpha$ Queue = Queue of {F : $\alpha$ list, R : $\alpha$ list}
        (* Invariant: F is empty only if R is also empty *)

  exception EMPTY

  val empty = Queue {F = [ ], R = [ ]}
  fun isEmpty (Queue {F = $f$, R = $r$}) = null $f$

  fun queue {F = [ ], R = $r$) = Queue {F = rev $r$, R = [ ]}
    | queue $q$ = Queue $q$

  fun snoc (Queue {F = $f$, R = $r$), $x$) = queue {F = $f$, R = $x$ :: $r$}

  fun head (Queue {F = [ ], ...}) = raise EMPTY
    | head (Queue {F = $x$ :: $f$, ...}) = $x$
  fun tail (Queue {F = [ ], ...}) = raise EMPTY
    | tail (Queue {F = $x$ :: $f$, R = $r$}) = queue {F = $f$, R = $r$}
end
```

Figure 3.2: A common implementation of purely functional queues [Gri81, HM81, Bur82].

Using the physicist's method, we define the potential function Φ to be the length of the rear list. Then every $snoc$ into a non-empty queue takes one actual step and increases the potential by one, for an amortized cost of two. Every $tail$ that does not reverse the rear list takes one actual step and leaves the potential unchanged, for an amortized cost of one. Finally, every $tail$ that does reverse the rear list takes $m+1$ actual steps and sets the new rear list to [ ], decreasing the potential by m, for an amortized cost of $m + 1 - m = 1$.

In this simple example, the proofs are virtually identical. Even so, the physicist's method is slightly simpler for the following reason. Using the banker's method, we must first choose a credit invariant, and then decide for each function when to allocate or spend credits. The credit invariant provides guidance in this decision, but does not make it automatic. For instance, should $snoc$ allocate one credit and spend none, or allocate two credits and spend one? The net effect is the same, so this freedom is just one more potential source of confusion. On the other hand, using the physicist's method, we have only one decision to make — the choice of the potential function. After that, the analysis is mere calculation, with no more freedom of choice.

3.2 Persistence: The Problem of Multiple Futures

In the above analyses, we implicitly assumed that queues were used ephemerally (i.e., in a single-threaded fashion). What happens if we try to use these queues persistently?

Let q be the result of inserting n elements into an initially empty queue, so that the front list of q contains a single element and the rear list contains $n - 1$ elements. Now, suppose we use q persistently by taking its tail n times. Each call of $tail\ q$ takes n actual steps. The total actual cost of this sequence of operations, including the time to build q, is $n^2 + n$. If the operations truly took $O(1)$ amortized time each, then the total actual cost would be only $O(n)$. Clearly, using these queues persistently invalidates the $O(1)$ amortized time bounds proved above. Where do these proofs go wrong?

In both cases, a fundamental requirement of the analysis is violated by persistent data structures. The banker's method requires that no credit be spent more than once, while the physicist's method requires that the output of one operation be the input of the next operation (or, more generally, that no output be used as input more than once). Now, consider the second call to $tail\ q$ in the example above. The first call to $tail\ q$ spends all the credits on the rear list of q, leaving none to pay for the second and subsequent calls, so the banker's method breaks. And the second call to $tail\ q$ reuses q rather than the output of the first call, so the physicist's method breaks.

Both these failures reflect the inherent weakness of any accounting system based on accumulated savings — that savings can only be spent once. The traditional methods of amortization operate by accumulating savings (as either credits or potential) for future use. This works well in an ephemeral setting, where every operation has only a single logical future. But with persistence, an operation might have multiple logical futures, each competing to spend the same savings.

3.2.1 Execution Traces and Logical Time

What exactly do we mean by the "logical future" of an operation?

We model logical time with *execution traces*, which give an abstract view of the history of a computation. An execution trace is a directed graph whose nodes represent "interesting" operations, usually just update operations on the data type in question. An edge from v to v' indicates that operation v' uses some result of operation v. The *logical history* of operation v, denoted $\hat{v}$, is the set of all operations on which the result of v depends (including v itself). In other words, $\hat{v}$ is the set of all nodes w such that there exists a path (possibly of length 0) from w to v. A *logical future* of a node v is any path from v to a terminal node (i.e., a node with out-degree zero). If there is more than one such path, then node v has multiple logical futures. We will sometimes refer to the logical history or logical future of an object, meaning the logical history or logical future of the operation that created the object.

Execution traces generalize the notion of *version graphs* [DSST89], which are often used to model the histories of persistent data structures. In a version graph, nodes represent the various versions of a single persistent identity and edges represent dependencies between versions. Thus, version graphs model the results of operations and execution traces model the operations themselves. Execution traces are often more convenient for combining the histories of several persistent identities (perhaps not even of the same data type) or for reasoning about operations that do not return a new version (e.g., queries) or that return several results (e.g., splitting a list into two sublists).

For ephemeral data structures, the out-degree of every node in a version graph or execution trace is typically restricted to be at most one, reflecting the limitation that objects can be updated at most once. To model various flavors of persistence, version graphs allow the out-degree of every node to be unbounded, but make other restrictions. For instance, version graphs are often limited to be trees (forests) by restricting the in-degree of every node to be at most one. Other version graphs allow in-degrees of greater than one, but forbid cycles, making every graph a dag. We make none of these restrictions on execution traces. Nodes with in-degree greater than one correspond to operations that take more than one argument, such as list catenation or set union. Cycles arise from recursively defined objects, which are supported by many lazy languages. We even allow multiple edges between a single pair of nodes, as might occur if a list is catenated with itself.

We will use execution traces in Section 3.4.1 when we extend the banker's method to cope with persistence.

3.3 Reconciling Amortization and Persistence

In the previous section, we saw that traditional methods of amortization break in the presence of persistence because they assume a unique future, in which the accumulated savings will be spent at most once. However, with persistence, multiple logical futures might all try to spend the same savings. In this section, we show how the banker's and physicist's methods can be repaired by replacing the notion of accumulated savings with accumulated debt, where debt measures the cost of unevaluated lazy computations. The intuition is that, although savings can only be spent once, it does no harm to pay off debt more than once.

3.3.1 The Role of Lazy Evaluation

Recall that an *expensive* operation is one whose actual costs are greater than its (desired) amortized costs. For example, suppose some application $f\ x$ is expensive. With persistence, a malicious adversary might call $f\ x$ arbitrarily often. (Note that each operation is a new logical future of x.) If each operation takes the same amount of time, then the amortized bounds degrade to the worst-case bounds. Hence, we must find a way to guarantee that if the first application of f to x is expensive, then subsequent applications of f to x will not be.

Without side-effects, this is impossible under call-by-value (i.e., strict evaluation) or call-by-name (i.e., lazy evaluation without memoization), because every application of f to x will take exactly the same amount of time. Therefore, amortization cannot be usefully combined with persistence in languages supporting only these evaluation orders.

But now consider call-by-need (i.e., lazy evaluation with memoization). If x contains some suspended component that is needed by f, then the first application of f to x will force the (potentially expensive) evaluation of that component and memoize the result. Subsequent operations may then access the memoized result directly. This is exactly the desired behavior!

Remark: In retrospect, the relationship between lazy evaluation and amortization is not surprising. Lazy evaluation can be viewed as a form of self-modification, and amortization often involves self-modification [ST85, ST86b]. However, lazy evaluation is a particularly disciplined form of self-modification — not all forms of self-modification typically used in amortized ephemeral data structures can be encoded as lazy evaluation. In particular, *splaying* [ST85] does not appear to be amenable to this technique. $\diamond$

3.3.2 A Framework for Analyzing Lazy Data Structures

We have just shown that lazy evaluation is necessary to implement amortized data structures purely functionally. Unfortunately, analyzing the running times of programs involving lazy evaluation is notoriously difficult. Historically, the most common technique for analyzing lazy programs has been to pretend that they are actually strict. However, this technique is completely inadequate for analyzing lazy amortized data structures. We next describe a basic framework to support such analyses. In the remainder of this chapter, we will adapt the banker's and physicist's methods to this framework, yielding both the first techniques for analyzing persistent amortized data structures and the first practical techniques for analyzing non-trivial lazy programs.

We classify the costs of any given operation into several categories. First, the *unshared cost* of an operation is the actual time it would take to execute the operation under the assumption that every suspension in the system at the beginning of the operation has already been forced and memoized (i.e., under the assumption that $force$ always takes $O(1)$ time, except for those suspensions that are created and forced within the same operation). The *shared cost* of an operation is the time that it would take to execute every suspension created but not evaluated by the operation (under the same assumption as above). The *complete cost* of an operation is the sum of its shared and unshared costs. Note that the complete cost is what the actual cost of the operation would be if lazy evaluation were replaced with strict evaluation.

We further partition the total shared costs of a sequence of operations into realized and unrealized costs. *Realized costs* are the shared costs for suspensions that are executed during the overall computation. *Unrealized costs* are the shared costs for suspensions that are never executed. The *total actual cost* of a sequence of operations is the sum of the unshared costs and the realized shared costs — unrealized costs do not contribute to the actual cost. Note that the amount that any particular operation contributes to the total actual cost is at least its unshared cost, and at most its complete cost, depending on how much of its shared cost is realized.

We account for shared costs using the notion of *accumulated debt*. Initially, the accumulated debt is zero, but every time a suspension is created, we increase the accumulated debt by the shared cost of the suspension (and any nested suspensions). Each operation then pays off a portion of the accumulated debt. The *amortized cost* of an operation is the unshared cost of the operation plus the amount of accumulated debt paid off by the operation. We are not allowed to force a suspension until the debt associated with the suspension is entirely paid off. This treatment of debt is reminiscent of a *layaway plan*, in which one reserves an item and then makes regular payments, but receives the item only when it is entirely paid off.

There are three important moments in the life cycle of a suspension: when it is created, when it is entirely paid off, and when it is executed. The proof obligation is to show that the second moment precedes the third. If every suspension is paid off before it is forced, then the total amount of debt that has been paid off is an upper bound on the realized shared costs, and therefore the total amortized cost (i.e., the total unshared cost plus the total amount of debt that has been paid off) is an upper bound on the total actual cost (i.e., the total unshared cost plus the realized shared costs). We will formalize this argument in Section 3.4.1.

One of the most difficult problems in analyzing the running time of lazy programs is reasoning about the interactions of multiple logical futures. We avoid this problem by reasoning about each logical future *as if it were the only one*. From the point of view of the operation that creates a suspension, any logical future that forces the suspension must itself pay for the suspension. If two logical futures wish to force the same suspension, then both must pay for the suspension individually — they may not cooperate and each pay only a portion of the debt. An alternative view of this restriction is that we are allowed to force a suspension *only when the debt for that suspension has been paid off within the logical history of current operation*. Using this method, we will sometimes pay off a debt more than once, thereby overestimating the total time required for a particular computation, but this does no harm and is a small price to pay for the simplicity of the resulting analyses.

3.4 The Banker's Method

We adapt the banker's method to account for accumulated debt rather than accumulated savings by replacing credits with debits. Each debit represents a constant amount of suspended work. When we initially suspend a given computation, we create a number of debits proportional to its shared cost and associate each debit with a location in the object. The choice of location for each debit depends on the nature of the computation. If the computation is *monolithic* (i.e., once begun, it runs to completion), then all debits are usually assigned to the root of the result. On the other hand, if the computation is *incremental* (i.e., decomposable into fragments that may be executed independently), then the debits may be distributed among the roots of the partial results.

The amortized cost of an operation is the unshared cost of the operation plus the number of debits discharged by the operation. Note that the number of debits created by an operation is *not* included in its amortized cost. The order in which debits should be discharged depends on how the object will be accessed; debits on nodes likely to be accessed soon should be discharged first. To prove an amortized bound, we must show that, whenever we access a location (possibly triggering the execution of a suspension), all debits associated with that location have already been discharged (and hence the suspended computation has been paid for). This guarantees that the total number of debits discharged by a sequence of operations is an upper bound on the realized shared costs of the operations. The total amortized costs are therefore an upper bound on the total actual costs. Debits leftover at the end of the computation correspond to unrealized shared costs, and are irrelevant to the total actual costs.

Incremental functions play an important role in the banker's method because they allow debits to be dispersed to different locations in a data structure, each corresponding to a nested suspension. Then, each location can be accessed as soon as its debits are discharged, without waiting for the debits at other locations to be discharged. In practice, this means that the initial partial results of an incremental computation can be paid for very quickly, and that subsequent partial results may be paid for as they are needed. Monolithic functions, on the other hand, are much less flexible. The programmer must anticipate when the result of an expensive monolithic computation will be needed, and set up the computation far enough in advance to be able to discharge all its debits by the time its result is needed.

3.4.1 Justifying the Banker's Method

In this section, we justify the claim that the total amortized cost is an upper bound on the total actual cost. The total amortized cost is the total unshared cost plus the total number of debits discharged (counting duplicates); the total actual cost is the total unshared cost plus the realized shared costs. Therefore, we must show that the total number of debits discharged is an upper bound on the realized shared costs.

We can view the banker's method abstractly as a graph labelling problem, using the execution traces of Section 3.2.1. The problem is to label every node in a trace with three (multi)sets $s(v)$, $a(v)$, and $r(v)$ such that

$$\begin{array}{cl} \text{(I)} & v \neq v' \Rightarrow s(v) \cap s(v') = \emptyset \\ \text{(II)} & a(v) \subseteq \bigcup_{w \in \hat{v}} s(w) \\ \text{(III)} & r(v) \subseteq \bigcup_{w \in \hat{v}} a(w) \end{array}$$

$s(v)$ is a set, but $a(v)$ and $r(v)$ may be multisets (i.e., may contain duplicates). Conditions II and III ignore duplicates.

$s(v)$ is the set of debits allocated by operation v. Condition I states that no debit may be allocated more than once. $a(v)$ is the multiset of debits discharged by v. Condition II insists that no debit may be discharged before it is created, or more specifically, that an operation can only discharge debits that appear in its logical history. Finally, $r(v)$ is the multiset of debits *realized* by v (that is, the multiset of debits corresponding to the suspensions forced by v). Condition III requires that no debit may be realized before it is discharged, or more specifically, that no debit may realized unless it has been discharged within the logical history of the current operation.

Why are $a(v)$ and $r(v)$ multisets rather than sets? Because a single operation might discharge the same debits more than once or realize the same debits more than once (by forcing the same suspensions more than once). Although we never deliberately discharge the same debit more than once, it could happen if we were to combine a single object with itself. For example, suppose in some analysis of a list catenation function, we discharge a few debits from the first argument and a few debits from the second argument. If we then catenate a list with itself, we might discharge the same few debits twice.

Given this abstract view of the banker's method, we can easily measure various costs of a computation. Let V be the set of all nodes in the execution trace. Then, the total shared cost is $\sum_{v \in V} |s(v)|$ and the total number of debits discharged is $\sum_{v \in V} |a(v)|$. Because of memoization, the realized shared cost is not $\sum_{v \in V} |r(v)|$, but rather $|\bigcup_{v \in V} r(v)|$, where $\bigcup$ discards duplicates. By Condition III, we know that $\bigcup_{v \in V} r(v) \subseteq \bigcup_{v \in V} a(v)$. Therefore,

$$|\bigcup_{v \in V} r(v)| \leq |\bigcup_{v \in V} a(v)| \leq \sum_{v \in V} |a(v)|$$

So the realized shared cost is bounded by the total number of debits discharged, and the total actual cost is bounded by the total amortized cost, as desired.

Remark: This argument once again emphasizes the importance of memoization. Without memoization (i.e., if we were using call-by-name rather than call-by-need), the total realized cost would be $\sum_{v \in V} |r(v)|$, and there is no reason to expect this sum to be less than $\sum_{v \in V} |a(v)|$. $\diamond$

3.4.2 Example: Queues

We next develop an efficient persistent implementation of queues, and prove that every operation takes only $O(1)$ amortized time using the banker's method.

Based on the discussion in the previous section, we must somehow incorporate lazy evaluation into the design of the data structure, so we replace the pair of lists in the previous implementation with a pair of streams.[1] To simplify later operations, we also explicitly track the lengths of the two streams.

```
datatype $\alpha$ Queue = Queue {F : $\alpha$ Stream, LenF : int, R : $\alpha$ Stream, LenR : int}
```

Note that a pleasant side effect of maintaining this length information is that we can trivially support a constant-time $size$ function.

Now, waiting until the front list becomes empty to reverse the rear list does not leave sufficient time to pay for the reverse. Instead, we periodically *rotate* the queue by moving all the elements of the rear stream to the end of the front stream, replacing F with $F + reverse\ R$ and setting the new rear stream to empty (**$**$Nil$). Note that this transformation does not affect the relative ordering of the elements.

When should we rotate the queue? Recall that $reverse$ is a monolithic function. We must therefore set up the computation far enough in advance to be able to discharge all its debits by the time its result is needed. The $reverse$ computation takes $|R|$ steps, so we will allocate $|R|$ debits to account for its cost. (For now we ignore the cost of the + operation). The earliest the $reverse$ suspension could be forced is after $|F|$ applications of $tail$, so if we rotate the queue when $|R| \approx |F|$ and discharge one debit per operation, then we will have paid for the reverse by the time it is executed. In fact, we will rotate the queue whenever R becomes one longer than F, thereby maintaining the invariant that $|F| \geq |R|$. Incidentally, this guarantees that F is empty only if R is also empty. The major queue functions can now be written as follows:

```
fun snoc (Queue {F = $f$, LenF = $lenF$, R = $r$, LenR = $lenR$}, $x$) =
      queue {F = $f$, LenF = $lenF$, R = $Cons ($x$, $r$), LenR = $lenR$+1}
fun head (Queue {F = $Cons ($x$, $f$), ...}) = $x$
fun tail (Queue {F = $Cons ($x$, $f$), LenF = $lenF$, R = $r$, LenR = $lenR$}) =
      queue {F = $f$, LenF = $lenF$−1, R = $r$, LenR = $lenR$}
```

where the pseudo-constructor $queue$ guarantees that $|F| \geq |R|$.

```
fun queue ($q$ as {F = $f$, LenF = $lenF$, R = $r$, LenR = $lenR$}) =
      if $lenR \leq lenF$ then Queue $q$
      else Queue {F = $f$ + reverse $r$, LenF = $lenF$+$lenR$, R = $Nil, LenR = 0}
```

The complete code for this implementation appears in Figure 3.3.

[1] Actually, it would be enough to replace only the front list with a stream, but we replace both for simplicity.

```
structure BankersQueue : QUEUE =
struct
  datatype α Queue = Queue {F : α Stream, LenF : int, R : α Stream, LenR : int}
      (* Invariants: |F| ≥ |R|, LenF = |F|, LenR = |R| *)

  exception EMPTY

  val empty = Queue {F = $Nil, LenF = 0, R = $Nil, LenR = 0}
  fun isEmpty (Queue {LenF = lenF, ...}) = (lenF = 0)

  fun queue (q as {F = f, LenF = lenF, R = r, LenR = lenR}) =
      if lenR ≤ lenF then Queue q
      else Queue {F = f ++ reverse r, LenF = lenF+lenR, R = $Nil, LenR = 0}

  fun snoc (Queue {F = f, LenF = lenF, R = r, LenR = lenR}, x) =
      queue {F = f, LenF = lenF, R = $Cons (x, r), LenR = lenR+1}

  fun head (Queue {F = $Nil, ...}) = raise EMPTY
    | head (Queue {F = $Cons (x, f), ...}) = x
  fun tail (Queue {F = $Nil, ...}) = raise EMPTY
    | tail (Queue {F = $Cons (x, f), LenF = lenF, R = r, LenR = lenR}) =
      queue {F = f, LenF = lenF−1, R = r, LenR = lenR}
end
```

Figure 3.3: Amortized queues using the banker's method.

To understand how this implementation deals efficiently with persistence, consider the following scenario. Let q_0 be some queue whose front and rear streams are both of length m, and let $q_i = \mathit{tail}\ q_{i-1}$, for $0 < i \leq m + 1$. The queue is rotated during the first application of tail, and the $\mathit{reverse}$ suspension created by the rotation is forced during the last application of tail. This reversal takes m steps, and its cost is amortized over the sequence $q_1 \ldots q_m$. (For now, we are concerned only with the cost of the $\mathit{reverse}$ — we ignore the cost of the ++.)

Now, choose some branch point k, and repeat the calculation from q_k to q_{m+1}. (Note that q_k is used persistently.) Do this d times. How often is the $\mathit{reverse}$ executed? It depends on whether the branch point k is before or after the rotation. Suppose k is after the rotation. In fact, suppose $k = m$ so that each of the repeated branches is a single tail. Each of these branches forces the $\mathit{reverse}$ suspension, but they each force the *same* suspension, so the $\mathit{reverse}$ is executed only once. Memoization is crucial here — without memoization the $\mathit{reverse}$ would be re-executed each time, for a total cost of $m(d + 1)$ steps, with only $m + 1 + d$ operations over which to amortize this cost. For large d, this would result in an $O(m)$ amortized cost per operation, but memoization gives us an amortized cost of only $O(1)$ per operation.

It is possible to re-execute the $\mathit{reverse}$ however. Simply take $k = 0$ (i.e., make the branch point just before the rotation). Then the first $tail$ of each branch repeats the rotation and creates a new $reverse$ suspension. This new suspension is forced in the last $tail$ of each branch, executing the $reverse$. Because these are different suspensions, memoization does not help at all. The total cost of all the reversals is $m \cdot d$, but now we have $(m+1)(d+1)$ operations over which to amortize this cost, yielding an amortized cost of $O(1)$ per operation. The key is that we duplicate work only when we also duplicate the sequence of operations over which to amortize the cost of that work.

This informal argument shows that these queues require only $O(1)$ amortized time per operation even when used persistently. We formalize this proof using the banker's method.

By inspection, the unshared cost of every queue operation is $O(1)$. Therefore, to show that the amortized cost of every queue operation is $O(1)$, we must prove that discharging $O(1)$ debits per operation suffices to pay off every suspension before it is forced. (In fact, only $snoc$ and $tail$ must discharge any debits.)

Let $d(i)$ be the number of debits on the ith node of the front stream and let $D(i) = \sum_{j=0}^{i} d(j)$ be the cumulative number of debits on all nodes up to and including the ith node. We maintain the following *debit invariant*:

$$D(i) \leq \min(2i, |F| - |R|)$$

The $2i$ term guarantees that all debits on the first node of the front stream have been discharged (since $d(0) = D(0) \leq 2 \cdot 0 = 0$), so this node may be forced at will (for instance, by $head$ or $tail$). The $|F| - |R|$ term guarantees that all debits in the entire queue have been discharged whenever the streams are of equal length (i.e., just before the next rotation).

Theorem 3.1 *The $snoc$ and $tail$ operations maintain the debit invariant by discharging one and two debits, respectively.*

Proof: Every $snoc$ operation that does not cause a rotation simply adds a new element to the rear stream, increasing $|R|$ by one and decreasing $|F| - |R|$ by one. This will cause the invariant to be violated at any node for which $D(i)$ was previously equal to $|F| - |R|$. We can restore the invariant by discharging the first debit in the queue, which decreases every subsequent cumulative debit total by one. Similarly, every $tail$ that does not cause a rotation simply removes an element from the front stream. This decreases $|F|$ by one (and hence $|F| - |R|$ by one), but, more importantly, it decreases the index i of every remaining node by one, which in turn decreases $2i$ by two. Discharging the first two debits in the queue restores the invariant. Finally, consider a $snoc$ or $tail$ that causes a rotation. Just before the rotation, we are guaranteed that all debits in the queue have been discharged, so, after the rotation, the only debits are those generated by the rotation itself. If $|F| = m$ and $|R| = m + 1$ at the time of the rotation, then there will be m debits for the append and $m + 1$ debits for the $reverse$. The append function is incremental so we place one of its debits on each of the first m nodes. On the other hand, the $reverse$ function is monolithic so we place all $m+1$ of its debits on node m, the first node of the reversed stream. Thus, the debits are distributed such that

$$d(i) = \begin{cases} 1 & \text{if } i < m \\ m+1 & \text{if } i = m \\ 0 & \text{if } i > m \end{cases} \quad \text{and} \quad D(i) = \begin{cases} i+1 & \text{if } i < m \\ 2m+1 & \text{if } i \geq m \end{cases}$$

This distribution violates the invariant at both node 0 and node m, but discharging the debit on the first node restores the invariant. □

The format of this argument is typical. Debits are distributed across several nodes for incremental functions, and all on the same node for monolithic functions. Debit invariants measure, not just the number of debits on a given node, but the number of debits along the path from the root to the given node. This reflects the fact that accessing a node requires first accessing all its ancestors. Therefore, the debits on all those nodes must be zero as well.

This data structure also illustrates a subtle point about nested suspensions — the debits for a nested suspension may be allocated, and even discharged, before the suspension is physically created. For example, consider how $+\!\!+$ (append) works. The suspension for the second node in the stream is not physically created until the suspension for the first node is forced. However, because of memoization, the suspension for the second node will be shared whenever the suspension for the first node is shared. Therefore, we consider a nested suspension to be implicitly created at the time that its enclosing suspension is created. Furthermore, when considering debit arguments or otherwise reasoning about the shape of an object, we ignore whether a node has been physically created or not. Rather we reason about the shape of an object as if all nodes were in their final form, i.e., as if all suspensions in the object had been forced.

3.5 The Physicist's Method

Like the banker's method, the physicist's method can also be adapted to work with accumulated debt rather than accumulated savings. In the traditional physicist's method, one describes a potential function Φ that represents a lower bound on the accumulated savings. To work with debt instead of savings, we replace Φ with a function Ψ that maps each object to a potential representing an upper bound on the accumulated debt (or at least, an upper bound on this object's portion of the accumulated debt). Roughly speaking, the amortized cost of an operation is then the complete cost of the operation (i.e., the shared and unshared costs) minus the change in potential. Recall that an easy way to calculate the complete cost of an operation is to pretend that all computation is strict.

Any changes in the accumulated debt are reflected by changes in the potential. If an operation does not pay any shared costs, then the change in potential is equal to its shared cost, so the amortized cost of the operation is equal to its unshared cost. On the other hand if an operation does pay some of its shared cost, or shared costs of previous operations, then the change in potential is smaller than its shared cost (i.e., the accumulated debt increases by less than the shared cost), so the amortized cost of the operation is greater than its unshared cost. However, the change in potential may never be more than the shared cost — the amortized cost of an operation may not be less than its unshared cost.

We can justify the physicist's method by relating it back to the banker's method. Recall that in the banker's method, the amortized cost of an operation was its unshared cost plus the number of debits discharged. In the physicist's method, the amortized cost is the complete cost minus the change in potential, or, in other words, the unshared cost plus the difference between the shared cost and the change in potential. If we consider one unit of potential to be equivalent to one debit, then the shared cost is the number of debits by which the accumulated debt could have increased, and the change in potential is the number of debits by which the accumulated debt did increase. The difference must have been made up by discharging some debits. Therefore, the amortized cost in the physicist's method can also be viewed as the unshared cost plus the number of debits discharged.

Sometimes, we wish to force a suspension in an object when the potential of the object is not zero. In that case, we add the object's potential to the amortized cost. This typically happens in queries, where the cost of forcing the suspension cannot be reflected by a change in potential because the operation does not return a new object.

The major difference between the banker's and physicist's methods is that, in the banker's method, we are allowed to force a suspension as soon as the debits for that suspension have been paid off, without waiting for the debits for other suspensions to be discharged, but in the physicist's method, we can force a shared suspension only when we have reduced the entire accumulated debt of an object, as measured by the potential, to zero. Since potential measures only the accumulated debt of an object as a whole and does not distinguish between different locations, we must pessimistically assume that the entire outstanding debt is associated with the particular suspension we wish to force. For this reason, the physicist's method appears to be less powerful than the banker's method. The physicist's method is also weaker in other ways. For instance, it has trouble with operations that take multiple objects as arguments or return multiple objects as results, for which it is difficult to define exactly what "change in potential" means. However, when it applies, the physicist's method tends to be much simpler than the banker's method.

Since the physicist's method cannot take advantage of the piecemeal execution of nested suspensions, there is no reason to prefer incremental functions to monolithic functions. In fact, a good hint that the physicist's method might be applicable is if all or most suspensions are monolithic.

3.5.1 Example: Queues

We next adapt our implementation of queues to use the physicist's method. Again, we show that every operation takes only $O(1)$ amortized time.

Because there is no longer any reason to prefer incremental suspensions over monolithic suspensions, we use suspended lists instead of streams. In fact, the rear list need not be suspended at all, so we represent it with an ordinary list. Again, we explicitly track the lengths of the lists and guarantee that the front list is always at least as long as the rear list.

Since the front list is suspended, we cannot access its first element without executing the entire suspension. We therefore keep a working copy of a prefix of the front list. This working copy is represented as an ordinary list for efficient access, and is non-empty whenever the front list is non-empty. The final datatype is

```
datatype α Queue = Queue of {W : α list, F : α list susp, LenF : int, R : α list, LenR : int}
```

The major functions on queues may then be written

```
fun snoc (Queue {W = w, F = f, LenF = lenF, R = r, LenR = lenR}, x) =
        queue {W = w, F = f, LenF = lenF, R = x :: r, LenR = lenR+1}
fun head (Queue {W = x :: w, ...}) = x
fun tail (Queue {W = x :: w, F = f, LenF = lenF, R = r, LenR = lenR}) =
        queue {W = w, F = $tl (force f), LenF = lenF−1, R = r, LenR = lenR})
```

The pseudo-constructor *queue* must enforce two invariants: that R is no longer than F, and that W is non-empty whenever F is non-empty.

```
fun checkW {W = [ ], F = f, LenF = lenF, R = r, LenR = lenR}) =
        Queue {W = force f, F = f, LenF = lenF, R = r, LenR = lenR})
  | checkW q = Queue q
fun checkR (q as {W = w, F = f, LenF = lenF, R = r, LenR = lenR}) =
        if lenR ≤ lenF then q
        else let val w′ = force f
             in {W = w′, F = $(w′ @ rev r), LenF = lenF+lenR, R = [ ], LenR = 0} end
fun queue q = checkW (checkR q)
```

The complete implementation of these queues appears in Figure 3.4.

To analyze these queues using the physicist's method, we choose a potential function Ψ in such a way that the potential will be zero whenever we force the suspended list. This happens in two situations: when W becomes empty and when R becomes longer than F. We therefore choose Ψ to be

$$\Psi(q) = \min(2|W|, |F| - |R|)$$

```
structure PhysicistsQueue : QUEUE =
struct
   datatype α Queue = Queue of {W : α list, F : α list susp, LenF : int, R : α list, LenR : int}
         (∗ Invariants: W is a prefix of force F, W = [ ] only if F = $[ ], ∗)
         (∗             |F| ≥ |R|, LenF = |F|, LenR = |R|              ∗)

   exception EMPTY

   val empty = Queue {W = [ ], F = $[ ], LenF = 0, R = [ ], LenR = 0}
   fun isEmpty (Queue {LenF = lenF, ...}) = (lenF = 0)

   fun checkW {W = [ ], F = f, LenF = lenF, R = r, LenR = lenR}) =
         Queue {W = force f, F = f, LenF = lenF, R = r, LenR = lenR})
     | checkW q = Queue q
   fun checkR (q as {W = w, F = f, LenF = lenF, R = r, LenR = lenR}) =
         if lenR ≤ lenF then q
         else let val w′ = force f
              in {W = w′, F = $(w′ @ rev r), LenF = lenF + lenR, R = [ ], LenR = 0} end
   fun queue q = checkW (checkR q)

   fun snoc (Queue {W = w, F = f, LenF = lenF, R = r, LenR = lenR}, x) =
         queue {W = w, F = f, LenF = lenF, R = x :: r, LenR = lenR+1}

   fun head (Queue {W = [ ], ...}) = raise EMPTY
     | head (Queue {W = x :: w, ...}) = x
   fun tail (Queue {W = [ ], ...}) = raise EMPTY
     | tail (Queue {W = x :: w, F = f, LenF = lenF, R = r, LenR = lenR}) =
         queue {W = w, F = $tl (force f), LenF = lenF−1, R = r, LenR = lenR})
end
```

Figure 3.4: Amortized queues using the physicist's method.

Theorem 3.2 *The amortized costs of $snoc$ and $tail$ are at most two and four, respectively.*

Proof: Every $snoc$ that does not cause a rotation simply adds a new element to the rear list, increasing $|R|$ by one and decreasing $|F| - |R|$ by one. The complete cost of the $snoc$ is one, and the decrease in potential is at most one, for an amortized cost of at most $1 - (-1) = 2$. Every $tail$ that does not cause a rotation removes the first element from the working list and lazily removes the same element from the front list. This decreases $|W|$ by one and $|F| - |R|$ by one, which decreases the potential by at most two. The complete cost of $tail$ is two, one for the unshared costs (including removing the first element from W) and one for the shared cost of lazily removing the head of F. The amortized cost is therefore at most $2 - (-2) = 4$.

Finally, consider a $snoc$ or $tail$ that causes a rotation. In the initial queue, $|F| = |R|$ so

$\Psi = 0$. Just before the rotation, $|F| = m$ and $|R| = m + 1$. The shared cost of the rotation is $2m + 1$ and the potential of the resulting queue is $2m$. The amortized cost of $snoc$ is thus $1 + (2m + 1) - 2m = 2$. The amortized cost of $tail$ is $2 + (2m + 1) - 2m = 3$. (The difference is that $tail$ must also account for the shared cost of removing the first element of F.)
$\square$

Finally, we consider two variations of these queues that on the surface appear to be modest improvements, but which actually break the amortized bounds. These variations illustrate common mistakes in designing persistent amortized data structures.

In the first variation, we observe that $checkR$ forces F during a rotation and installs the result in W. Wouldn't it be "lazier", and therefore better, to never force F until W becomes empty? The answer is no, and a brief consideration of the potential function reveals why. If W were very short, then the potential would only increase to $2|W|$ after the rotation. This increase would not be large enough to offset the large shared cost of the rotation. Another way of looking at it is that, if $|W| = 1$ at the time of the rotation, then the front list could be forced during the very next $tail$, which does not leave enough time to pay for the rotation.

In the second variation, we observe that during a $tail$, we replace F with **$**$tl$ $(force\ F)$. Creating and forcing suspensions have non-trivial overheads that, even if $O(1)$, can contribute to a large constant factor. Wouldn't it be "lazier", and therefore better, to not change F, but instead to merely decrement $LenF$ to indicate that the element has been removed? The answer is again no, because the removed elements would be discarded all at once when the front list was finally forced. This would contribute to the unshared cost of the operation, not the shared cost, making the unshared cost linear in the worst case. Since the amortized cost can never be less than the unshared cost, this would also make the amortized cost linear.

3.5.2 Example: Bottom-Up Mergesort with Sharing

The majority of examples in the remaining chapters use the banker's method rather than the physicist's method. Therefore, we give a second example of the physicist's method here.

Imagine that you want to sort several similar lists, such as xs and $x :: xs$, or xs @ zs and ys @ zs. For efficiency, you wish to take advantage of the fact that these lists share common tails, so that you do not repeat the work of sorting those tails. We call an abstract data type for this problem a *sortable collection*.

Figure 3.5 gives a signature for sortable collections. Note that the new function, which creates an empty collection, is parameterized by the "less than" relation on the elements to be sorted.

Now, if we create a sortable collection xs' by adding each of the elements in xs, then we can sort both xs and $x :: xs$ by calling $sort\ xs'$ and $sort\ (add\ (x, xs'))$.

```
signature SORTABLE =
sig
  type α Sortable

  val new : {Less : α × α → bool} → α Sortable    (* sort in increasing order by Less *)
  val add : α × α Sortable → α Sortable
  val sort : α Sortable → α list
end
```

Figure 3.5: Signature for sortable collections.

One possible representation for sortable collections is balanced binary search trees. Then add takes $O(\log n)$ worst-case time and $sort$ takes $O(n)$ time. We achieve the same bounds, but in an amortized sense, using *bottom-up mergesort*.

Bottom-up mergesort first splits a list into n ordered segments, where each segment initially contains a single element. It then merges equal-sized segments in pairs until only one segment of each size remains. Finally, segments of unequal size are merged, from smallest to largest.

Suppose we take a snapshot just before the final cleanup phase. Then the sizes of all segments are distinct powers of 2, corresponding to the one bits of n. This is the representation we will use for sortable collections. Then similar collections will share all the work of bottom-up mergesort except for the final cleanup phase merging unequal-sized segments. The complete representation is a suspended list of segments, each of which is an α $list$, together with the comparison function and the size.

```
type α Sortable = {Less : α × α → bool, Size : int, Segments : α list list susp}
```

The individual segments are stored in increasing order of size, and the elements in each segment are stored in increasing order as determined by the comparison function.

The fundamental operation on segments is $merge$, which merges two ordered lists. Except for being parameterized on $less$, this function is completely standard.

```
fun merge less (xs, ys) =
      let fun mrg ([ ], ys) = ys
            | mrg (xs, [ ]) = xs
            | mrg (x :: xs, y :: ys) = if less (x, y) then x :: mrg (xs, y :: ys)
                                       else y :: mrg (x :: xs, ys)
      in mrg (xs, ys) end
```

To add a new element, we create a new singleton segment. If the smallest existing segment is also a singleton, we merge the two segments and continue merging until the new segment is smaller than the smallest existing segment. This merging is controlled by the bits of n. If the lowest bit of n is zero, then we simply cons the new segment onto the segment list. If the lowest bit is one, then we merge the two segments and repeat. Of course, all this is done lazily.

```
fun add (x, {Less = less, Size = size, Segments = segs}) =
      let fun addSeg (seg, segs, size) =
                  if size mod 2 = 0 then seg :: segs
                  else addSeg (merge less (seg, hd segs), tl segs, size div 2)
      in {Less = less, Size = size+1, Segments = $addSeg ([x], force segs, size)} end
```

Finally, to $sort$ a collection, we merge the segments from smallest to largest.

```
fun sort {Less = less, Segments = segs, ...} =
      let fun mergeAll (xs, [ ]) = xs
              | mergeAll (xs, seg :: segs) = mergeAll (merge less (xs, seg), segs)
      in mergeAll ([ ], force segs) end
```

Remark: $mergeAll$ can be viewed as computing

$$[\,] \bowtie s_1 \bowtie \cdots \bowtie s_m$$

where s_i is the ith segment and $\bowtie$ is left-associative, infix notation for $merge$. This is a specific instance of a very common program schema, which can be written

$$c \oplus x_1 \oplus \cdots \oplus x_m$$

for any c and left-associative $\oplus$. Other instances of this schema include summing a list of integers ($c = 0$ and $\oplus = +$) or finding the maximum of a list of natural numbers ($c = 0$ and $\oplus = \max$). One of the greatest strengths of functional languages is the ability to define schemas like this as *higher-order functions* (i.e., functions that take functions as arguments or return functions as results). For example, the above schema might be written

```
fun foldl (f, c, [ ]) = c
   | foldl (f, c, x :: xs) = foldl (f, f (c, x), xs)
```

Then $sort$ could be written

```
fun sort {Less = less, Segments = segs, ...} = foldl (merge less, [ ], force segs)
```

This also takes advantage of the fact that $merge$ is written as a *curried function*. A curried function is a multiargument function that can be *partially applied* (i.e., applied to just some of its arguments). The result is a function that takes the remaining arguments. In this case, we have applied $merge$ to just one of its three arguments, $less$. The remaining two arguments will be supplied by $foldl$. ◇

```
structure BottomUpMergeSort : SORTABLE =
struct
  type α Sortable = {Less : α × α → bool, Size : int, Segments : α list list susp}

  fun merge less (xs, ys) =
        let fun mrg ([ ], ys) = ys
              | mrg (xs, [ ]) = xs
              | mrg (x :: xs, y :: ys) = if less (x, y) then x :: mrg (xs, y :: ys)
                                         else y :: mrg (x :: xs, ys)
        in mrg (xs, ys) end

  fun new {Less = less} = {Less = less, Size = 0, Segments = $[ ]}
  fun add (x, {Less = less, Size = size, Segments = segs}) =
        let fun addSeg (seg, segs, size) =
                  if size mod 2 = 0 then seg :: segs
                  else addSeg (merge less (seg, hd segs), tl segs, size div 2)
        in {Less = less, Size = size+1, Segments = $addSeg ([x], force segs, size)} end
  fun sort {Less = less, Segments = segs, . . . } =
        let fun mergeAll (xs, [ ]) = xs
              | mergeAll (xs, seg :: segs) = mergeAll (merge less (xs, seg), segs)
        in mergeAll ([ ], force segs) end
end
```

Figure 3.6: Sortable collections based on bottom-up mergesort.

The complete code for this implementation of sortable collections appears in Figure 3.6.

We show that add takes $O(\log n)$ amortized time and $sort$ takes $O(n)$ amortized time using the physicist's method. We begin by defining the potential function Ψ, which is completely determined by the size of the collection:

$$\Psi(n) = 2n - 2\sum_{i=0}^{\infty} b_i(n \bmod 2^i + 1)$$

where b_i is the ith bit of n. Note that $\Psi(n)$ is bounded above by $2n$ and that $\Psi(n) = 0$ exactly when $n = 2^k - 1$ for some k.

We first calculate the complete cost of add. Its unshared cost is one and its shared cost is the cost of performing the merges in $addSeg$. Suppose that the lowest k bits of n are one (i.e., $b_i = 1$ for $i < k$ and $b_k = 0$). Then $addSeg$ performs k merges. The first combines two lists of size 1, the second combines two lists of size 2, and so on. Since merging two lists of size m takes $2m$ steps, $addSeg$ takes $(1+1)+(2+2)+\cdots+(2^{k-1}+2^{k-1}) = 2(\sum_{i=0}^{k-1} 2^i) = 2(2^k - 1)$ steps. The complete cost of add is therefore $2(2^k - 1) + 1 = 2^{k+1} - 1$.

Next, we calculate the change in potential. Let $n' = n + 1$ and let b'_i be the ith bit of n'. Then,

$$\begin{array}{rcl}\Psi(n') - \Psi(n) & = & 2n' - 2\sum_{i=0}^{\infty} b'_i(n' \bmod 2^i + 1) - (2n - 2\sum_{i=0}^{\infty} b_i(n \bmod 2^i + 1)) \\ & = & 2 + 2\sum_{i=0}^{\infty}(b_i(n \bmod 2^i + 1) - b'_i(n' \bmod 2^i + 1)) \\ & = & 2 + 2\sum_{i=0}^{\infty} \delta(i)\end{array}$$

where $\delta(i) = b_i(n \bmod 2^i + 1) - b'_i(n' \bmod 2^i + 1)$. We consider three cases: $i < k$, $i = k$, and $i > k$.

- ($i < k$): Since $b_i = 1$ and $b'_i = 0$, $\delta(k) = n \bmod 2^i + 1$. But $n \bmod 2^i = 2^i - 1$ so $\delta(k) = 2^i$.
- ($i = k$): Since $b_k = 0$ and $b'_k = 1$, $\delta(k) = -(n' \bmod 2^k + 1)$. But $n' \bmod 2^k = 0$ so $\delta(k) = -1 = -b'_k$.
- ($i > k$): Since $b'_i = b_i$, $\delta(k) = b'_i(n \bmod 2^i - n' \bmod 2^i)$. But $n' \bmod 2^i = (n + 1) \bmod 2^i = n \bmod 2^i + 1$ so $\delta(i) = b'_i(-1) = -b'_i$.

Therefore,

$$\begin{array}{rcl}\Psi(n') - \Psi(n) & = & 2 + 2\sum_{i=0}^{\infty} \delta(i) \\ & = & 2 + 2\sum_{i=0}^{k-1} 2^i + 2\sum_{i=k}^{\infty}(-b'_i) \\ & = & 2 + 2(2^k - 1) - 2\sum_{i=k}^{\infty} b'_i \\ & = & 2^{k+1} - 2B'\end{array}$$

where B' is the number of one bits in n'. Then the amortized cost of add is

$$(2^{k+1} - 1) - (2^{k+1} - 2B') = 2B' - 1$$

Since B' is $O(\log n)$, so is the amortized cost of add.

Finally, we calculate the amortized cost of $sort$. The first action of $sort$ is to force the suspended list of segments. Since the potential is not necessarily zero, this adds $\Psi(n)$ to the amortized cost of the operation. It next merges the segments from smallest to largest. The worst case is when $n = 2^k - 1$, so that there is one segment of each size from 1 to 2^{k-1}. Merging these segments takes

$$\begin{array}{l}(1+2) + (1+2+4) + (1+2+4+8) + \cdots + (1+2+\cdots+2^{k-1}) \\ \quad = \sum_{i=1}^{k-1}\sum_{j=0}^{i} 2^j = \sum_{i=1}^{k-1}(2^{i+1} - 1) = (2^{k+1} - 4) - (k-1) = 2n - k - 1\end{array}$$

steps altogether. The amortized cost of $sort$ is therefore $O(n) + \Psi(n) = O(n)$.

3.6 Related Work

Debits Some analyses using the traditional banker's method, such as Tarjan's analysis of path compression [Tar83], include both credits and debits. Whenever an operation needs more credits than are currently available, it creates a credit-debit pair and immediately spends the credit. The debit remains as an obligation that must be fulfilled. Later, a surplus credit may be used to discharge the credit.[2] Any debits that remain at the end of the computation add to the total actual cost. Although there are some similarities between the two kinds of debits, there are also some clear differences. For instance, with the debits introduced in this chapter, any debits leftover at the end of the computation are silently discarded.

It is interesting that debits arise in Tarjan's analysis of path compression since path compression is essentially an application of memoization to the $find$ function.

Amortization and Persistence Until this work, amortization and persistence were thought to be incompatible. Several researchers [DST94, Ram92] had noted that amortized data structures could not be made efficiently persistent using existing techniques for adding persistence to ephemeral data structures, such as [DSST89, Die89], for reasons similar to those cited in Section 3.2. Ironically, these techniques produce persistent data structures with amortized bounds, but the underlying data structure must be worst-case. (These techniques have other limitations as well. Most notably, they cannot be applied to data structures supporting functions that combine two or more versions. Examples of offending functions include list catenation and set union.)

The idea that lazy evaluation could reconcile amortization and persistence first appeared, in rudimentary form, in [Oka95c]. The theory and practice of this technique was further developed in [Oka95a, Oka96b].

Amortization and Functional Data Structures In his thesis, Schoenmakers [Sch93] studies amortized data structures in a strict functional language, concentrating on formal derivations of amortized bounds using the traditional physicist's method. He avoids the problems of persistence by insisting that data structures only be used in a single-threaded fashion.

Queues Gries [Gri81, pages 250–251] and Hood and Melville [HM81] first proposed the queues in Section 3.1.1. Burton [Bur82] proposed a similar implementation, but without the restriction that the front list be non-empty whenever the queue is non-empty. (Burton combines $head$ and $tail$ into a single operation, and so does not require this restriction to support $head$ efficiently.) The queues in Section 3.4.2 first appeared in [Oka96b].

[2]There is a clear analogy here to the spontaneous creation and mutual annihilation of particle-antiparticle pairs in physics. In fact, a better name for these debits might be "anticredits".

Time-Analysis of Lazy Programs Several researchers have developed theoretical frameworks for analyzing the time complexity of lazy programs [BH89, San90, San95, Wad88]. However, these frameworks are not yet mature enough to be useful in practice. One difficulty is that these frameworks are, in some ways, too general. In each of these systems, the cost of a program is calculated with respect to some context, which is a description of how the result of the program will be used. However, this approach is often inappropriate for a methodology of program development in which data structures are designed as abstract data types whose behavior, including time complexity, is specified in isolation. In contrast, our analyses prove results that are independent of context (i.e., that hold regardless of how the data structures are used).

Chapter 4

Eliminating Amortization

Most of the time, we do not care whether a data structure has amortized bounds or worst-case bounds; our primary criteria for choosing one data structure over another are overall efficiency and simplicity of implementation (and perhaps availability of source code). However, in some application areas, it is important to bound the running times of individual operations, rather than sequences of operations. In these situations, a worst-case data structure will often be preferable to an amortized data structure, even if the amortized data structure is simpler and faster overall. Raman [Ram92] identifies several such application areas, including

- **Real-time systems:** In real-time systems, predictability is more important than raw speed [Sta88]. If an expensive operation causes the system to miss a hard deadline, it does not matter how many cheap operations finished well ahead of schedule.
- **Parallel systems:** If one processor in a synchronous system executes an expensive operation while the other processors execute cheap operations, then the other processors may sit idle until the slow processor finishes.
- **Interactive systems:** Interactive systems are similar to real-time systems — users often value consistency more than raw speed [But83]. For instance, users might prefer 100 1-second response times to 99 0.25-second response times and 1 25-second response time, even though the latter scenario is twice as fast.

Remark: Raman also identified a fourth application area — persistent data structures. As discussed in the previous chapter, amortization was thought to be incompatible with persistence. But, of course, we now know this to be untrue. $\diamond$

Does this mean that amortized data structures are of no interest to programmers in these areas? Not at all. Since amortized data structures are often simpler than worst-case data structures, it is sometimes easier to design an amortized data structure, and then convert it to a worst-case data structure, than to design a worst-case data structure from scratch.

In this chapter, we describe *scheduling* — a technique for converting many lazy amortized data structures to worst-case data structures by systematically forcing lazy components in such a way that no suspension ever takes very long to execute. Scheduling extends every object with an extra component, called a *schedule*, that regulates the order in which the lazy components of that object are forced.

4.1 Scheduling

Amortized and worst-case data structures differ mainly in when the computations charged to a given operation occur. In a worst-case data structure, all computations charged to an operation occur during the operation. In an amortized data structure, some computations charged to an operation may actually occur during later operations. From this, we see that virtually all nominally worst-case data structures become amortized when implemented in an entirely lazy language because many computations are unnecessarily suspended. To describe true worst-case data structures, we therefore need a strict language. If we want to describe both amortized and worst-case data structures, we need a language that supports both lazy and strict evaluation. Given such a language, we can also consider an intriguing hybrid approach: worst-case data structures that use lazy evaluation internally. We will obtain such data structures by beginning with lazy amortized data structures and modifying them in such a way that every operation runs in the allotted time.

In a lazy amortized data structure, any specific operation might take longer than the stated bounds. However, this only occurs when the operation forces a suspension that has been paid off, but that takes a long time to execute. To achieve worst-case bounds, we must guarantee that every suspension executes in less than the allotted time.

Define the *intrinsic cost* of a suspension to be the amount of time it takes to force the suspension under the assumption that all other suspensions on which it depends have already been forced and memoized, and therefore each take only $O(1)$ time to execute. (This is similar to the definition of the unshared cost of an operation.) The first step in converting an amortized data structure to a worst-case data structure is to reduce the intrinsic cost of every suspension to less than the desired bounds. Usually, this involves rewriting expensive monolithic functions as incremental functions. However, just being incremental is not always good enough — the granularity of each incremental function must be sufficiently fine. Typically, each fragment of an incremental function will have an $O(1)$ intrinsic cost.

Even if every suspension has a small intrinsic cost, however, some suspensions might still take longer than the allotted time to execute. This happens when one suspension depends on another suspension, which in turn depends on a third, and so on. If none of the suspensions have been previously executed, then forcing the first suspension will result in a cascade of forces. For example, consider the following computation:

$$(\cdots((s_1 +\!\!+\ s_2) +\!\!+\ s_3) +\!\!+ \cdots) +\!\!+\ s_k$$

$+\!\!+$ is the canonical incremental function on streams. It does only one step of the append at a time, and each step has an $O(1)$ intrinsic cost. However, it also forces the first node of its left argument. In this example, forcing the first node of the stream returned by the outermost $+\!\!+$ forces the first node of the stream returned by the next $+\!\!+$, and so on. Altogether, this takes $O(k)$ time to execute (or even more if the first node of s_1 is expensive to force).

The second step in converting an amortized data structure to a worst-case data structure is to avoid cascading forces by arranging that, whenever we force a suspension, any other suspensions on which it depends have already been forced and memoized. Then, no suspension takes longer than its intrinsic cost to execute. We accomplish this by systematically *scheduling* the execution of each suspension so that each is ready by the time we need it. The trick is to regard paying off debt as a literal activity, and to force each suspension as it is paid for.

We extend every object with an extra component, called the *schedule*, that, at least conceptually, contains a pointer to every unevaluated suspension in the object. (Some of the suspensions in the schedule may have already been evaluated in a different logical future, but forcing these suspensions a second time does no harm since it can only make our algorithms run faster than expected, not slower.) Every operation, in addition to whatever other manipulations it performs on an object, forces the first few suspensions in the schedule. The exact number of suspensions forced is governed by the amortized analysis; typically, every suspension takes $O(1)$ time to execute, so we force a number of suspensions proportional to the amortized cost of the operation. Depending on the data structure, maintaining the schedule can be non-trivial. For this technique to apply, adding new suspensions to the schedule, or retrieving the next suspension to be forced, cannot require more time than the desired worst-case bounds.

4.2 Real-Time Queues

As an example of this technique, we convert the amortized banker's queues of Section 3.4.2 to worst-case queues. Queues such as these that support all operations in $O(1)$ worst-case time are called *real-time queues* [HM81].

In the original data structure, queues are rotated using $+\!\!+$ and $reverse$. Since $reverse$ is monolithic, our first task is finding a way to perform rotations incrementally. This can be done by executing one step of the reverse for every step of the $+\!\!+$. We define a function $rotate$ such that

$$\text{rotate}\ (f,\ r,\ a) = f +\!\!+ \text{reverse}\ r +\!\!+\ a$$

Then

$$\text{rotate}\ (f,\ r,\ \$\text{Nil}) = f +\!\!+ \text{reverse}\ r$$

The extra argument a is called an *accumulating parameter* and is used to accumulate the partial results of reversing r. It is initially empty.

Rotations occur when $|R| = |F| + 1$, so initially $|r| = |f| + 1$. This relationship is preserved throughout the rotation, so when f is empty, r contains a single element. The base case is therefore

$$\begin{aligned}\text{rotate (\$Nil, \$Cons } (y, \text{ \$Nil}), a) &= (\text{\$Nil}) +\!\!+ \text{ reverse (\$Cons } (y, \text{ \$Nil})) +\!\!+ a \\ &= \text{\$Cons } (y, a)\end{aligned}$$

In the recursive case,

$$\begin{aligned}\text{rotate (\$Cons } (x, f), \text{ \$Cons } (y, r), a) &= (\text{\$Cons } (x, f)) +\!\!+ \text{ reverse (\$Cons } (y, r)) +\!\!+ a \\ &= \text{\$Cons } (x, f +\!\!+ \text{ reverse (\$Cons } (y, r)) +\!\!+ a) \\ &= \text{\$Cons } (x, f +\!\!+ \text{ reverse } r +\!\!+ \text{ \$Cons } (y, a)) \\ &= \text{\$Cons } (x, \text{rotate } (f, r, \text{\$Cons } (y, a)))\end{aligned}$$

The complete code for $rotate$ is

```
fun rotate (f, r, a) = $case (f, r) of
                         ($Nil, $Cons (y, _)) ⇒ Cons (y, a)
                       | ($Cons (x, f′), $Cons (y, r′)) ⇒ Cons (x, rotate (f′, r′, $Cons (y, a)))
```

Note that the intrinsic cost of every suspension created by $rotate$ is $O(1)$. Just rewriting the pseudo-constructor $queue$ to call $rotate$ (f, r, \$$Nil$) instead f ++ $reverse\ r$, and making no other changes, already drastically improves the worst-case behavior of the queue operations from $O(n)$ to $O(\log n)$ (see [Oka95c]), but we can further improve the worst-case behavior to $O(1)$ using scheduling.

We begin by adding a schedule to the datatype. The original datatype is

```
datatype α Queue = Queue {F : α Stream, LenF : int, R : α Stream, LenR : int}
```

We add a new field S of type $\alpha\ Stream$ that represents a schedule for forcing the nodes of F. S is some suffix of F such that all the nodes before S in F have already been forced and memoized. To force the next suspension in F, we simply inspect the first node of S.

Besides adding S, we make two further changes to the datatype. First, to emphasize the fact that the nodes of R need not be scheduled, we change R from a stream to a list. This involves minor changes to $rotate$. Second, we eliminate the length fields. As we will see shortly, we no longer need the length fields to determine when R becomes longer than F — instead, we will obtain this information from the schedule. The new datatype is thus

```
datatype α Queue = Queue of {F : α stream, R : α list, S : α stream}
```

Now, the major queue functions are simply

```
structure RealTimeQueue : QUEUE =
struct
   datatype $\alpha$ Queue = Queue of {F : $\alpha$ stream, R : $\alpha$ list, S : $\alpha$ stream}
         (* Invariant: |S| = |F| − |R| *)

   exception EMPTY

   val empty = Queue {F = $Nil, R = [ ], S = $Nil}
   fun isEmpty (Queue {F = $f$, ...}) = null $f$

   fun rotate ($f$, $r$, $a$) = $case ($f$, $r$) of
                                  ($Nil, $Cons ($y$, _)) ⇒ Cons ($y$, $a$)
                                | ($Cons ($x$, $f'$), $Cons ($y$, $r'$)) ⇒ Cons ($x$, rotate ($f'$, $r'$, $Cons ($y$, $a$)))

   fun queue {F = $f$, R = $r$, S = $Cons ($x$, $s$)} = Queue {F = $f$, R = $r$, S = $s$}
     | queue {F = $f$, R = $r$, S = $Nil} = let val $f'$ = rotate ($f$, $r$, $Nil)
                                             in Queue {F = $f'$, R = [ ], S = $f'$} end

   fun snoc (Queue {F = $f$, R = $r$, S = $s$}, $x$) = queue {F = $f$, R = $x$ :: $r$, S = $s$}

   fun head (Queue {F = $Nil, ...}) = raise EMPTY
     | head (Queue {F = $Cons ($x$, $f$), ...}) = $x$
   fun tail (Queue {F = $Nil, ...}) = raise EMPTY
     | tail (Queue {F = $Cons ($x$, $f$), R = $r$, S = $s$}) = queue {F = $f$, R = $r$, S = $s$}
end
```

Figure 4.1: Real-time queues based on scheduling [Oka95c].

```
   fun snoc (Queue {F = $f$, R = $r$, S = $s$}, $x$) = queue {F = $f$, R = $x$ :: $r$, S = $s$}
   fun head (Queue {F = $Cons ($x$, $f$), ...}) = $x$
   fun tail (Queue {F = $Cons ($x$, $f$), R = $r$, S = $s$}) = queue {F = $f$, R = $r$, S = $s$}
```

The pseudo-constructor $queue$ maintains the invariant that $|S| = |F| - |R|$ (which incidentally guarantees that $|F| \geq |R|$ since $|S|$ cannot be negative). $snoc$ increases $|R|$ by one and $tail$ decreases $|F|$ by one, so when $queue$ is called, $|S| = |F| - |R| + 1$. If S is non-empty, then we restore the invariant by simply taking the tail of S. If S is empty, then R is one longer than F, so we rotate the queue. In either case, inspecting S to determine whether or not it is empty forces and memoizes the next suspension in the schedule.

```
   fun queue {F = $f$, R = $r$, S = $Cons ($x$, $s$)} = Queue {F = $f$, R = $r$, S = $s$}
     | queue {F = $f$, R = $r$, S = $Nil} = let val $f'$ = rotate ($f$, $r$, $Nil)
                                             in Queue {F = $f'$, R = [ ], S = $f'$} end
```

The complete code for this implementation appears in Figure 4.1.

In the amortized analysis, the unshared cost of every queue operation is $O(1)$. Therefore, every queue operation does only $O(1)$ work outside of forcing suspensions. Hence, to show that all queue operations run in $O(1)$ worst-case time, we must prove that no suspension takes more than $O(1)$ time to execute.

Only three forms of suspensions are created by the various queue functions.

- \$$Nil$ is created by $empty$ and $queue$ (in the initial call to $rotate$). This suspension is trivial and therefore executes in $O(1)$ time regardless of whether it has been forced and memoized previously.
- \$$Cons$ (y, a) is created in the second line of $rotate$ and is also trivial.
- Every call to $rotate$ immediately creates a suspension of the form

```
**$case** (f, r, a) **of**
    ($Nil, [y], a) ⇒ Cons (y, a)
  | ($Cons (x, f′), y :: r′, a) ⇒ Cons (x, rotate (f′, r′, $Cons (y, a)))
```

  The intrinsic cost of this suspension is $O(1)$. However, it also forces the first node of f, creating the potential for a cascade of forces. But note that f is a suffix of the front stream that existed just before the previous rotation. The treatment of the schedule S guarantees that *every* node in that stream was forced and memoized prior to the rotation. Forcing the first node of f simply looks up that memoized value in $O(1)$ time. The above suspension therefore takes only $O(1)$ time altogether.

Since every suspension executes in $O(1)$ time, every queue operation takes only $O(1)$ worst-case time.

Hint to Practitioners: These queues are not particularly fast when used ephemerally, because of overheads associated with memoizing values that are never looked at again, but are the fastest known real-time implementation when used persistently.

4.3 Bottom-Up Mergesort with Sharing

As a second example, we modify the sortable collections from Section 3.5.2 to support add in $O(\log n)$ worst-case time and $sort$ in $O(n)$ worst-case time.

The only use of lazy evaluation in the amortized implementation is the suspended call to $addSeg$ in add. This suspension is clearly monolithic, so the first task is to perform this computation incrementally. In fact, we need only make *merge* incremental; since *addSeg* takes only $O(\log n)$ steps, we can afford to execute it strictly. We therefore represent segments as streams rather than lists, and eliminate the suspension on the collection of segments. The new type for the *Segments* field is thus α *Stream list* rather than α *list list susp*.

Rewriting *merge*, *add*, and *sort* to use this new type is straightforward, except that *sort* must convert the final sorted stream back to a list. This is accomplished by the *streamToList* conversion function.

```
fun streamToList ($Nil) = [ ]
  | streamToList ($Cons (x, xs)) = x :: streamToList xs
```

The new version of *merge*, shown in Figure 4.2, performs one step of the merge at a time, with an $O(1)$ intrinsic cost per step. Our second goal is to execute enough merge steps per *add* to guarantee that any sortable collection contains only $O(n)$ unevaluated suspensions. Then *sort* executes at most $O(n)$ unevaluated suspensions in addition to its own $O(n)$ work. Executing these unevaluated suspensions takes at most $O(n)$ time, so *sort* takes only $O(n)$ time altogether.

In the amortized analysis, the amortized cost of *add* was approximately $2B'$, where B' is the number of one bits in $n' = n + 1$. This suggests that *add* should execute two suspensions per one bit, or equivalently, two suspensions per segment. We maintain a separate schedule for each segment. Each schedule is an α *Stream list* containing the partial results of the merge sequence that created this segment. The complete type is therefore

```
type α Schedule = α Stream list
type α Sortable = {Less : α × α → bool, Size : int, Segments : (α Stream × α Schedule) list}
```

To execute one merge step from a schedule, we call the function *exec1*.

```
fun exec1 [ ] = [ ]
  | exec1 (($Nil) :: sched) = exec1 sched
  | exec1 (($Cons (x, xs)) :: sched) = xs :: sched
```

In the second clause, we reach the end of one stream and execute the first step of the next stream. This cannot loop because only the first stream in a schedule can ever be empty. The function *exec2PerSeg* invokes *exec1* twice per segment.

```
fun exec2PerSeg [ ] = [ ]
  | exec2PerSeg ((xs, sched) :: segs) = (xs, exec1 (exec1 sched)) :: exec2PerSeg segs
```

Now, *add* calls *exec2PerSeg*, but it is also responsible for building the schedule for the new segment. If the lowest k bits of n are one, then adding a new element will trigger k merges, of the form

$$((s_0 \bowtie s_1) \bowtie s_2) \bowtie \cdots \bowtie s_k$$

where s_0 is the new singleton segment and $s_1 \dots s_k$ are the first k segments of the existing collection. The partial results of this computation are $s'_0 \dots s'_k$, where $s'_0 = s_0$ and $s'_i = s'_{i-1} \bowtie s_i$. Since the suspensions in s'_i depend on the suspensions in s'_{i-1}, we must schedule the execution of s'_{i-1} before the execution of s'_i. The suspensions in s'_i also depend on the suspensions in s_i, but we guarantee that $s_1 \dots s_k$ have been completely evaluated at the time of the call to add.

The final version of add, that creates the new schedule and executes two suspensions per segment, is

```
fun add ($x$, {Less = $less$, Size = $size$, Segments = $segs$}) =
    let fun addSeg ($xs$, $segs$, $size$, $rsched$) =
            if $size$ mod 2 = 0 then ($xs$, rev ($xs$ :: $rsched$)) :: $segs$
            else let val (($xs'$, [ ]) :: $segs'$) = $segs$
                in addSeg (merge $less$ ($xs$, $xs'$), $segs'$, $size$ div 2, $xs$ :: $rsched$)
        val $segs'$ = addSeg ($Cons ($x$, $Nil), $segs$, $size$, [ ])
    in {Less = $less$, Size = $size$+1, Segments = exec2PerSeg $segs'$} end
```

The accumulating parameter $rsched$ collects the newly merged streams in reverse order. Therefore, we reverse it back to the correct order on the last step. The pattern match in line 4 asserts that the old schedule for that segment is empty, i.e., that it has already been completely executed. We will see shortly why this true.

The complete code for this implementation is shown in Figure 4.2. add has an unshared cost of $O(\log n)$ and $sort$ has an unshared cost of $O(n)$, so to prove the desired worst-case bounds, we must show that the $O(\log n)$ suspensions forced by add take $O(1)$ time each, and that the $O(n)$ unevaluated suspensions forced by $sort$ take $O(n)$ time altogether.

Every merge step forced by add (through $exec2PerSeg$ and $exec1$) depends on two other streams. If the current step is part of the stream s'_i, then it depends on the streams s'_{i-1} and s_i. The stream s'_{i-1} was scheduled before s'_i, so s'_{i-1} has been completely evaluated by the time we begin evaluating s'_i. Furthermore, s_i was completely evaluated before the add that created s'_i. Since the intrinsic cost of each merge step is $O(1)$, and the suspensions forced by each step have already been forced and memoized, every merge step forced by add takes only $O(1)$ worst-case time.

The following lemma establishes both that any segment involved in a merge by $addSeg$ has been completely evaluated and that the collection as a whole contains at most $O(n)$ unevaluated suspensions.

Lemma 4.1 *In any sortable collection of size n, the schedule for a segment of size $m = 2^k$ contains a total of at most $2m - 2(n \bmod m + 1)$ elements.*

Proof: Consider a sortable collection of size n, where the lowest k bits of n are ones (i.e., n can be written $c2^{k+1} + (2^k - 1)$, for some integer c). Then add produces a new segment of size

```
structure ScheduledBottomUpMergeSort : SORTABLE =
struct
  type α Schedule = α Stream list
  type α Sortable = {Less : α × α → bool, Size : int, Segments : (α Stream × α Schedule) list}

  fun merge less (xs, ys) =
        let fun mrg ($Nil, ys) = ys
              | mrg (xs, $Nil) = xs
              | mrg (xs as $Cons (x, xs'), ys as $Cons (y, ys')) =
                  if less (x, y) then $Cons (x, mrg (xs', ys))
                  else $Cons (y, mrg (xs, ys'))
        in mrg (xs, ys) end

  fun exec1 [ ] = [ ]
    | exec1 (($Nil) :: sched) = exec1 sched
    | exec1 (($Cons (x, xs)) :: sched) = xs :: sched
  fun exec2PerSeg [ ] = [ ]
    | exec2PerSeg ((xs, sched) :: segs) = (xs, exec1 (exec1 sched)) :: exec2PerSeg segs

  fun new {Less = less} = {Less = less, Size = 0, Segments = [ ]}
  fun add (x, {Less = less, Size = size, Segments = segs}) =
        let fun addSeg (xs, segs, size, rsched) =
                  if size mod 2 = 0 then (xs, rev (xs :: rsched)) :: segs
                  else let val ((xs', [ ]) :: segs') = segs
                       in addSeg (merge less (xs, xs'), segs', size div 2, xs :: rsched)
            val segs' = addSeg ($Cons (x, $Nil), segs, size, [ ])
        in {Less = less, Size = size+1, Segments = exec2PerSeg segs'} end
  fun sort {Less = less, Segments = segs, . . .} =
        let fun mergeAll (xs, [ ]) = xs
              | mergeAll (xs, (xs', sched) :: segs) = mergeAll (merge less (xs, xs'), segs)
            fun streamToList ($Nil) = [ ]
              | streamToList ($Cons (x, xs)) = x :: streamToList xs
        in streamToList (mergeAll ($Nil, segs)) end
end
```

Figure 4.2: Scheduled bottom-up mergesort.

$m = 2^k$, whose schedule contains streams of sizes $1, 2, 4, \ldots, 2^k$. The total size of this schedule is $2^{k+1} - 1 = 2m - 1$. After executing two steps, the size of the schedule is $2m - 3$. The size of the new collection is $n' = n+1 = c2^{k+1}+2^k$. Since $2m-3 < 2m-2(n' \bmod m+1) = 2m-2$, the lemma holds for this segment.

Every segment of size m' larger than m is unaffected by the add, except for the execution of two steps from the segment's schedule. The size of the new schedule is bounded by

$$2m' - 2(n \bmod m' + 1) - 2 = 2m' - 2(n' \bmod m' + 1),$$

so the lemma holds for these segments as well. □

Now, whenever the k lowest bits of n are ones (i.e., whenever the next add will merge the first k segments), we know by Lemma 4.1 that, for any segment of size $m = 2^i$, where $i < k$, the number of elements in that segment's schedule is at most

$$2m - 2(n \bmod m + 1) = 2m - 2((m-1) + 1) = 0$$

In other words, that segment has been completely evaluated.

Finally, the combined schedules for all segments comprise at most

$$2\sum_{i=0}^{\infty} b_i(2^i - (n \bmod 2^i + 1)) = 2n - 2\sum_{i=0}^{\infty} b_i(n \bmod 2^i + 1)$$

elements, where b_i is the ith bit of n. Note the similarity to the potential function from the physicist's analysis in Section 3.5.2. Since this total is bounded by $2n$, the collection as a whole contains only $O(n)$ unevaluated suspensions, and therefore $sort$ takes only $O(n)$ worst-case time.

4.4 Related Work

Eliminating Amortization Dietz and Raman [DR91, DR93, Ram92] have devised a framework for eliminating amortization based on *pebble games*, where the derived worst-case algorithms correspond to winning strategies in some game. Others have used ad hoc techniques similar to scheduling to eliminate amortization from specific data structures such as *relaxed heaps* [DGST88] and *implicit binomial queues* [CMP88]. The form of scheduling described here was first applied to queues in [Oka95c] and later generalized in [Oka96b].

Queues The queue implementation in Section 4.2 first appeared in [Oka95c]. Hood and Melville [HM81] presented the first purely functional implementation of real-time queues, based on a technique known as *global rebuilding* [Ove83], which will be discussed further in the next chapter. Their implementation does not use lazy evaluation and is more complicated than ours.

Chapter 5

Lazy Rebuilding

The next four chapters describe general techniques for designing functional data structures. We begin in this chapter with *lazy rebuilding*, a variant of *global rebuilding* [Ove83].

5.1 Batched Rebuilding

Many data structures obey balance invariants that guarantee efficient access. The canonical example is balanced binary search trees, which improve the worst-case running time of many tree operations from the $O(n)$ required by unbalanced trees to $O(\log n)$. One approach to maintaining a balance invariant is to rebalance the structure after every update. For most balanced structures, there is a notion of *perfect balance*, which is a configuration that minimizes the cost of subsequent operations. However, since it is usually too expensive to restore perfect balance after every update, most implementations settle for approximations of perfect balance that are at most a constant factor slower. Examples of this approach include AVL trees [AVL62] and red-black trees [GS78].

However, provided no update disturbs the balance too drastically, an attractive alternative is to postpone rebalancing until after a sequence of updates, and then to rebalance the entire structure, restoring it to perfect balance. We call this approach *batched rebuilding*. Batched rebuilding yields good amortized time bounds provided that (1) the data structure is not rebuilt too often, and (2) individual updates do not excessively degrade the performance of later operations. More precisely, condition (1) states that, if one hopes to achieve a bound of $O(f(n))$ amortized time per operation, and the global transformation requires $O(g(n))$ time, then the global transformation cannot be executed any more frequently than every $c \cdot g(n)/f(n)$ operations, for some constant c. For example, consider binary search trees. Rebuilding a tree to perfect balance takes $O(n)$ time, so if one wants each operation to take $O(\log n)$ amortized time, then the data structure must not be rebuilt more often than every $c \cdot n / \log n$ operations, for some constant c.

Assume that a data structure is to be rebuilt every $c \cdot g(n)/f(n)$ operations, and that an individual operation on a newly rebuilt data structure requires $O(f(n))$ time (worst-case or amortized). Then, condition (2) states that, after up to $c \cdot g(n)/f(n)$ updates to a newly rebuilt data structure, individual operations must still take only $O(f(n))$ time (i.e., the cost of an individual operation can only degrade by a constant factor). Update functions satisfying condition (2) are called *weak updates*.

For example, consider the following approach to implementing a delete function on binary search trees. Instead of physically removing the specified node from the tree, leave it in the tree but mark it as deleted. Then, whenever half the nodes in the tree have been deleted, make a global pass removing the deleted nodes and restoring the tree to perfect balance. Does this approach satisfy both conditions, assuming we want deletions to take $O(\log n)$ amortized time?

Suppose a tree contains n nodes, up to half of which are marked as deleted. Then removing the deleted nodes and restoring the tree to perfect balance takes $O(n)$ time. We execute the transformation only every $\frac{1}{2}n$ delete operations, so condition (1) is satisfied. In fact, condition (1) would allow us to rebuild the data structure even more often, as often as every $c \cdot n / \log n$ operations. The naive delete algorithm finds the desired node and marks it as deleted. This takes $O(\log n)$ time, even if up to half the nodes have been marked as deleted, so condition (2) is satisfied. Note that even if half the nodes in the tree are marked as deleted, the average depth per active node is only about one greater than it would be if the deleted nodes had been physically removed. This degrades each operation by only a constant additive factor, whereas condition (2) allows for each operation to be degraded by a constant multiplicative factor. Hence, condition (2) would allow us to rebuild the data structure even less often.

In the above discussion, we described only deletions, but of course binary search trees typically support insertions as well. Unfortunately, insertions are not *weak* because they can create a deep path very quickly. However, a hybrid approach is possible, in which insertions are handled by local rebalancing after every update, as in AVL trees or red-black trees, but deletions are handled via batched rebuilding.

As a second example of batched rebuilding, consider the batched queues of Section 3.1.1. The global rebuilding transformation reverses the rear list into the front list, restoring the queue to a state of perfect balance in which every element is contained in the front list. As we have already seen, batched queues have good amortized efficiency, but only when used ephemerally. Under persistent usage, the amortized bounds degrade to the cost of the rebuilding transformation because it is possible to trigger the transformation arbitrarily often. In fact, this is true for all data structures based on batched rebuilding.

5.2 Global Rebuilding

Overmars [Ove83] developed a technique for eliminating the amortization from batched rebuilding. He called this technique *global rebuilding*. The basic idea is to execute the rebuilding transformation incrementally, performing a few steps per normal operation. This can be usefully viewed as running the rebuilding transformation as a coroutine. The tricky part of global rebuilding is that the coroutine must be started early enough that it can finish by the time the rebuilt structure is needed.

Concretely, global rebuilding is accomplished by maintaining two copies of each object. The primary, or *working*, copy is the ordinary structure. The secondary copy is the one that is gradually being rebuilt. All queries and updates operate on the working copy. When the secondary copy is completed, it becomes the new working copy and the old working copy is discarded. A new secondary copy might be started immediately, or the object may carry on for a while without a secondary structure, before eventually starting the next rebuilding phase.

There is a further complication to handle updates that occur while the secondary copy is being rebuilt. The working copy will be updated in the normal fashion, but the secondary copy must be updated as well or the effect of the update will be lost when the secondary copy takes over. However, the secondary copy will not in general be represented in a form that can be efficiently updated. Thus, these updates to the secondary copy are buffered and executed, a few at a time, after the secondary copy has been rebuilt, but before it takes over as the working copy.

Global rebuilding can be implemented purely functionally, and has been several times. For example, the real-time queues of Hood and Melville [HM81] are based on this technique. Unlike batched rebuilding, global rebuilding has no problems with persistence. Since no one operation is particularly expensive, arbitrarily repeating operations has no effect on the time bounds. Unfortunately, global rebuilding is often quite complicated. In particular, representing the secondary copy, which amounts to capturing the intermediate state of a coroutine, can be quite messy.

5.3 Lazy Rebuilding

The implementation of queues in Section 3.5.1, based on the physicist's method, is closely related to global rebuilding, but there is an important difference. As in global rebuilding, this implementation keeps two copies of the front list, the working copy W and the secondary copy F, with all queries being answered by the working copy. Updates to F (i.e., $tail$ operations) are buffered, to be executed during the next rotation, by writing

… F = **$**tl (force f) …

In addition, this implementation takes care to start (or at least set up) the rotation long before its result is needed. However, unlike global rebuilding, this implementation does not *execute* the rebuilding transformation (i.e., the rotation) concurrently with the normal operations; rather, it *pays for* the rebuilding transformation concurrently with the normal operations, but then executes the transformation all at once at some point after it has been paid for. In essence, we have replaced the complications of explicitly or implicitly coroutining the rebuilding transformation with the simpler mechanism of lazy evaluation. We call this variant of global rebuilding *lazy rebuilding*.

The implementation of queues in Section 3.4.2, based on the banker's method, reveals a further simplification possible under lazy rebuilding. By incorporating nested suspensions into the basic data structure — for instance, by using streams instead of lists — we can often eliminate the distinction between the working copy and the secondary copy and employ a single structure that combines aspects of both. The "working" portion of that structure is the part that has already been paid for, and the "secondary" portion is the part that has not yet been paid for.

Global rebuilding has two advantages over batched rebuilding: it is suitable for implementing persistent data structures and it yields worst-case bounds rather than amortized bounds. Lazy rebuilding shares the first advantage, but, at least in its simplest form, yields amortized bounds. However, if desired, worst-case bounds can often be recovered using the scheduling techniques of Chapter 4. For example, the real-time queues in Section 4.2 combine lazy rebuilding with scheduling to achieve worst-case bounds. In fact, when lazy rebuilding is combined with scheduling, it can be viewed as an instance of global rebuilding in which the coroutines are reified in a particularly simple way using lazy evaluation.

5.4 Double-Ended Queues

As further examples of lazy rebuilding, we next present several implementations of double-ended queues, also known as *deques*. Deques differ from FIFO queues in that elements can be both inserted and deleted from either end of the queue. A signature for deques appears in Figure 5.1. This signature extends the signature for queues with three new functions: $cons$ (insert an element at the front), $last$ (return the rearmost element), and $init$ (remove the rearmost element).

Remark: Note that the signature for queues is a strict subset of the signature for deques — the same names have been chosen for the types, exceptions, and overlapping functions. Because deques are thus a strict extension of queues, Standard ML will allow us to use a deque module wherever a queue module is expected. $\diamond$

```
signature DEQUE =
sig
  type α Queue

  exception EMPTY

  val empty   : α Queue
  val isEmpty : α Queue → bool

  (* insert, inspect, and remove the front element *)
  val cons    : α × α Queue → α Queue
  val head    : α Queue → α           (* raises EMPTY if queue is empty *)
  val tail    : α Queue → α Queue     (* raises EMPTY if queue is empty *)

  (* insert, inspect, and remove the rear element *)
  val snoc    : α Queue × α → α Queue
  val last    : α Queue → α           (* raises EMPTY if queue is empty *)
  val init    : α Queue → α Queue     (* raises EMPTY if queue is empty *)
end
```

Figure 5.1: Signature for double-ended queues.

5.4.1 Output-restricted Deques

First, note that extending the queue implementations from Chapters 3 and 4 to support $cons$, in addition to $snoc$, is trivial. A queue that supports insertions at both ends, but deletions from only one end, is called an *output-restricted deque*.

For example, we can implement $cons$ for the banker's queues of Section 3.4.2 as follows:

```
fun cons (x, Queue {F = f, LenF = lenF, R = r, LenR = lenR}) =
      Queue {F = $Cons (x, f), LenF = lenF+1, R = r, LenR = lenR}
```

Note that we invoke the true constructor $Queue$ rather than the pseudo-constructor $queue$ because adding an element to F cannot possibly make F shorter than R.

Similarly, we can easily extend the real-time queues of Section 4.2.

```
fun cons (x, Queue {F = f, R = r, S = s}) =
      Queue {F = $Cons (x, f), R = r, S = $Cons (x, s)})
```

We add x to S only to maintain the invariant that $|S| = |F| - |R|$. Again, we invoke the true constructor $Queue$ rather than the pseudo-constructor $queue$.

5.4.2 Banker's Deques

Deques can be represented in essentially the same way as queues, as two streams (or lists) F and R, plus some associated information to help maintain balance. For queues, the notion of perfect balance is for all the elements to be in the front stream. For deques, the notion of perfect balance is for the elements to be evenly divided between the front and rear streams. Since we cannot afford to restore perfect balance after every operation, we will settle for guaranteeing that neither stream is more than about c times longer than the other, for some constant $c > 1$. Specifically, we maintain the following balance invariant:

$$|F| \leq c|R| + 1 \quad \wedge \quad |R| \leq c|F| + 1$$

The "+1" in each term allows for the only element of a singleton queue to be stored in either stream. Note that both streams will be non-empty whenever the queue contains at least two elements. Whenever the invariant would otherwise be violated, we restore the queue to perfect balance by transferring elements from the longer stream to the shorter stream until both streams have the same length.

Using these ideas, we can adapt either the banker's queues of Section 3.4.2 or the physicist's queues of Section 3.5.1 to obtain deques that support every operation in $O(1)$ amortized time. Because the banker's queues are slightly simpler, we choose to begin with that implementation.

The type of double-ended queues is precisely the same as for ordinary queues.

```
datatype α Queue = Queue {F : α Stream, LenF : int, R : α Stream, LenR : int}
```

The functions on the front element are defined as follows:

```
fun cons (Queue {F = f, LenF = lenF, R = r, LenR = lenR}, x) =
      queue {F = $Cons (x, f), LenF = lenF+1, R = r, LenR = lenR}
fun head (Queue {F = $Nil, R = $Cons (x, _), ...} = x
  | head (Queue {F = $Cons (x, f), ...}) = x
fun tail (Queue {F = $Nil, R = $Cons (x, _), ...} = empty
  | tail (Queue {F = $Cons (x, f), LenF = lenF, R = r, LenR = lenR}) =
      queue {F = f, LenF = lenF−1, R = r, LenR = lenR}
```

The first clauses of *head* and *tail* handle singleton queues where the single element is stored in the rear stream. The functions on the rear element — *snoc*, *last*, and *init* — are defined symmetrically on R rather than F.

The interesting portion of this implementation is the pseudo-constructor *queue*, which restores the queue to perfect balance when one stream becomes too long by first truncating the longer stream to half the combined length of both streams and then transferring the remaining elements of the longer stream onto the back of the shorter stream. For example, if

$|F| > c|R| + 1$, then $queue$ replaces F with $take\ (i, F)$ and R with R ++ $reverse\ (drop\ (i, F))$, where $i = \lfloor(|F| + |R|)/2\rfloor$. The full definition of $queue$ is

```
fun queue (q as {F = f, LenF = lenF, R = r, LenR = lenR}) =
      if lenF > c*lenR + 1 then
         let val i = (lenF + lenR) div 2          val j = lenF + lenR − i
             val f′ = take (i, f)                 val r′ = r ++ reverse (drop (i, f))
         in Queue {F = f′, LenF = i, R = r′, LenR = j} end
      else if lenR > c*lenF + 1 then
         let val i = (lenF + lenR) div 2          val j = lenF + lenR − i
             val f′ = f ++ reverse (drop (j, r))  val r′ = take (j, r)
         in Queue {F = f′, LenF = i, R = r′, LenR = j} end
      else Queue q
```

The complete implementation appears in Figure 5.2.

Remark: Because of the symmetry of this implementation, we can reverse a deque in $O(1)$ time by simply swapping the roles of F and R.

```
fun reverse (Queue {F = f, LenF = lenF, R = r, LenR = lenR}) =
      Queue {F = r, LenF = lenR, R = f, LenR = lenF}
```

Many other implementations of deques share this property [Hoo92b, CG93]. Rather than essentially duplicating the code for the functions on the front element and the functions on the rear element, we could define the functions on the rear element in terms of $reverse$ and the corresponding functions on the front element. For example, we could implement $init$ as

```
fun init q = reverse (tail (reverse q))
```

Of course, $init$ will be slightly faster if implemented directly. ◇

To analyze these deques, we again turn to the banker's method. For both the front and rear streams, let $d(i)$ be the number of debits on element i of the stream, and let $D(i) = \sum_{j=0}^{i} d(j)$. We maintain the debit invariants that, for both the front and rear streams,

$$D(i) \leq \min(ci + i, cs + 1 - t)$$

where $s = \min(|F|, |R|)$ and $t = \max(|F|, |R|)$. Since $D(0) = 0$ for both streams, we can always access the first and last elements of the queue via $head$ or $last$.

Theorem 5.1 *$cons$ and $tail$ (symmetrically, $snoc$ and $init$) maintain the debit invariants on both the front and rear streams by discharging at most 1 and $c + 1$ debits per stream, respectively.*

```
functor BankersDeque (val $c$ : int) : DEQUE =        (* $c > 1$ *)
struct
  datatype $\alpha$ Queue = Queue {F : $\alpha$ Stream, LenF : int, R : $\alpha$ Stream, LenR : int}
        (* Invariants: $|F| \leq c|R| + 1$, $|R| \leq c|F| + 1$, LenF = $|F|$, LenR = $|R|$ *)

  exception EMPTY

  val empty = Queue {F = $Nil, LenF = 0, R = $Nil, LenR = 0}
  fun isEmpty (Queue {LenF = $lenF$, LenR = $lenR$, ...}) = ($lenF + lenR = 0$)

  fun queue ($q$ as {F = $f$, LenF = $lenF$, R = $r$, LenR = $lenR$}) =
        if $lenF > c*lenR + 1$ then
          let val $i$ = ($lenF + lenR$) div 2          val $j$ = $lenF + lenR - i$
              val $f'$ = take ($i, f$)                 val $r'$ = $r$ ++ reverse (drop ($i, f$))
          in Queue {F = $f'$, LenF = $i$, R = $r'$, LenR = $j$} end
        else if $lenR > c*lenF + 1$ then
          let val $i$ = ($lenF + lenR$) div 2          val $j$ = $lenF + lenR - i$
              val $f'$ = $f$ ++ reverse (drop ($j, r$))  val $r'$ = take ($j, r$)
          in Queue {F = $f'$, LenF = $i$, R = $r'$, LenR = $j$} end
        else Queue $q$

  fun cons (Queue {F = $f$, LenF = $lenF$, R = $r$, LenR = $lenR$}, $x$) =
        queue {F = $Cons ($x, f$), LenF = $lenF$+1, R = $r$, LenR = $lenR$}
  fun head (Queue {F = $Nil, R = $Nil, ...}) = raise EMPTY
    | head (Queue {F = $Nil, R = $Cons ($x$, _), ...} = $x$
    | head (Queue {F = $Cons ($x, f$), ...}) = $x$
  fun tail (Queue {F = $Nil, R = $Nil, ...}) = raise EMPTY
    | tail (Queue {F = $Nil, R = $Cons ($x$, _), ...} = empty
    | tail (Queue {F = $Cons ($x, f$), LenF = $lenF$, R = $r$, LenR = $lenR$}) =
        queue {F = $f$, LenF = $lenF-1$, R = $r$, LenR = $lenR$}

  ...snoc, last, and init defined symmetrically...
end
```

Figure 5.2: An implementation of deques based on lazy rebuilding and the banker's method.

Proof: Similar to the proof of Theorem 3.1 on page 27. □

By inspection, every operation has an $O(1)$ unshared cost, and by Theorem 5.1, every operation discharges no more than $O(1)$ debits. Therefore, every operation runs in $O(1)$ amortized time.

5.4.3 Real-Time Deques

Real-time deques support every operation in $O(1)$ worst-case time. We obtain real-time deques from the deques of the previous section by scheduling both the front and rear streams.

As always, the first step in applying the scheduling technique is to convert all monolithic functions to incremental functions. In the previous implementation, the rebuilding transformation rebuilt F and R as $take\ (i,\ F)$ and $R \mathbin{+\!\!+} reverse\ (drop\ (i,\ F))$ (or vice versa). $take$ and $\mathbin{+\!\!+}$ are already incremental, but $reverse$ and $drop$ are monolithic. We therefore rewrite $R \mathbin{+\!\!+} reverse\ (drop\ (i,\ F))$ as $rotateDrop\ (R,\ i,\ F)$ where $rotateDrop$ performs c steps of the $drop$ for every step of the $\mathbin{+\!\!+}$ and eventually calls $rotateRev$, which in turn performs c steps of the $reverse$ for every remaining step of the $\mathbin{+\!\!+}$. $rotateDrop$ can be implemented as

```
fun rotateDrop ($r$, $i$, $f$) =
      if $i < c$ then rotateRev ($r$, drop ($i$, $f$), $Nil)
      else let val ($Cons ($x$, $r'$)) = $r$ in $Cons ($x$, rotateDrop ($r'$, $i - c$, drop ($c$, $f$))) end
```

Initially, $|f| = c|r| + 1 + k$ where $1 \leq k \leq c$. Every call to $rotateDrop$ drops c elements of f and processes one element of r, except the last, which drops $i \bmod c$ elements of f and leaves r unchanged. Therefore, at the time of the first call to $rotateRev$, $|f| = c|r|+1+k-(i \bmod c)$. It will be convenient to insist that $|f| \geq c|r|$, so we require that $1 + k - (i \bmod c) \geq 0$. This is guaranteed only if c is two or three, so these are the only values of c that we allow. Then we can implement $rotateRev$ as

```
fun rotateRev ($Nil, $f$, $a$) = reverse $f$ ++ $a$
   | rotateRev ($Cons ($x$, $r$), $f$, $a$) =
      $Cons ($x$, rotateRev ($r$, drop ($c$, $f$), reverse (take ($c$, $f$)) ++ $a$))
```

Note that $rotateDrop$ and $rotateRev$ make frequent calls to $drop$ and $reverse$, which were exactly the functions we were trying to eliminate. However, now $drop$ and $reverse$ are always called with arguments of bounded size, and therefore execute in $O(1)$ steps.

Once we have converted the monolithic functions to incremental functions, the next step is to schedule the execution of the suspensions in F and R. We maintain a separate schedule for each stream and execute a few suspensions per operation from each schedule. As with the real-time queues of Section 4.2, the goal is to ensure that both schedules are completely evaluated before the next rotation. Assume that both streams have length m immediately after a rotation. How soon can the next rotation occur? It will occur soonest if all the insertions occur on one end and all the deletions occur on the other end. If i is the number of insertions and d is the number of deletions, then the next rotation will occur when

$$m + i > c(m - d) + 1$$

Rewriting both sides yields

$$i + cd > m(c - 1) + 1$$

The next rotation will occur sooner for $c = 2$ than for $c = 3$, so substitute 2 for c.

$$i + 2d > m + 1$$

Therefore, executing one suspension per stream per insertion and two suspensions per stream per deletion is enough to guarantee that both schedules are completely evaluated before the next rotation.

The complete implementation appears in Figure 5.3.

5.5 Related Work

Global Rebuilding Overmars introduced global rebuilding in [Ove83]. It has since been used in many situations, including real-time queues [HM81], real-time deques [Hoo82, GT86, Sar86, CG93], catenable deques [BT95], and the order maintenance problem [DS87].

Deques Hood [Hoo82] first modified the real-time queues of [HM81] to obtain real-time deques based on global rebuilding. Several other researchers later duplicated this work [GT86, Sar86, CG93]. These implementations are all similar to techniques used to simulate multihead Turing machines [Sto70, FMR72, LS81]. Hoogerwoord [Hoo92b] proposed amortized deques based on batched rebuilding, but, as always with batched rebuilding, his implementation is not efficient when used persistently. The real-time deques in Figure 5.3 first appeared in [Oka95c].

Coroutines and Lazy Evaluation Streams (and other lazy data structures) have frequently been used to implement a form of coroutining between the producer of a stream and the consumer of a stream. Landin [Lan65] first pointed out this connection between streams and coroutines. See [Hug89] for some compelling applications of this feature.

```
functor RealTimeDeque (val $c$ : int) : DEQUE =        (* $c = 2$ or $c = 3$ *)
struct
  datatype $\alpha$ Queue = Queue {F : $\alpha$ Stream, LenF : int, SF : $\alpha$ Stream,
                                   R : $\alpha$ Stream, LenR : int, SR : $\alpha$ Stream}
      (* Invariants: $|F| \leq c|R| + 1$, $|R| \leq c|F| + 1$, LenF = $|F|$, LenR = $|R|$ *)

  exception EMPTY

  val empty = Queue {F = $Nil, LenF = 0, SF = $Nil, R = $Nil, LenR = 0, SR = $Nil}
  fun isEmpty (Queue {LenF = $lenF$, LenR = $lenR$, ...}) = ($lenF$+$lenR$ = 0)

  fun exec1 ($Cons ($x$, $s$)) = $s$
    | exec1 $s$ = $s$
  fun exec2 $s$ = exec1 (exec1 $s$)

  fun rotateRev ($Nil, $f$, $a$) = reverse $f$ ++ $a$
    | rotateRev ($Cons ($x$, $r$), $f$, $a$) =
      $Cons ($x$, rotateRev ($r$, drop ($c$, $f$), reverse (take ($c$, $f$)) ++ $a$))
  fun rotateDrop ($r$, $i$, $f$) =
      if $i < c$ then rotateRev ($r$, drop ($i$, $f$), $Nil)
      else let val ($Cons ($x$, $r'$)) = $r$ in $Cons ($x$, rotateDrop ($r'$, $i - c$, drop ($c$, $f$))) end

  fun queue ($q$ as {F = $f$, LenF = $lenF$, SF = $sf$, R = $r$, LenR = $lenR$, SR = $sr$}) =
      if $lenF > c*lenR$ + 1 then
        let val $i$ = ($lenF$ + $lenR$) div 2        val $j$ = $lenF$ + $lenR - i$
            val $f'$ = take ($i$, $f$)               val $r'$ = rotateDrop ($i$, $r$, $f$)
        in Queue {F = $f'$, LenF = $i$, SF = $f'$, R = $r'$, LenR = $j$, SR = $r'$} end
      else if $lenR > c*lenF$ + 1 then
        let val $i$ = ($lenF$ + $lenR$) div 2        val $j$ = $lenF$ + $lenR - i$
            val $f'$ = rotateDrop ($j$, $f$, $r$)     val $r'$ = take ($j$, $r$)
        in Queue {F = $f'$, LenF = $i$, SF = $f'$, R = $r'$, LenR = $j$, SR = $r'$} end
      else Queue $q$

  fun cons (Queue {F = $f$, LenF = $lenF$, SF = $sf$, R = $r$, LenR = $lenR$, SR = $sr$}, $x$) =
      queue {F = $Cons ($x$, $f$), LenF = $lenF$+1, SF = exec1 $sf$,
             R = $r$, LenR = $lenR$, SR = exec1 $sr$}
  fun head (Queue {F = $Nil, R = $Nil, ...}) = raise EMPTY
    | head (Queue {F = $Nil, R = $Cons ($x$, _), ...} = $x$
    | head (Queue {F = $Cons ($x$, $f$), ...}) = $x$
  fun tail (Queue {F = $Nil, R = $Nil, ...}) = raise EMPTY
    | tail (Queue {F = $Nil, R = $Cons ($x$, _), ...} = empty
    | tail (Queue {F = $Cons ($x$, $f$), LenF = $lenF$, SF = $sf$, R = $r$, LenR = $lenR$, SR = $sr$}) =
      queue {F = $f$, LenF = $lenF-1$, SF = exec2 $sf$, R = $r$, LenR = $lenR$, SR = exec2 $sr$}

  ...snoc, last, and init defined symmetrically...
end
```

Figure 5.3: Real-time deques via lazy rebuilding and scheduling.

Chapter 6

Numerical Representations

Consider the usual representations of lists and natural numbers, along with several typical functions on each data type.

```
datatype α List =                      datatype Nat =
        Nil                                    Zero
      | Cons of α × α List                   | Succ of Nat

fun tail (Cons (x, xs)) = xs           fun pred (Succ n) = n

fun append (Nil, ys) = ys              fun plus (Zero, n) = n
    | append (Cons (x, xs), ys) =          | plus (Succ m, n) =
        Cons (x, append (xs, ys))              Succ (plus (m, n))
```

Other than the fact that lists contain elements and natural numbers do not, these two implementations are virtually identical. This suggests a strong analogy between representations of the number n and representations of container objects of size n. Functions on the container strongly resemble arithmetic functions on the number. For example, inserting an element resembles incrementing a number, deleting an element resembles decrementing a number, and combining two containers resembles adding two numbers. This analogy can be exploited to design new implementations of container abstractions — simply choose a representation of natural numbers with certain desired properties and define the functions on the container objects accordingly. Call an implementation designed in this fashion a *numerical representation*.

The typical representation of lists can be viewed as a numerical representation based on unary numbers. However, numerical representations based on binary numbers are also common; the best known of these is the binomial queues of Vuillemin [Vui78]. Incrementing a unary number takes $O(1)$ time, so inserting an element into a unary representation also usually takes $O(1)$ time. However, adding two unary numbers takes $O(n)$ time, so combining two containers in a unary representation takes $O(n)$ time. Binary numbers improve the time required for addition (and hence the time required to combine two containers) to $O(\log n)$, but also slow the time required to increment a number or insert an element to $O(\log n)$. In this chapter, we consider several variations of binary numbers that achieve the best of both worlds by supporting the increment function in $O(1)$ time and addition in $O(\log n)$ time. Numerical representations based on these variations naturally support inserting an element in $O(1)$ time and combining two containers in $O(\log n)$ time.

Example abstractions for which numerical representations are particularly useful include *random-access lists* (also known as *flexible arrays*) and *heaps* (also known as *priority queues*).

6.1 Positional Number Systems

A *positional number system* [Knu73] is a notation for writing a number as a sequence of digits $b_0 \dots b_{m-1}$. The digit b_0 is called the *least significant digit* and the digit b_{m-1} is called the *most significant digit*. Except when writing ordinary, decimal numbers, we will always write sequences of digits from least significant to most significant.

Each digit b_i has weight w_i, so the value of the sequence $b_0 \dots b_{m-1}$ is $\sum_{i=0}^{m-1} b_i w_i$. For any given positional number system, the sequence of weights is fixed, as is the set of digits D_i from which each b_i is chosen. For unary numbers, $w_i = 1$ and $D_i = \{\mathtt{1}\}$ for all i, and for binary numbers $w_i = 2^i$ and $D_i = \{\mathtt{0}, \mathtt{1}\}$. (By convention, we write all digits in typewriter font except for ordinary, decimal digits.) A number is said to be written in base B if $w_i = B^i$ and $D_i = \{\mathtt{0}, \dots, B-1\}$. Usually, but not always, weights are increasing sequences of powers, and the set D_i is the same for every digit.

A number system is said to be *redundant* if there is more than one way to represent some numbers. For example, we can obtain a redundant system of binary numbers by taking $w_i = 2^i$ and $D_i = \{\mathtt{0}, \mathtt{1}, \mathtt{2}\}$. Then the decimal number 13 can be written `1011`, or `1201`, or `122`. If we allow trailing `0`s, then almost all positional number systems are redundant, since $b_0 \dots b_{m-1}$ is always equivalent to $b_0 \dots b_{m-1}\mathtt{0}$. Therefore, we disallow trailing `0`s.

Computer representations of positional number systems can be *dense* or *sparse*. A dense representation is simply a list (or some other kind of sequence) of digits, including those digits that happen to be `0`. A sparse representation, on the other hand, includes only non-zero digits. It must then include information on either the rank (i.e., the index) or the weight of each digit. For example, Figure 6.1 shows two different representations of binary numbers in Standard ML— one dense and one sparse — along with several representative functions on each.

```
structure Dense =
struct
  datatype Digit = Zero | One
  type Nat = Digit list      (* increasing order of significance, no trailing Zeros *)

  fun inc [ ] = [One]
    | inc (Zero :: $ds$) = One :: $ds$
    | inc (One :: $ds$) = Zero :: inc $ds$     (* carry *)

  fun dec [One] = [ ]
    | dec (One :: $ds$) = Zero :: $ds$
    | dec (Zero :: $ds$) = One :: dec $ds$     (* borrow *)

  fun add ($ds$, [ ]) = $ds$
    | add ([ ], $ds$) = $ds$
    | add ($d$ :: $ds_1$, Zero :: $ds_2$) = $d$ :: add ($ds_1$, $ds_2$)
    | add (Zero :: $ds_1$, $d$ :: $ds_2$) = $d$ :: add ($ds_1$, $ds_2$)
    | add (One :: $ds_1$, One :: $ds_2$) = Zero :: inc (add ($ds_1$, $ds_2$))     (* carry *)
end
```

```
structure SparseByWeight =
struct
  type Nat = int list      (* increasing list of weights, each a power of two *)

  (* add a new weight to a list, recurse if weight is already present *)
  fun carry ($w$, [ ]) = [$w$]
    | carry ($w$, $ws$ as $w'$ :: $rest$) = if $w < w'$ then $w$ :: $ws$ else carry (2*$w$, $rest$)

  (* borrow from a digit of weight $w$, recurse if weight is not present *)
  fun borrow ($w$, $ws$ as $w'$ :: $rest$) = if $w = w'$ then $rest$ else $w$ :: borrow (2*$w$, $ws$)

  fun inc $ws$ = carry (1, $ws$)
  fun dec $ws$ = borrow (1, $ws$)

  fun add ($ws$, [ ]) = $ws$
    | add ([ ], $ws$) = $ws$
    | add ($m$ as $w_1$ :: $ws_1$, $n$ as $w_2$ :: $ws_2$) =
        if $w_1 < w_2$ then $w_1$ :: add ($ws_1$, $n$)
        else if $w_2 < w_1$ then $w_2$ :: add ($m$, $ws_2$)
        else carry (2*$w_1$, add ($ws_1$, $ws_2$))
end
```

Figure 6.1: Two implementations of binary numbers.

6.2 Binary Representations

Given a positional number system, we can implement a numerical representation based on that number system as a sequence of trees. The number and sizes of the trees representing a collection of size n are governed by the representation of n in the positional number system. For each weight w_i, there are b_i trees of that size. For example, the binary representation of 73 is `1001001`, so a collection of size 73 in a binary numerical representation would comprise three trees, of sizes 1, 8, and 64, respectively.

Trees in numerical representations typically exhibit a very regular structure. For example, in binary numerical representations, all trees have sizes that are powers of 2. Three common kinds of trees that exhibit this structure are *complete binary leaf trees* [KD96], *binomial trees* [Vui78], and *pennants* [SS90].

Definition 6.1 (Complete binary leaf trees) A complete binary tree of rank 0 is a leaf and a complete binary tree of rank $r > 0$ is a node with two children, each of which is a complete binary tree of rank $r - 1$. A leaf tree is a tree that contains elements only at the leaves, unlike ordinary trees that contain elements at every node. A complete binary tree of rank r has $2^{r+1} - 1$ nodes, but only 2^r leaves. Hence, a complete binary leaf tree of rank r contains 2^r elements.

Definition 6.2 (Binomial trees) A binomial tree of rank r is a node with r children $c_1 \dots c_r$, where c_i is a binomial tree of rank $r - i$. Alternatively, a binomial tree of rank $r > 0$ is a binomial tree of rank $r - 1$ to which another binomial tree of rank $r - 1$ has been added as the leftmost child. From the second definition, it is easy to see that a binomial tree of rank r contains 2^r nodes.

Definition 6.3 (Pennants) A pennant of rank 0 is a single node and a pennant of rank $r > 0$ is a node with a single child that is a complete binary tree of rank $r - 1$. The complete binary tree contains $2^r - 1$ elements, so the pennant contains 2^r elements.

Figure 6.2 illustrates the three kinds of trees. Which kind of tree is superior for a given data structure depends on the properties the data structure must maintain, such as the order in which elements should be stored in the trees. A key factor in the suitability of a particular kind of tree for a given data structure is how easily the tree supports functions analogous to carries and borrows in binary arithmetic. When simulating a carry, we *link* two trees of rank r to form a tree of rank $r + 1$. Symmetrically, when simulating a borrow, we *unlink* a tree of rank $r > 0$ to obtain two trees of rank $r - 1$. Figure 6.3 illustrates the link operation (denoted $\oplus$) on each of the three kinds of trees. Assuming that elements are not rearranged, each of the three kinds of trees can be linked or unlinked in $O(1)$ time.

We next describe two existing data structures in terms of this framework: the one-sided flexible arrays of Kaldewaij and Dielissen [KD96], and the binomial queues of Vuillemin [Vui78, Bro78].

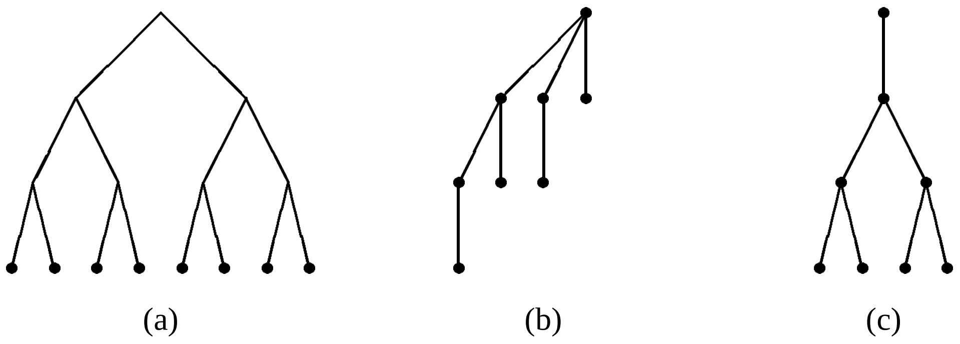

Figure 6.2: Three trees of rank 3: (a) a complete binary leaf tree, (b) a binomial tree, and (c) a pennant.

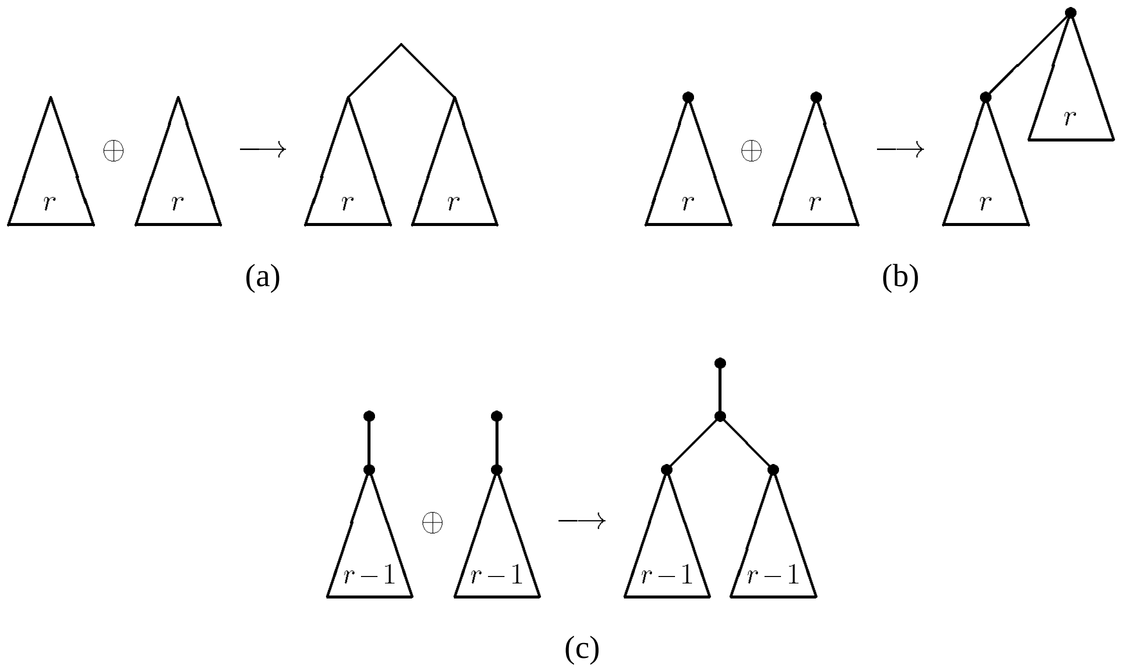

Figure 6.3: Linking two trees of rank r to obtain a tree of rank $r + 1$ for (a) complete binary leaf trees, (b) binomial trees, and (c) pennants.

```
signature RANDOMACCESSLIST =
sig
   type α RList

   exception EMPTY and INDEX

   val empty   : α RList
   val isEmpty : α RList → bool

   val cons    : α × α RList → α RList
   val head    : α RList → α                 (* raises EMPTY if list is empty *)
   val tail    : α RList → α RList           (* raises EMPTY if list is empty *)

   val lookup  : α RList × int → α           (* raises INDEX if out of bounds *)
   val update  : α RList × int × α → α RList (* raises INDEX if out of bounds *)
end
```

Figure 6.4: Signature for random-access lists.

6.2.1 Binary Random-Access Lists

A *random-access list*, also called a one-sided flexible array, is a data structure that supports array-like $lookup$ and $update$ functions, as well as the usual $cons$, $head$, and $tail$ functions on lists. A signature for random-access lists is shown in Figure 6.4.

Kaldewaij and Dielissen [KD96] describe an implementation of random-access lists in terms of leftist left-perfect leaf trees. We can easily translate their implementation into the framework of numerical representations as a binary representation using complete binary leaf trees. A binary random-access list of size n thus contains a complete binary leaf tree for each `1` in the binary representation of n. The rank of each tree corresponds to the rank of the corresponding digit; if the ith bit of n is `1`, then the random-access list contains a tree of size 2^i. For this example, we choose a dense representation, so the type of binary random-access lists is

```
datatype α Tree = Leaf of α | Node of int × α Tree × α Tree
datatype α Digit = Zero | One of α Tree
type α RList = α Digit list
```

The integer in each node is the size of the tree. This number is redundant since the size of every tree is completely determined by the size of its parent or by its position in the list of digits, but we include it anyway for convenience. Trees are stored in increasing order of size, and the order of elements (both within and between trees) is left-to-right. Thus, the head of the random-access list is the leftmost leaf of the smallest tree. Figure 6.5 shows a binary random-access list of size 7. Note that the maximum number of trees in a list of size n is $\lfloor\log(n+1)\rfloor$

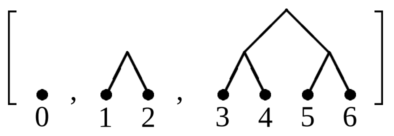

Figure 6.5: A binary random-access list containing the elements $0 \ldots 6$.

and the maximum depth of any tree is $\lfloor \log n \rfloor$.

Now, insertion into a binary random-access list (i.e., *cons*) is analogous to incrementing a binary number. Recall the increment function on dense binary numbers:

```
fun inc [ ] = [One]
  | inc (Zero :: ds) = One :: ds
  | inc (One :: ds) = Zero :: inc ds
```

To insert an element with *cons*, we first convert the element into a leaf, and then insert the leaf into the list of trees using a helper function *insTree* that follows the rules of *inc*.

```
fun cons (x, ts) = insTree (Leaf x, ts)

fun insTree (t, [ ]) = [One t]
  | insTree (t, Zero :: ts) = One t :: ts
  | insTree (t1, One t2 :: ts) = Zero :: insTree (link (t1, t2), ts)
```

The *link* helper function is a pseudo-constructor for *Node* that automatically calculates the size of the new tree from the sizes of its children.

Deleting an element from a binary random-access list (using *tail*) is analogous to decrementing a binary number. Recall the decrement function on dense binary numbers:

```
fun dec [One] = [ ]
  | dec (One :: ds) = Zero :: ds
  | dec (Zero :: ds) = One :: dec ds
```

Essentially, this function resets the first `1` to `0`, while setting all the preceding `0`s to `1`s. The analogous operation on lists of trees is *borrowTree*. When applied to a list whose first digit has rank r, *borrowTree* returns a pair containing a tree of rank r, and the new list without that tree.

```
fun borrowTree [One t] = (t, [ ])
  | borrowTree (One t :: ts) = (t, Zero :: ts)
  | borrowTree (Zero :: ts) = let val (Node (_, t1, t2), ts') = borrowTree ts
                              in (t1, One t2 :: ts') end
```

The $head$ and $tail$ functions “borrow” the leftmost leaf using $borrowTree$ and then either return that leaf's element or discard the leaf, respectively.

```
**fun** head $ts$ = **let val** (Leaf $x$, _) = borrowTree $ts$ **in** $x$ **end**
**fun** tail $ts$ = **let val** (_, $ts'$) = borrowTree $ts$ **in** $ts'$ **end**
```

The $lookup$ and $update$ functions do not have analogous arithmetic operations. Rather, they take advantage of the organization of binary random-access lists as logarithmic-length lists of logarithmic-depth trees. Looking up an element is a two-stage process. We first search the list for the correct tree, and then search the tree for the correct element. The helper function $lookupTree$ uses the size field in each node to determine whether the ith element is in the left subtree or the right subtree.

```
fun lookup (Zero :: $ts$, $i$) = lookup ($ts$, $i$)
  | lookup (One $t$ :: $ts$, $i$) =
      if $i <$ size $t$ then lookupTree ($t$, $i$) else lookup ($ts$, $i -$ size $t$)
fun lookupTree (Leaf $x$, 0) = $x$
  | lookupTree (Node ($w$, $t_1$, $t_2$), $i$) =
      if $i < w$ div 2 then lookupTree ($t_1$, $i$) else lookupTree ($t_2$, $i - w$ div 2)
```

$update$ works in same way but also reconstructs the path from the root to the updated leaf. This reconstruction is called *path copying* [ST86a] and is necessary for persistence.

```
fun update (Zero :: $ts$, $i$, $y$) = Zero :: update ($ts$, $i$, $y$)
  | update (One $t$ :: $ts$, $i$, $y$) =
      if $i <$ size $t$ then One (updateTree ($t$, $i$, $y$)) :: $ts$ else One $t$ :: update ($ts$, $i -$ size $t$, $y$)
fun updateTree (Leaf $x$, 0, $y$) = Leaf $y$
  | updateTree (Node ($w$, $t_1$, $t_2$), $i$, $y$) =
      if $i < w$ div 2 then Node ($w$, updateTree ($t_1$, $i$, $y$), $t_2$)
      else Node ($w$, $t_1$, updateTree ($t_2$, $i - w$ div 2, $y$))
```

The complete code for this implementation is shown in Figure 6.6.

$cons$, $head$, and $tail$ perform at most $O(1)$ work per digit and so run in $O(\log n)$ worst-case time. $lookup$ and $update$ take at most $O(\log n)$ time to find the right tree, and then at most $O(\log n)$ time to find the right element in that tree, for a total of $O(\log n)$ worst-case time.

6.2.2 Binomial Heaps

Binomial queues [Vui78, Bro78] are a classical implementation of mergeable priority queues. To avoid confusion with FIFO queues, we will henceforth refer to priority queues as *heaps* and binomial queues as *binomial heaps*. Heaps support four main functions: inserting an element

```
structure BinaryRandomAccessList : RANDOMACCESSLIST =
struct
  datatype $\alpha$ Tree = Leaf of $\alpha$ | Node of int $\times$ $\alpha$ Tree $\times$ $\alpha$ Tree    (* int is size of tree *)
  datatype $\alpha$ Digit = Zero | One of $\alpha$ Tree
  type $\alpha$ RList = $\alpha$ Digit list

  exception EMPTY and INDEX

  val empty = [ ]
  fun isEmpty $ts$ = null $ts$

  fun size (Leaf $x$) = 1
    | size (Node ($w$, $t_1$, $t_2$)) = $w$
  fun link ($t_1$, $t_2$) = Node (size $t_1$+size $t_2$, $t_1$, $t_2$)
  fun insTree ($t$, [ ]) = [One $t$]
    | insTree ($t$, Zero :: $ts$) = One $t$ :: $ts$
    | insTree ($t_1$, One $t_2$ :: $ts$) = Zero :: insTree (link ($t_1$, $t_2$), $ts$)
  fun borrowTree [ ] = raise EMPTY
    | borrowTree [One $t$] = ($t$, [ ])
    | borrowTree (One $t$ :: $ts$) = ($t$, Zero :: $ts$)
    | borrowTree (Zero :: $ts$) = let val (Node (_, $t_1$, $t_2$), $ts'$) = borrowTree $ts$
                                  in ($t_1$, One $t_2$ :: $ts'$) end

  fun cons ($x$, $ts$) = insTree (Leaf $x$, $ts$)
  fun head $ts$ = let val (Leaf $x$, _) = borrowTree $ts$ in $x$ end
  fun tail $ts$ = let val (_, $ts'$) = borrowTree $ts$ in $ts'$ end

  fun lookupTree (Leaf $x$, 0) = $x$
    | lookupTree (Leaf $x$, $i$) = raise INDEX
    | lookupTree (Node ($w$, $t_1$, $t_2$), $i$) =
        if $i < w$ div 2 then lookupTree ($t_1$, $i$) else lookupTree ($t_2$, $i - w$ div 2)
  fun updateTree (Leaf $x$, 0, $y$) = Leaf $y$
    | updateTree (Leaf $x$, $i$, $y$) = raise INDEX
    | updateTree (Node ($w$, $t_1$, $t_2$), $i$, $y$) =
        if $i < w$ div 2 then Node ($w$, updateTree ($t_1$, $i$, $y$), $t_2$)
        else Node ($w$, $t_1$, updateTree ($t_2$, $i - w$ div 2, $y$))

  fun lookup ([ ], $i$) = raise INDEX
    | lookup (Zero :: $ts$, $i$) = lookup ($ts$, $i$)
    | lookup (One $t$ :: $ts$, $i$) =
        if $i <$ size $t$ then lookupTree ($t$, $i$) else lookup ($ts$, $i -$ size $t$)
  fun update ([ ], $i$, $y$) = raise INDEX
    | update (Zero :: $ts$, $i$, $y$) = Zero :: update ($ts$, $i$, $y$)
    | update (One $t$ :: $ts$, $i$, $y$) =
        if $i <$ size $t$ then One (updateTree ($t$, $i$, $y$)) :: $ts$ else One $t$ :: update ($ts$, $i -$ size $t$, $y$)
end
```

Figure 6.6: Binary random-access lists.

```
signature ORDERED =
sig
  type T                    (* type of ordered elements *)
  val leq : T × T → bool    (* total ordering relation *)
end

signature HEAP =
sig
  structure Elem : ORDERED

  type Heap

  exception EMPTY

  val empty     : Heap
  val isEmpty   : Heap → bool

  val insert    : Elem.T × Heap → Heap
  val merge     : Heap × Heap → Heap

  val findMin   : Heap → Elem.T   (* raises EMPTY if heap is empty *)
  val deleteMin : Heap → Heap     (* raises EMPTY if heap is empty *)
end
```

Figure 6.7: Signature for heaps.

($insert$), merging two heaps ($merge$), finding the minimum element ($findMin$), and deleting the minimum element ($deleteMin$). A Standard ML signature for heaps appears Figure 6.7.

Remark: Heaps are similar to the sortable collections of Section 3.5.2, but use a different mechanism for specifying the desired comparison function. For sortable collections, the comparison function is supplied when a new object is created, and every object can have a different comparison function. This approach is very flexible, but causes problems in the presence of an function that combines two objects, such as $merge$. If the two objects being merged have different comparison functions, which should the resulting object keep? To avoid this ambiguity, we fix the comparison function (and therefore the type of elements being compared) when the Standard ML structure implementing heaps is created. Then, we can be sure that any two objects being merged shared the same comparison function. ◇

In the framework of numerical representations, binomial heaps are a binary representation with heap-ordered, binomial trees. A tree is *heap-ordered* if the element at every node is smaller than the elements at any of its children, with ties broken arbitrarily. As with binary random-access lists, binomial heaps contain one tree for each `1` in the binary representation of the size of the heap, and the trees have the same weights as their matching digits.

Assuming an ORDERED structure $Elem$ that specifies the element type and comparison function, the types of binomial trees and binomial heaps can be written as follows:

```
datatype Tree = Node of int × Elem.T × Tree list
type Heap = Tree list
```

This time, we have chosen a sparse representation, where the integer at each node is the rank of the tree. For reasons that will become clear later, we maintain the list of trees representing a heap in increasing order of rank, but maintain the list of trees representing the children of a node in decreasing order of rank.

Remark: The rank information on each node that is not a root is redundant since the ith child of a node of rank r always has rank $r - i$. However, we maintain this information anyway because doing so simplifies the code slightly. ◇

The fundamental operation on binomial trees is $link$, which compares the roots of two trees of rank r and makes the tree with the larger root a child of the tree with the smaller root, producing a tree of rank $r + 1$.

```
fun link ($t_1$ as Node ($r$, $x_1$, $c_1$), $t_2$ as Node (_, $x_2$, $c_2$)) =
        if Elem.leq ($x_1$, $x_2$) then Node ($r$+1, $x_1$, $t_2$ :: $c_1$) else Node ($r$+1, $x_2$, $t_1$ :: $c_2$)
```

Since the children of a tree are maintained in decreasing order of rank, adding the new child to the list takes only $O(1)$ time.

Now, inserting an element into a binomial heap is similar to the increment function on sparse binary numbers. Whenever we find two trees of the same rank, we $link$ them and reinsert the linked tree into the list. This corresponds to a carry in binary arithmetic. We use the $insTree$ helper function to insert new trees into the list of trees; $insert$ builds a new singleton tree and calls $insTree$.

```
fun insTree ($t$, [ ]) = [$t$]
  | insTree ($t_1$, $ts$ as $t_2$ :: $rest$) =
        if rank $t_1$ < rank $t_2$ then $t_1$ :: $ts$ else insTree (link ($t_1$, $t_2$), $rest$)

fun insert ($x$, $ts$) = insTree (Node (0, $x$, [ ]), $ts$)
```

$merge$ is similar to addition on sparse binary numbers, where again we $link$ trees of equal rank whenever there is a carry.

```
fun merge ($ts_1$, [ ]) = $ts_1$
  | merge ([ ], $ts_2$) = $ts_2$
  | merge ($t_1$ :: $ts_1$, $t_2$ :: $ts_2$) =
        if rank $t_1$ < rank $t_2$ then $t_1$ :: merge ($ts_1$, $t_2$ :: $ts_2$)
        else if rank $t_2$ < rank $t_1$ then $t_2$ :: merge ($t_1$ :: $ts_1$, $ts_2$)
        else insTree (link ($t_1$, $t_2$), merge ($ts_1$, $ts_2$))
```

Since every tree is heap-ordered, we know that the minimum element within any given tree is the root. However, we do not know which tree has the minimum root, so $\mathit{findMin}$ scans all the roots in the heap.

```
fun findMin [$t$] = root $t$
  | findMin ($t$ :: $ts$) = let val $x$ = root $t$
                              val $y$ = findMin $ts$
                          in if Elem.leq ($x$, $y$) then $x$ else $y$ end
```

Finally, $\mathit{deleteMin}$ begins by removing the tree with the minimum root. (In the case of ties, we should take care to remove the tree with the same root as returned by $\mathit{findMin}$.) Once we have discarded the root of this tree, we are left with two lists of trees: one representing the children of the discarded tree, and one representing the remaining trees in the heap. To obtain a single heap, we simply merge these two lists, but since the lists are maintained in opposite orders, we first reverse the list of children.

```
fun deleteMin $ts$ =
    let fun getMin [$t$] = ($t$, [ ])
          | getMin ($t$ :: $ts$) =
              let val ($t'$, $ts'$) = getMin $ts$
              in if Elem.leq (root $t$, root $t'$) then ($t$, $ts$) else ($t'$, $t$ :: $ts'$) end
        val (Node (_, $x$, $ts_1$), $ts_2$) = getMin $ts$
    in merge (rev $ts_1$, $ts_2$) end
```

The complete implementation of binomial heaps appears in Figure 6.8. Since heaps contain no more than $\lfloor \log(n+1) \rfloor$ trees, and binomial trees have no more than $\lfloor \log n \rfloor$ children, each of these functions takes $O(\log n)$ worst-case time.

6.3 Segmented Binary Numbers

We next explore two variations of binary numbers that allow a number to be incremented or decremented in $O(1)$ worst-case time. Basing a numerical representation on these variations, rather than ordinary binary numbers, reduces the running time of many insertion and deletion functions from $O(\log n)$ to $O(1)$. First, we present a somewhat complicated representation and sketch the design of random-access lists and heaps based on this representation. In the next section, we present a much simpler representation that is usually superior in practice.

The problem with ordinary binary numbers is that carries and borrows can cascade. For example, incrementing $2^k - 1$ causes k carries in binary arithmetic. Symmetrically, decrementing 2^k causes k borrows. *Segmented binary numbers* solve this problem by allowing multiple carries or borrows to be executed in a single step.

```
functor BinomialHeap (structure E : ORDERED) : HEAP =
struct
  structure Elem = E

  datatype Tree = Node of int × Elem.T × Tree list      (* the integer is the rank of the tree *)
  type Heap = Tree list

  exception EMPTY

  val empty = [ ]
  fun isEmpty ts = null ts

  fun rank (Node (r, x, c)) = r
  fun root (Node (r, x, c)) = x
  fun link (t_1 as Node (r, x_1, c_1), t_2 as Node (_, x_2, c_2)) =
        if Elem.leq (x_1, x_2) then Node (r+1, x_1, t_2 :: c_1) else Node (r+1, x_2, t_1 :: c_2)
  fun insTree (t, [ ]) = [t]
    | insTree (t_1, ts as t_2 :: rest) =
        if rank t_1 < rank t_2 then t_1 :: ts else insTree (link (t_1, t_2), rest)

  fun insert (x, ts) = insTree (Node (0, x, [ ]), ts)
  fun merge (ts_1, [ ]) = ts_1
    | merge ([ ], ts_2) = ts_2
    | merge (t_1 :: ts_1, t_2 :: ts_2) =
        if rank t_1 < rank t_2 then t_1 :: merge (ts_1, t_2 :: ts_2)
        else if rank t_2 < rank t_1 then t_2 :: merge (t_1 :: ts_1, ts_2)
        else insTree (link (t_1, t_2), merge (ts_1, ts_2))

  fun findMin [ ] = raise EMPTY
    | findMin [t] = root t
    | findMin (t :: ts) = let val x = root t
                              val y = findMin ts
                          in if Elem.leq (x, y) then x else y end

  fun deleteMin [ ] = raise EMPTY
    | deleteMin ts =
        let fun getMin [t] = (t, [ ])
              | getMin (t :: ts) =
                  let val (t', ts') = getMin ts
                  in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
            val (Node (_, x, ts_1), ts_2) = getMin ts
        in merge (rev ts_1, ts_2) end
end
```

Figure 6.8: Binomial heaps [Vui78, Bro78].

Note that incrementing a binary number takes k steps whenever the number begins with a block of k `1`s. Similarly, decrementing a binary number takes k steps whenever the number begins with a block of k `0`s. Segmented binary numbers group contiguous sequences of identical digits into blocks so that we can execute a carry or borrow on an entire block in a single step. We represent segmented binary numbers as alternating blocks of `0`s and `1`s using the following datatype:

```
datatype DigitBlock = Zeros of int | Ones of int
type Nat = DigitBlock list
```

where the integer in each $DigitBlock$ represents the block's length. Note that since we have forbidden trailing `0`s, the last block (if any) always contains `1`s.

We use the pseudo-constructors $zeros$ and $ones$ to add new blocks to the front of a list of blocks. These pseudo-constructors merge adjacent blocks of the same digit and discard empty blocks. In addition, the $zeros$ pseudo-constructor discards any trailing block of `0`s.

```
fun zeros (i, [ ]) = [ ]
  | zeros (i, Zeros j :: blks) = Zeros (i+j) :: blks
  | zeros (0, blks) = blks
  | zeros (i, blks) = Zeros i :: blks
fun ones (i, Ones j :: blks) = Ones (i+j) :: blks
  | ones (0, blks) = blks
  | ones (i, blks) = Ones i :: blks
```

Now, to increment a segmented binary number, we inspect the first block of digits (if any). If the first block contains i `0`s, then we replace the first `0` with a `1`, creating a new singleton block of `1`s and shrinking the block of `0`s by one. If the first block contains i `1`s, then we perform i carries in a single step by changing the `1`s to `0`s and incrementing the next digit.

```
fun inc [ ] = [Ones 1]
  | inc (Zeros i :: blks) = ones (1, zeros (i−1, blks))
  | inc (Ones i :: blks) = Zeros i :: inc blks
```

In the third line, we know the recursive call to inc cannot loop because the next block, if any, must contain `0`s. In the second line, the pseudo-constructors deal gracefully with the special cases that occur when the leading block contains a single `0`.

Decrementing a segmented binary number is almost exactly the same, but with the roles of `0`s and `1`s reversed.

```
fun dec (Ones i :: blks) = zeros (1, ones (i−1, blks))
  | dec (Zeros i :: blks) = Ones i :: dec blks
```

6.3.1 Segmented Binomial Random-Access Lists and Heaps

In both the binary random-access lists of Section 6.2.1 and the binomial heaps of Section 6.2.2, we linked two trees into a new, larger tree for every carry. In a cascade of k carries, we linked a new singleton tree with existing trees of sizes $2^0, 2^1, \ldots, 2^{k-1}$ to obtain a new tree of size 2^k. Similarly, in binary random-access lists, a cascade of borrows decomposes a tree of size 2^k into a singleton tree and k trees of sizes $2^0, 2^1, \ldots, 2^{k-1}$.

Segmented binary numbers support fast carries and borrows, but to take advantage of this in a numerical representation, we must choose a tree representation that will allow us to link and unlink many trees in a single step. Of the three kinds of trees described earlier, only binomial trees support this behavior. A node of rank r consists of an element and a sequence of trees of ranks $0, \ldots, r-1$. Therefore, we can combine an element and a sequence of trees into a new tree — or decompose a tree into an element and a sequence of trees — in $O(1)$ time.

Adapting the earlier implementations of binary random-access lists and binomial heaps to use segmented binary arithmetic rather than ordinary binary arithmetic, and in the case of binary random-access lists, to use binomial trees rather than complete binary leaf trees, is tedious, but mostly straightforward, except for the following issues:

- To link and unlink multiple trees in a single step, we must use the same representation for the sequence of trees corresponding to a block of `1`s (called a *segment*) and for the children of a node. So, for example, we cannot maintain one in increasing order of rank and the other in decreasing order of rank as we did for binomial heaps. For both segmented binomial heaps and segmented binomial random-access lists, we need easy access to the smallest tree in a segment, but we also need easy access to the largest child of a node. Therefore, we represent both kinds of sequences as real-time deques.

- For binomial heaps, the cascade of links that produces a new tree also compares the roots of trees as it goes to find the minimum element in the tree. For segmented binomial heaps, we do not have time to search a segment for the root with the minimum element, so we insist that the smallest tree in any segment always have the minimum root. Then, whenever we create a new tree from a new element and a segment of trees of ranks $0, \ldots, r-1$, we simply compare the new element with the first root in the segment (i.e., the root of the rank 0 tree). The smaller element becomes the new root and the larger element becomes the rank 0 child of the root. Whenever we add a new tree of rank r to a segment whose smallest tree has rank $r+1$, we decompose the tree of rank $r+1$ into two trees of rank r. We then keep the tree with the smallest root, and link the remaining two trees into a new tree of rank $r+1$.

With these changes segmented binomial random-access lists support $cons$, $head$, and $tail$ in $O(1)$ worst-case time, and $lookup$ and $update$ in $O(\log n)$ worst-case time. Segmented binomial heaps support $insert$ in $O(1)$ worst-case time, and $merge$, $findMin$, and $deleteMin$ in $O(\log n)$ worst-case time.

6.4 Skew Binary Numbers

Numerical representations based on segmented binary numbers rather than ordinary binary numbers improve the asymptotic behavior of certain operations from $O(\log n)$ to $O(1)$, while retaining the same asymptotic behavior for all other operations. Unfortunately, such data structures are too complicated to be useful in practice. We next consider another number system, *skew binary numbers*, that usually achieves similar asymptotic benefits, but that is simpler and faster in practice.

In skew binary numbers [Mye83, Oka95b], the weight w_i of the ith digit is $2^{i+1} - 1$, rather than 2^i as in ordinary binary numbers. Digits may be `0`, `1`, or `2` (i.e., $D_i = \{0, 1, 2\}$). For example, the decimal number 92 could be written `002101` (least-significant digit first).

This number system is redundant, but, if we add the further constraint that only the lowest non-`0` digit may be `2`, then we regain unique representations. Such a number is said to be in *canonical form*. Henceforth, we will assume that all skew binary numbers are in canonical form.

Theorem 6.1 (Myers [Mye83]) *Every natural number has a unique skew binary canonical form.*

Recall that the weight of digit i is $2^{i+1} - 1$ and note that $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$. This implies that we can increment a skew binary number whose lowest non-`0` digit is `2` by resetting the `2` to `0` and incrementing the next digit from `0` to `1` or from `1` to `2`. (The next digit cannot already be `2`.) Incrementing a skew binary number that does not contain a `2` is even easier — simply increment the lowest digit from `0` to `1` or from `1` to `2`. In both cases, the result is still in canonical form. And, assuming we can find the lowest non-`0` digit in $O(1)$ time, both cases take only $O(1)$ time!

We cannot use a dense representation for skew binary numbers since scanning for the lowest non-`0` digit would take more than $O(1)$ time. Instead, we choose a sparse representation, so that we always have immediate access to the lowest non-`0` digit.

```
type Nat = int list
```

The integers represent either the rank or weight of each non-`0` digit. For now, we use weights. The weights are stored in increasing order, except that the smallest two weights may be identical, indicating that the lowest non-`0` digit is `2`. Given this representation, we implement inc as follows:

```
fun inc ($ws$ as $w_1$ :: $w_2$ :: $rest$) =
        if $w_1$ = $w_2$ then (1+$w_1$+$w_2$) :: $rest$ else 1 :: $ws$
    | inc $ws$ = 1 :: $ws$
```

The first clause checks whether the first two weights are equal and then either combines the weights into the next larger weight (incrementing the next digit) or adds a new weight of 1 (incrementing the smallest digit). The second clause handles the case that ws is empty or contains only a single weight. Clearly, inc runs in only $O(1)$ worst-case time.

Decrementing a skew binary number is just as easy as incrementing a number. If the lowest digit is non-`0`, then we simply decrement that digit from `2` to `1` or from `1` to `0`. Otherwise, we decrement the lowest non-`0` digit and reset the previous `0` to `2`. This can be implemented as follows:

```
fun dec (1 :: $ws$) = $ws$
    | dec ($w$ :: $ws$) = ($w$ div 2) :: ($w$ div 2) :: $ws$
```

In the second line, note that if $w = 2^{k+1} - 1$, then $\lfloor w/2 \rfloor = 2^k - 1$. Clearly, dec also runs in only $O(1)$ worst-case time.

6.4.1 Skew Binary Random-Access Lists

We next design a numerical representation for random-access lists, based on skew binary numbers. The basic representation is a list of trees, with one tree for each `1` digit and two trees for each `2` digit. The trees are maintained in increasing order of size, except that the smallest two trees are the same size when the lowest non-`0` digit is `2`.

The sizes of the trees should correspond to the weights in skew binary numbers, so a tree representing the ith digit should have size $2^{i+1} - 1$. Up until now, we have mainly considered trees whose sizes are powers of two, but we have also encountered a kind of tree whose sizes have the desired form: complete binary trees. Therefore, we represent skew binary random-access lists as lists of complete binary trees.

To support $head$ efficiently, the first element in the random-access list should be the root of the first tree, so we store the elements within each tree in left-to-right preorder and with the elements in each tree preceding the elements in the next tree.

In previous examples, we have stored a size or rank in every node, even when that information was redundant. For this example, we adopt the more realistic approach of maintaining size information only for the root of each tree in the list, and not for every subtree as well. The type of skew binary random-access lists is therefore

```
datatype $\alpha$ Tree = Leaf of $\alpha$ | Node of $\alpha \times \alpha$ Tree $\times\ \alpha$ Tree
type $\alpha$ RList = (int $\times\ \alpha$ Tree) list
```

Now, we can define $cons$ in analogy to inc.

```
fun cons ($x$, $ts$ as ($w_1$, $t_1$) :: ($w_2$, $t_2$) :: $rest$) =
      if $w_1$ = $w_2$ then (1+$w_1$+$w_2$, Node ($x$, $t_1$, $t_2$)) :: $rest$) else (1, Leaf $x$) :: $ts$
  | cons ($x$, $ts$) = (1, Leaf $x$) :: $ts$
```

$head$ and $tail$ inspect and remove the root of the first tree. $tail$ returns the children of the root (if any) back to the front of the list, where they represent a new 2 digit.

```
fun head ((1, Leaf $x$) :: $ts$) = $x$
  | head (($w$, Node ($x$, $t_1$, $t_2$)) :: $ts$) = $x$
fun tail ((1, Leaf $x$) :: $ts$) = $ts$
  | tail (($w$, Node ($x$, $t_1$, $t_2$)) :: $ts$) = ($w$ div 2, $t_1$) :: ($w$ div 2, $t_2$) :: $ts$
```

To lookup an element, we first search the list for the right tree, and then search the tree for the right element.

```
fun lookup (($w$, $t$) :: $ts$, $i$) = if $i < w$ then lookupTree ($w$, $t$, $i$) else lookup ($ts$, $i-w$)

fun lookupTree (1, Leaf $x$, 0) = $x$
  | lookupTree ($w$, Node ($x$, $t_1$, $t_2$), 0) = $x$
  | lookupTree ($w$, Node ($x$, $t_1$, $t_2$), $i$) =
      if $i < w$ div 2 then lookupTree ($w$ div 2, $t_1$, $i-1$)
      else lookupTree ($w$ div 2, $t_2$, $i - 1 - w$ div 2)
```

Note that in the penultimate line, we subtract one from i because we have skipped over x. In the last line, we subtract $1 + \lfloor w/2 \rfloor$ from i because we have skipped over x and all the elements in t_1. $update$ and $updateTree$ are defined similarly, and are shown in Figure 6.9, which contains the complete implementation.

It is easy to verify that $cons$, $head$, and $tail$ run in $O(1)$ worst-case time. Like binary random-access lists, skew binary random-access lists are logarithmic-length lists of logarithmic-depth trees. Hence, $lookup$ and $update$ run in $O(\log n)$ worst-case time. In fact, every unsuccessful step of $lookup$ or $update$ discards at least one element, so this bound can be improved slightly to $O(\min(i, \log n))$.

Hint to Practitioners: Skew binary random-access lists are a good choice for applications that take advantage of both the list-like aspects and the array-like aspects of random-access lists. Although there are better implementations of lists, and better implementations of (persistent) arrays, none are better at both [Oka95b].

```
structure SkewBinaryRandomAccessList : RANDOMACCESSLIST =
struct
  datatype $\alpha$ Tree = Leaf of $\alpha$ | Node of $\alpha \times \alpha$ Tree $\times \alpha$ Tree
  type $\alpha$ RList = (int $\times \alpha$ Tree) list        (* integer is the weight of the tree *)

  exception EMPTY and INDEX

  val empty = [ ]
  fun isEmpty $ts$ = null $ts$

  fun cons ($x$, $ts$ as ($w_1$, $t_1$) :: ($w_2$, $t_2$) :: $ts'$) =
        if $w_1 = w_2$ then (1+$w_1$+$w_2$, Node ($x$, $t_1$, $t_2$)) :: $ts'$) else (1, Leaf $x$) :: $ts$
    | cons ($x$, $ts$) = (1, Leaf $x$) :: $ts$
  fun head [ ] = raise EMPTY
    | head ((1, Leaf $x$) :: $ts$) = $x$
    | head (($w$, Node ($x$, $t_1$, $t_2$)) :: $ts$) = $x$
  fun tail [ ] = raise EMPTY
    | tail ((1, Leaf $x$) :: $ts$) = $ts$
    | tail (($w$, Node ($x$, $t_1$, $t_2$)) :: $ts$) = ($w$ div 2, $t_1$) :: ($w$ div 2, $t_2$) :: $ts$

  fun lookupTree (1, Leaf $x$, 0) = $x$
    | lookupTree (1, Leaf $x$, $i$) = raise INDEX
    | lookupTree ($w$, Node ($x$, $t_1$, $t_2$), 0) = $x$
    | lookupTree ($w$, Node ($x$, $t_1$, $t_2$), $i$) =
        if $i < w$ div 2 then lookupTree ($w$ div 2, $t_1$, $i-1$)
        else lookupTree ($w$ div 2, $t_2$, $i - 1 - w$ div 2)
  fun updateTree (1, Leaf $x$, 0, $y$) = Leaf $y$
    | updateTree (1, Leaf $x$, $i$, $y$) = raise INDEX
    | updateTree ($w$, Node ($x$, $t_1$, $t_2$), 0, $y$) = Node ($y$, $t_1$, $t_2$)
    | updateTree ($w$, Node ($x$, $t_1$, $t_2$), $i$, $y$) =
        if $i < w$ div 2 then Node ($x$, updateTree ($w$ div 2, $t_1$, $i-1$, $y$), $t_2$)
        else Node ($x$, $t_1$, updateTree ($w$ div 2, $t_2$, $i - 1 - w$ div 2, $y$))

  fun lookup ([ ], $i$) = raise INDEX
    | lookup (($w$, $t$) :: $ts$, $i$) = if $i < w$ then lookupTree ($w$, $t$, $i$) else lookup ($ts$, $i-w$)
  fun update ([ ], $i$, $y$) = raise INDEX
    | update (($w$, $t$) :: $ts$, $i$, $y$) =
        if $i < w$ then updateTree ($w$, $t$, $i$, $y$) :: $ts$ else ($w$, $t$) :: update ($ts$, $i-w$, $y$)
end
```

Figure 6.9: Skew binary random-access lists.

6.4.2 Skew Binomial Heaps

Finally, we consider a hybrid numerical representation for heaps based on both skew binary numbers and ordinary binary numbers. Incrementing a skew binary number is both quick and simple, and serves admirably as a template for the $insert$ function. Unfortunately, addition of two arbitrary skew binary numbers is awkward. We therefore base the $merge$ function on ordinary binary addition, rather than skew binary addition.

A *skew binomial tree* is a binomial tree in which every node is augmented with a list of up to r elements, where r is the rank of the node in question.

```
datatype Tree = Node of int × Elem.T × Elem.T list × Tree list
```

Unlike ordinary binomial trees, the size of a skew binomial tree is not completely determined by its rank; rather the rank of a skew binomial tree determines a range of possible sizes.

Lemma 6.2 *If t is a skew binomial tree of rank r, then $2^r \leq |t| \leq 2^{r+1} - 1$.*

Proof: By induction. t has r children $t_1 \ldots t_r$, where t_i is a skew binomial tree of rank $r - i$, and $2^{r-i} \leq |t_i| \leq 2^{r-i+1} - 1$. In addition, the root of t is augmented with a list of k elements, where $0 \leq k \leq r$. Therefore, $|t| \geq 1 + 0 + \sum_{i=0}^{r-1} 2^i = 1 + (2^r - 1) = 2^r$ and $|t| \leq 1 + r + \sum_{i=0}^{r-1}(2^{i+1} - 1) = 1 + r + (2^{r+1} - r - 2) = 2^{r+1} - 1$. □

Note that a tree of rank r is always larger than a tree of rank $r - 1$.

Skew binomial trees may be *linked* or *skew linked*. The $link$ function combines two trees of rank r to form a tree of rank $r + 1$ by making the tree with the larger root a child of the tree with the smaller root.

```
fun link (t1 as Node (r, x1, xs1, c1), t2 as Node (_, x2, xs2, c2)) =
        if Elem.leq (x1, x2) then Node (r+1, x1, xs1, t2 :: c1) else Node (r+1, x2, xs2, t1 :: c2)
```

The $skewLink$ function combines two trees of rank r with an additional element to form a tree of rank $r + 1$ by first linking the two trees, and then comparing the root of the resulting tree with the additional element. The smaller of the two elements remains as the root, and the larger is added to the auxiliary list of elements.

```
fun skewLink (x, t1, t2) =
        let val Node (r, y, ys, c) = link (t1, t2)
        in if Elem.leq (x, y) then Node (r, x, y :: ys, c) else Node (r, y, x :: ys, c) end
```

A skew binomial heap is represented as a list of heap-ordered skew binomial trees of increasing rank, except that the first two trees may share the same rank. Since skew binomial trees of the same rank may have different sizes, there is no longer a direct correspondence between the trees in the heap and the digits in the skew binary number representing the size of the heap. For example, even though the skew binary representation of 4 is `11`, a skew binomial heap of size 4 may contain one rank 2 tree of size 4; two rank 1 trees, each of size 2; a rank 1 tree of size 3 and a rank 0 tree; or a rank 1 tree of size 2 and two rank 0 trees. However, the maximum number of trees in a heap is still $O(\log n)$.

The big advantage of skew binomial heaps is that we can insert a new element in $O(1)$ time. We first compare the ranks of the two smallest trees. If they are the same, we skew link the new element with these two trees. Otherwise, we simply add a new singleton tree to the list.

```
fun insert (x, ts as t1 :: t2 :: rest) =
        if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest else Node (0, x, [ ], [ ]) :: ts
    | insert (x, ts) = Node (0, x, [ ], [ ]) :: ts
```

We implement $merge$ in terms of two helper functions, $insTree$ and $mergeTrees$, that behave exactly like their counterparts from ordinary binomial heaps, performing a regular link (not a skew link!) whenever they find two trees of equal rank. Since $mergeTrees$ expects lists of strictly increasing rank, $merge$ normalizes its two arguments to remove any leading duplicates before calling $mergeTrees$.

```
fun normalize [ ] = [ ]
    | normalize (t :: ts) = insTree (t, ts)
fun merge (ts1, ts2) = mergeTrees (normalize ts1, normalize ts2)
```

$findMin$ also behaves exactly like its counterpart from ordinary binomial heaps; since it ignores the rank of each tree, it is unaffected by the possibility that the first two trees might have the same rank. It simply scans the list of trees for the minimum root.

```
fun findMin [t] = root t
    | findMin (t :: ts) = let val x = root t
                              val y = findMin ts
                          in if Elem.leq (x, y) then x else y end
```

Finally, $deleteMin$ on skew binomial heaps is similar to its counterpart for ordinary binomial heaps except that it must deal with the list of auxiliary elements that has been added to every node. We first find and remove the tree with the minimum root. After discarding this root, we merge the children of this root with the remaining trees. To do so, we must first reverse the list of children, since it is stored in decreasing order, and normalize the list of trees, since the first rank might be duplicated. Finally, we reinsert each of the elements from the auxiliary list.

```
fun deleteMin ts =
        let fun getMin [t] = (t, [ ])
                | getMin (t :: ts) = let val (t', ts') = getMin ts
                                     in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
        val (Node (_, $x$, $xs$, $c$), $ts'$) = getMin $ts$
        fun insertAll ([ ], $ts$) = $ts$
          | insertAll ($x$ :: $xs$, $ts$) = insertAll ($xs$, insert ($x$, $ts$))
    in insertAll ($xs$, mergeTrees (rev $c$, normalize $ts'$)) end
```

Figure 6.10 presents the complete implementation of skew binomial heaps.

insert clearly runs in $O(1)$ worst-case time, while *merge*, *findMin*, and *deleteMin* run in the same time as their counterparts for ordinary binomial queues, i.e., $O(\log n)$ worst-case time each. Note that the various phases of *deleteMin* — finding the tree with the minimum root, reversing the children, merging the children with the remaining trees, and reinserting the auxiliary elements — take $O(\log n)$ time each.

6.5 Discussion

In designing numerical representations, we draw analogies between container data structures and representations of natural numbers. However, this analogy can also be extended to other kinds of numbers. For example, *difference lists* [SS86] in Prolog support a notion of lists with negative length; appending a list of length 15 and a list of length -10 results in a list of length 5. This behavior is also possible using the catenable lists of Hughes [Hug86], which are the functional counterpart of difference lists.[1]

As another example, Brodal and Okasaki [BO96] support a *delete* function on heaps using two primitive heaps, one containing positive elements and one containing negative elements. The negative elements are ones that have been deleted, but that have not yet been physically removed from the positive heap. In this representation, it is possible to delete elements that have not yet been inserted. If the negative heap is larger than the positive heap, then the overall "size" of the heap is negative.

Can this analogy between data structures and representations of numbers be extended even further, to non-integral numbers? We know of no such examples, but it is intriguing to speculate on possible uses for such data structures. For instance, might a numerical representation based on floating point numbers be useful in approximation algorithms?

6.6 Related Work

Numerical Representations Data structures that can be cast as numerical representations are surprisingly common, but only rarely is the connection to a variant number system noted explicitly [GMPR77, Mye83, CMP88, KT96b].

[1]Thanks to Phil Wadler for this observation.

```
functor SkewBinomialHeap (structure E : ORDERED) : HEAP =
struct
  structure Elem = E

  datatype Tree = Node of int × Elem.T × Elem.T list × Tree list
  type Heap = Tree list

  exception EMPTY

  val empty = [ ]
  fun isEmpty ts = null ts

  fun rank (Node (r, x, xs, c)) = r
  fun root (Node (r, x, xs, c)) = x
  fun link (t1 as Node (r, x1, xs1, c1), t2 as Node (_, x2, xs2, c2)) =
        if Elem.leq (x1, x2) then Node (r+1, x1, xs1, t2 :: c1) else Node (r+1, x2, xs2, t1 :: c2)
  fun skewLink (x, t1, t2) =
        let val Node (r, y, ys, c) = link (t1, t2)
        in if Elem.leq (x, y) then Node (r, x, y :: ys, c) else Node (r, y, x :: ys, c) end
  fun insTree (t, [ ]) = [t]
    | insTree (t1, t2 :: ts) = if rank t1 < rank t2 then t1 :: t2 :: ts else insTree (link (t1, t2), ts)
  fun mergeTrees (ts1, [ ]) = ts1
    | mergeTrees ([ ], ts2) = ts2
    | mergeTrees (t1 :: ts1, t2 :: ts2) = if rank t1 < rank t2 then t1 :: mergeTrees (ts1, t2 :: ts2)
                                          else if rank t2 < rank t1 then t2 :: mergeTrees (t1 :: ts1,ts2)
                                          else insTree (link (t1, t2), mergeTrees (ts1, ts2))
  fun normalize [ ] = [ ]
    | normalize (t :: ts) = insTree (t, ts)

  fun insert (x, ts as t1 :: t2 :: rest) =
        if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest else Node (0, x, [ ], [ ]) :: ts
    | insert (x, ts) = Node (0, x, [ ], [ ]) :: ts
  fun merge (ts1, ts2) = mergeTrees (normalize ts1, normalize ts2)
  fun findMin [ ] = raise EMPTY
    | findMin [t] = root t
    | findMin (t :: ts) = let val x = root t and y = findMin ts
                          in if Elem.leq (x, y) then x else y end

  fun deleteMin [ ] = raise EMPTY
    | deleteMin ts =
        let fun getMin [t] = (t, [ ])
              | getMin (t :: ts) = let val (t', ts') = getMin ts
                                   in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
            val (Node (_, x, xs, c), ts') = getMin ts
            fun insertAll ([ ], ts) = ts
              | insertAll (x :: xs, ts) = insertAll (xs, insert (x, ts))
        in insertAll (xs, mergeTrees (rev c, normalize ts')) end
end
```

Figure 6.10: Skew binomial heaps.

Random-Access Lists Random-access lists are usually implemented in purely functional languages as balanced trees, such as AVL trees [Mye84], Braun trees [Hoo92a, Pau91], or leftist left-perfect leaf trees [KD96]. Such trees easily support $O(\log n)$ lookups and updates ($O(\log i)$ in the case of Braun trees), but require $O(\log n)$ time for $cons$ or $tail$.

Myers [Mye83] describes the first implementation of random-access lists based on skew binary numbers. He augments a standard singly-linked list with auxiliary pointers allowing one to skip arbitrarily far ahead in the list. The number of elements skipped by each auxiliary pointer is controlled by the digits of a skew binary number. His scheme supports $cons$, $head$, and $tail$ in $O(1)$ time, and $lookup$ in $O(\log n)$ time, but requires $O(i)$ time for $update$. The difficulty with updates is that his scheme contains many redundant pointers. Removing those redundant pointers yields a structure isomorphic to the skew binary random-access lists of Section 6.4.1, which first appeared in [Oka95b].

Kaplan and Tarjan [KT95] recently introduced the algorithmic notion of recursive slow-down, and used it to design a new, purely functional implementation of real-time deques. A pleasant accidental property of their data structure is that it also supports random access in $O(\log d)$ worst-case time, where d is the distance from the desired element to the nearest end of the deque (i.e., $d = \min(i, n - 1 - i)$). We will consider a simplification of their data structure in Chapter 8.

Finger search trees [GMPR77, Tsa85] support not only random access in $O(\log d)$ worst-case time, but also insertions and deletions at arbitrary locations. Kaplan and Tarjan apply their methods to purely functional finger search trees in [KT96b].

Binomial Heaps Binomial heaps were introduced by Vuillemin [Vui78] and extensively studied by Brown [Bro78]. King [Kin94] showed that binomial heaps could be implemented elegantly in a purely functional language (in his case, Haskell).

Fagerberg [Fag96] describes a generalization of binomial heaps in which the set D_i of allowable digits at position i in a sequence of digits can be different for each i. Varying the choices for each D_i allows a tradeoff between the costs of $insert$ and $meld$, and the cost of $deleteMin$.

Skew binomial heaps were originally presented, in a slightly different form, in [BO96].

Chapter 7

Data-Structural Bootstrapping

The term *bootstrapping* refers to "pulling yourself up by your bootstraps". This seemingly nonsensical image is representative of a common situation in computer science: problems whose solutions require solutions to (simpler) instances of the same problem.

For example, consider loading an operating system from disk or tape onto a bare computer. Without an operating system, the computer cannot even read from the disk or tape! One solution is a *bootstrap loader*, a very tiny, incomplete operating system whose only purpose is to read in and pass control to a somewhat larger, more capable operating system that in turn reads in and passes control to the actual, desired operating system. This can be viewed as a instance of bootstrapping a complete solution from an incomplete solution.

Another example is bootstrapping a compiler. A common activity is to write the compiler for a new language in the language itself. But then how do you compile that compiler? One solution is to write a very simple, inefficient interpreter for the language in some other, existing language. Then, using the interpreter, you can execute the compiler on itself, thereby obtaining an efficient, compiled executable for the compiler. This can be viewed as an instance of bootstrapping an efficient solution from an inefficient solution.

In his thesis [Buc93], Adam Buchsbaum describes two algorithmic design techniques he collectively calls *data-structural bootstrapping*. The first technique, *structural decomposition*, involves bootstrapping complete data structures from incomplete data structures. The second technique, *structural abstraction*, involves bootstrapping efficient data structures from inefficient data structures. In this chapter, we reexamine data-structural bootstrapping, and describe several functional data structures based on these techniques.

7.1 Structural Decomposition

Structural decomposition is a technique for bootstrapping complete data structures from incomplete data structures. Typically, this involves taking an implementation that can handle objects only up to some bounded size (perhaps even zero), and extending it to handle objects of unbounded size.

Consider typical recursive datatypes such as lists and binary leaf trees:

```
datatype α List = Nil | Cons of α × α List
datatype α Tree = Leaf of α | Node of α Tree × α Tree
```

In some ways, these can be regarded as instances of structural decomposition. Both consist of a simple implementation of objects of some bounded size (zero for lists and one for trees) and a rule for recursively decomposing larger objects into smaller objects until eventually each object is small enough to be handled by the bounded case.

However, both of these definitions are particularly simple in that the recursive component in each definition is identical to the type being defined. For instance, the recursive component in the definition of $\alpha\ List$ is also $\alpha\ List$. Such a datatype is called *uniformly recursive*.

In general, we reserve the term *structural decomposition* to describe recursive data structures that are *non-uniform*. For example, consider the following definition of sequences:

```
datatype α Seq = Empty | Seq of α × (α × α) Seq
```

Here, a sequence is either empty or a single element together with a sequence of pairs of elements. The recursive component $(\alpha \times \alpha)\ Seq$ is different from $\alpha\ Seq$ so this datatype is non-uniform. (In Chapter 8, we will consider an implementation of queues that is similar to this definition of sequences.)

Why might such a non-uniform definition be preferable to a uniform definition? The more sophisticated structure of non-uniform types often supports more efficient algorithms than their uniform cousins. For example, compare the following $size$ functions on lists and sequences.

```
fun sizeL Nil = 0                              fun sizeS Empty = 0
  | sizeL (Cons (x, xs)) = 1 + sizeL xs          | sizeS (Seq (x, ps)) = 1 + 2 ∗ sizeS ps
```

The function on lists runs in $O(n)$ time whereas the function on sequences runs in $O(\log n)$ time.

7.1.1 Non-Uniform Recursion and Standard ML

Although Standard ML allows the definition of non-uniform recursive datatypes, the type system disallows the definition of most useful functions on such datatypes. For instance, consider the $sizeS$ function on sequences. This function definition would be rejected by Standard ML because the type system requires that all recursive calls in the body of a recursive function have the same type as the enclosing function (i.e., recursive function definitions must be uniform). The $sizeS$ function violates this restriction because the enclosing $sizeS$ has type $\alpha\ Seq \rightarrow int$ but the inner $sizeS$ has type $(\alpha \times \alpha)\ Seq \rightarrow int$.

It is usually possible to convert a non-uniform type into a uniform type by introducing a new datatype to collapse the different instances into a single type. For example, by collapsing elements and pairs, the Seq type could be written

```
datatype α ElemOrPair = Elem of α | Pair of α ElemOrPair × α ElemOrPair
datatype α Seq = Empty | Seq of α ElemOrPair × α Seq
```

Then the $sizeS$ function would be perfectly legal as written; both the enclosing $sizeS$ and the inner $sizeS$ would have type $\alpha\ Seq \rightarrow int$.

Although necessary to satisfy the Standard ML type system, this solution is unsatisfactory in at least two ways. First, the programmer must manually insert $Elem$ and $Pair$ constructors everywhere. This is tedious and error-prone. Second, and more importantly, this definition of Seq is not isomorphic to the earlier, non-uniform definition of Seq. In particular, the first definition ensures that the outermost Seq constructor contains a single element, the second a pair of elements, the third a pair of pairs of elements, and so on. However, the second definition makes no such restriction; elements and pairs may be freely mixed. If such a restriction is desired, the programmer must establish it as a system invariant. But if the programmer accidentally violates this invariant — say, by using an element where a pair is expected — the type system will be of no help in catching the error.

For these reasons, we will often present code as if Standard ML supported non-uniform recursive function definitions, also known as *polymorphic recursion* [Myc84]. This code will not be executable but will be easier to read. We will then sketch the coercions necessary to eliminate the polymorphic recursion and make the code executable.

7.1.2 Queues Revisited

Consider the use of $+\!\!+$ in the banker's queues of Section 3.4.2. During a rotation, the front stream F is replaced by $F +\!\!+ reverse\ R$. After a series of rotations, F will have the form

$$(\cdots((f +\!\!+ reverse\ r_1) +\!\!+ reverse\ r_2) \cdots +\!\!+ reverse\ r_k)$$

Append is well-known to be inefficient in left-associative contexts like this because it repeatedly processes the elements of the leftmost streams. For example, in this case, the elements of f will be processed k times (once by each $+\!\!+$), and the elements of r_i will be processed $k-i+1$ times (once by $reverse$ and once for each following $+\!\!+$). In general, left-associative appends can easily lead to quadratic behavior. In this case, fortunately, the total cost of the appends is still linear because each r_i is at least twice as long as the one before. Still, this repeated processing does sometimes make these queues slow in practice. In this section, we use structural decomposition to eliminate this inefficiency.

Given that F has the above form and writing R as r, we can decompose a queue into three parts: f, r, and the collection $m = \{reverse\ r_1, \ldots, reverse\ r_k\}$. Previously, f, r, and each $reverse\ r_i$ was a stream, but now we can represent f and r as ordinary lists and each $reverse\ r_i$ as a suspended list. This eliminates the vast majority of suspensions and avoids almost all of the overheads associated with lazy evaluation. But how should we represent the collection m? As we will see, this collection is accessed in FIFO order, so using structural decomposition we can represent it as a queue of suspended lists. As with any recursive type, we need a base case, so we will represent empty queues with a special constructor.[1] The new representation is therefore

```
datatype α Queue =
        Empty
      | Queue of {F : α list, M : α list susp Queue, LenFM : int, R : α list, LenR : int}
```

$LenFM$ is the combined length of F and all the suspended lists in M (i.e., what used to be simply $LenF$ in the old representation). R can never be longer than this combined length. In addition, F must always be non-empty. (In the old representation, F could be empty if the entire queue was empty, but now we represent that case separately.)

As always, the queue functions are simple to describe.

```
fun snoc (Empty, x) = Queue {F = [x], M = Empty, LenFM = 1, R = [ ], LenR = 0}
  | snoc (Queue {F = f, M = m, LenFM = lenFM, R = r, LenR = lenR}, x) =
      queue {F = f, M = m, LenFM = lenFM, R = x :: r, LenR = lenR+1}
fun head (Queue {F = x :: f, ...}) = x
fun tail (Queue {F = x :: f, M = m, LenFM = lenFM, R = r, LenR = lenR}) =
      queue {F = f, M = m, LenFM = lenFM−1, R = r, LenR = lenR}
```

The real action is in the pseudo-constructor $queue$. If R is too long, $queue$ creates a suspension to reverse R and adds the suspension to M. After checking the length of R, $queue$ invokes a helper function $checkF$ that guarantees that F is non-empty. If both F and M are empty, then the entire queue is empty. Otherwise, if F is empty we remove the first suspension from M, force it, and install the resulting list as the new F.

```
fun queue (q as {F = f, M = m, LenFM = lenFM, R = r, LenR = lenR}) =
      if lenR ≤ lenFM then checkF q
      else checkF {F = f, M = snoc (m, $rev r), LenFM = lenFM+lenR, R = [ ], LenR = 0}
```

[1]A slightly more efficient alternative is to represent queues up to some fixed size simply as lists.

```
structure BootstrappedQueue : QUEUE =    (* assumes polymorphic recursion! *)
struct
  datatype α Queue =
        Empty
      | Queue of {F : α list, M : α list susp Queue, LenFM : int, R : α list, LenR : int }

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty
    | isEmpty (Queue _) = false

  fun queue (q as {F = f, M = m, LenFM = lenFM, R = r, LenR = lenR}) =
        if lenR ≤ lenFM then checkF q
        else checkF {F = f, M = snoc (m, $rev r), LenFM = lenFM+lenR, R = [ ], LenR = 0}
  and checkF {F = [ ], M = Empty, ...} = Empty
    | checkF {F = [ ], M = m, LenFM = lenFM, R = r, LenR = lenR}) =
        Queue {F = force (head m), M = tail m, LenFM = lenFM, R = r, LenR = lenR}
    | checkF q = Queue q

  and snoc (Empty, x) = Queue {F = [x], M = Empty, LenFM = 1, R = [ ], LenR = 0}
    | snoc (Queue {F = f, M = m, LenFM = lenFM, R = r, LenR = lenR}, x) =
        queue {F = f, M = m, LenFM = lenFM, R = x :: r, LenR = lenR+1}
  and head Empty = raise EMPTY
    | head (Queue {F = x :: f, ...}) = x
  and tail Empty = raise EMPTY
    | tail (Queue {F = x :: f, M = m, LenFM = lenFM, R = r, LenR = lenR}) =
        queue {F = f, M = m, LenFM = lenFM−1, R = r, LenR = lenR}
end
```

Figure 7.1: Bootstrapped queues based on structural decomposition.

```
  and checkF {F = [ ], M = Empty, ...} = Empty
    | checkF {F = [ ], M = m, LenFM = lenFM, R = r, LenR = lenR}) =
        Queue {F = force (head m), M = tail m, LenFM = lenFM, R = r, LenR = lenR}
    | checkF q = Queue q
```

Note that *queue* and *checkF* call *snoc* and *tail*, which in turn call *queue*. These functions must therefore all be defined mutually recursively. The complete implementation appears in Figure 7.1.

Remark: To implement these queues without polymorphic recursion, we redefine the datatype as

```
datatype α ElemOrList = Elem of α | List of α ElemOrList list susp
datatype α Queue =
         Empty
        | Queue of {F : α ElemOrList list, M : α Queue, LenFM : int,
                    R : α ElemOrList list, LenR : int }
```

Then $snoc$ and $head$ add and remove the $Elem$ constructor when inserting or inspecting an element, and $queue$ and $checkF$ add and remove the $List$ constructor when inserting or removing a list from M. ◇

These queues create a suspension to reverse the rear list at exactly the same time as banker's queues, and force the suspension one operation earlier than banker's queues. Thus, since the reverse computation contributes only $O(1)$ amortized time to each operation on banker's queues, it also contributes only $O(1)$ amortized time to each operation on bootstrapped queues. However, the running time of the $snoc$ and $tail$ operations is not constant! Note that $snoc$ calls $queue$, which in turn might call $snoc$ on M. In this way we might get a cascade of calls to $snoc$, one at each level of the queue. However, successive lists in M at least double in size so the length of M is $O(\log n)$. Since the size of the middle queue decreases by at least a logarithmic factor at each level, the entire queue can only have depth $O(\log^* n)$. $snoc$ performs $O(1)$ amortized work at each level, so in total $snoc$ requires $O(\log^* n)$ amortized time.

Similarly, $tail$ might result in recursive calls to both $snoc$ and $tail$. The $snoc$ might in turn recursively call $snoc$ and the $tail$ might recursively call both $snoc$ and $tail$. However, for any given level, $snoc$ and $tail$ can not both recursively call $snoc$. Therefore, both $snoc$ and $tail$ are each called at most once per level. Since both $snoc$ and $tail$ do $O(1)$ amortized work at each level, the total amortized cost of $tail$ is also $O(\log^* n)$.

Remark: $O(\log^* n)$ is constant in practice. To have a depth of more than five, a queue would need to contain at least 2^{65536} elements. In fact, if one represents queues of up to size 4 simply as lists, then all queues with fewer than about 4 billion elements will have no more than three levels. ◇

Although it makes no difference in practice, one could reduce the amortized running time of $snoc$ and $tail$ to $O(1)$ by wrapping M in a suspension and executing all operations on M lazily. The type of M then becomes $\alpha\ list\ susp\ Queue\ susp$.

Yet another variation that yields $O(1)$ behavior is to abandon structural decomposition and simply use a stream of type $\alpha\ list\ susp\ Stream$ for M. Then every queue has exactly two levels. Adding a new list suspension to the end of the stream with $+\!\!+$ takes $O(|M|)$ time, but, since $+\!\!+$ is incremental, this cost can be amortized over the operations on the top-level queue. Since these queues are not recursive, we have no need for polymorphic recursion. This variation is explored in greater detail in [Oka96a].

Hint to Practitioners: In practice, variations on these queues are the fastest known implementations for applications that use persistence sparingly, but that require good behavior even in pathological cases.

7.2 Structural Abstraction

The second kind of data-structural bootstrapping is *structural abstraction*, which is typically used to extend an implementation of collections, such as lists or heaps, with an efficient $join$ function for combining two collections. For many implementations, designing an efficient $insert$ function, which adds a single element to a collection, is easy, but designing an efficient $join$ function is difficult. Structural abstraction creates collections that contain other collections as elements. Then two collections can be joined by simply inserting one collection into the other.

The ideas of structural abstraction can largely be described at the level of types. Suppose $\alpha\ C$ is a collection datatype with elements of type α, and that this datatype supports an efficient $insert$ function, with signature

val insert : $\alpha \times \alpha$ C $\rightarrow \alpha$ C

Call $\alpha\ C$ the *primitive* type. From this type, we wish to derive a new datatype $\alpha\ B$, called the *bootstrapped* type, such that $\alpha\ B$ supports both $insert$ and $join$ efficiently, with signatures

val insert$_B$: $\alpha \times \alpha$ B $\rightarrow \alpha$ B
val join$_B$: α B $\times\ \alpha$ B $\rightarrow \alpha$ B

(We use the B subscript to distinguish functions on the bootstrapped type from functions on the primitive type.) In addition, $\alpha\ B$ should support an efficient $unit$ function for creating a new singleton collection.

val unit$_B$: $\alpha \rightarrow \alpha$ B

Then, $insert_B$ can be implemented simply as

fun insert$_B$ (x, b) = join$_B$ (unit$_B$ x, b)

The basic idea of structural abstraction is to somehow represent bootstrapped collections as primitive collections of other bootstrapped collections. Then $join_B$ can be implemented in terms of $insert$ (not $insert_B$!) roughly as

fun join$_B$ (b_1, b_2) = insert (b_1, b_2)

This inserts b_1 as an element of b_2. Alternatively, one could insert b_2 as an element of b_1, but the point is that join has been reduced to simple insertion.

Of course, the situation is not quite that simple. Based on the above description, we might attempt to define $\alpha\ B$ as

```
datatype α B = B of (α B) C
```

This definition can be viewed as specifying an isomorphism

$$\alpha\ B \cong (\alpha\ B)\ C$$

By unrolling this isomorphism a few times, we can quickly spot the flaw in this definition.

$$\alpha\ B \cong (\alpha\ B)\ C \cong ((\alpha\ B)\ C)\ C \cong \cdots \cong ((\cdots C)\ C)\ C$$

The primitive elements of type α have disappeared! We can solve this by making each bootstrapped collection a pair of a single element with a primitive collection.

```
datatype α B = B of α × (α B) C
```

Then, for instance, $unit_B$ can be defined as

```
fun unit_B x = B (x, empty)
```

where $empty$ is the empty primitive collection.

But now we have another problem. If every bootstrapped collection contains at least a single element, how do we represent the empty bootstrapped collection? We therefore refine the type one more time.

```
datatype α B = Empty | B of α × (α B) C
```

Remark: Actually, we will always arrange that the primitive collection C contains only non-empty bootstrapped collections. This situation can be described more precisely by the types

```
datatype α B⁺ = B⁺ of α × (α B⁺) C
datatype α B = Empty | NonEmpty of B⁺
```

Unfortunately, definitions of this form lead to more verbose code. Hence, for presentation purposes, we will use the earlier less precise, but more concise, definition. ◇

Now, we can refine the above templates for $insert_B$ and $join_B$ as

```
fun insert_B ($x$, Empty) = B ($x$, empty)
  | insert_B ($x$, B ($y$, $c$)) = B ($x$, insert ($unit_B\ y$, $c$))
fun join_B ($b$, Empty) = $b$
  | join_B (Empty, $b$) = $b$
  | join_B (B ($x$, $c$), $b$) = B ($x$, insert ($b$, $c$))
```

These templates can easily be varied in several ways. For instance, in the second clause of $insert_B$, we could reverse the roles of x and y. Similarly, in the third clause of $join_B$, we could reverse the roles of the first argument and the second argument.

For any given collection, there is typically some distinguished element that can be inspected or deleted, such as the first element or the smallest element. The $insert_B$ and $join_B$ templates should be instantiated in such a way that the distinguished element in the bootstrapped collection $B\ (x,\ c)$ is x itself. The creative part of designing a bootstrapped data structure using structural abstraction is implementing the $delete_B$ routine that discards the distinguished element x. After discarding x, we are left with a collection of type $(\alpha\ B)\ C$, which must then be converted into a bootstrapped collection of type $\alpha\ B$. The details of how this is accomplished vary from data structure to data structure.

We next instantiate these templates in two ways. First, we bootstrap queues to support catenation (i.e., append) efficiently. Second, we bootstrap heaps to support merge efficiently.

7.2.1 Lists With Efficient Catenation

The first data structure we will implement using structural abstraction is catenable lists, as specified by the signature in Figure 7.2. Catenable lists extend the usual list signature with an efficient append function (++). As a convenience, catenable lists also support $snoc$, even though we could easily simulate $snoc\ (xs,\ x)$ by $xs \mathbin{+\!\!+} cons\ (x,\ empty)$. Because of this ability to add elements to the rear of a list, a more accurate name for this data structure would be catenable output-restricted deques.

We obtain an efficient implementation of catenable lists that supports all operations in $O(1)$ amortized time by bootstrapping an efficient implementation of FIFO queues. The exact choice of implementation for the primitive queues is largely irrelevant; any of the persistent, constant-time queue implementations will do, whether amortized or worst-case.

Given an implementation Q of primitive queues matching the QUEUE signature, structural abstraction suggests that we can represent catenable lists as

```
datatype $\alpha$ Cat = Empty | Cat of $\alpha \times \alpha$ Cat Q.Queue
```

One way to interpret this type is as a tree where each node contains an element, and the children of each node are stored in a queue from left to right. Since we wish for the first element of the

```
signature CATENABLELIST =
sig
  type α Cat

  exception EMPTY

  val empty   : α Cat
  val isEmpty : α Cat → bool

  val cons    : α × α Cat → α Cat
  val snoc    : α Cat × α → α Cat
  val ++      : α Cat × α Cat → α Cat

  val head    : α Cat → α       (* raises EMPTY if list is empty *)
  val tail    : α Cat → α Cat   (* raises EMPTY if list is empty *)
end
```

Figure 7.2: Signature for catenable lists.

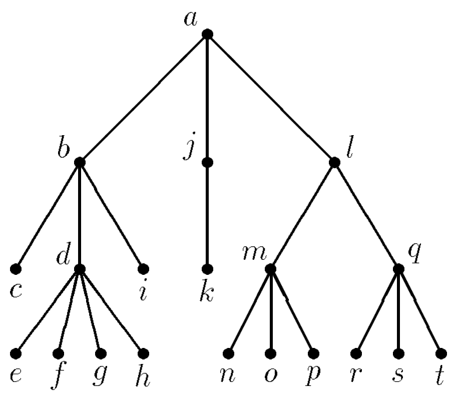

Figure 7.3: A tree representing the list $a \dots t$.

list to be easily accessible, we will store it at the root of the tree. This suggests ordering the elements in a preorder, left-to-right traversal of the tree. A sample list containing the elements $a \dots t$ is shown in Figure 7.3.

Now, $head$ is simply

```
fun head (Cat (x, _)) = x
```

To catenate two non-empty lists, we $link$ the two trees by making the second tree the last child of the first tree.

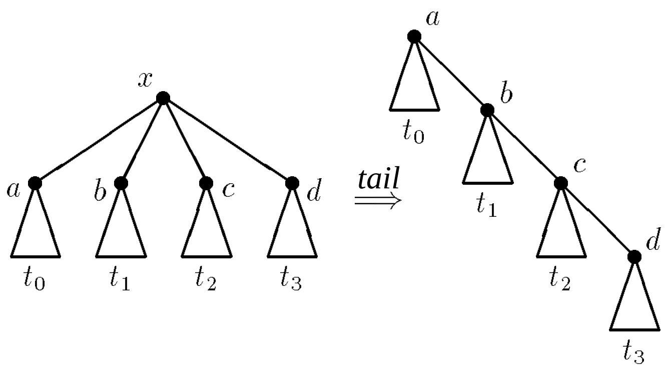

Figure 7.4: Illustration of the *tail* operation.

```
fun xs ++ Empty = xs
  | Empty ++ xs = xs
  | xs ++ ys = link (xs, ys)
```

where $link$ adds its second argument to the queue of its first argument.

```
fun link (Cat (x, q), s) = Cat (x, Q.snoc (q, s))
```

$cons$ and $snoc$ simply call ++.

```
fun cons (x, xs) = Cat (x, Q.empty) ++ xs
fun snoc (xs, x) = xs ++ Cat (x, Q.empty)
```

Finally, given a non-empty tree, $tail$ should discard the root and somehow combine the queue of children into a single tree. If the queue is empty, then $tail$ should return $Empty$. Otherwise we link all the children together.

```
fun tail (Cat (x, q)) = if Q.isEmpty q then Empty else linkAll q
```

Since catenation is associative, we can link the children in any order we desire. However, a little thought reveals that linking the children from right to left, as illustrated in Figure 7.4, will result in the least duplicated work on subsequent calls to $tail$. Therefore, we implement $linkAll$ as

```
fun linkAll q = let val t = Q.head q
                    val q' = Q.tail q
                in if Q.isEmpty q' then t else link (t, linkAll q') end
```

Remark: $linkAll$ is an instance of the $foldr1$ program schema. ◇

In this implementation, $tail$ may take as much as $O(n)$ time, but it is not difficult to show that the amortized cost of $tail$ is only $O(1)$, provided lists are used ephemerally. Unfortunately, this implementation is not efficient when used persistently.

To achieve good amortized bounds even in the face of persistence, we must somehow incorporate lazy evaluation into this implementation. Since $linkAll$ is the only routine that takes more than $O(1)$ time, it is the obvious candidate. We rewrite $linkAll$ to suspend every recursive call. This suspension is forced when a tree is removed from a queue.

```
fun linkAll q = let val $t = Q.head q
                    val q' = Q.tail q
                in if Q.isEmpty q' then t else link (t, $linkAll q') end
```

For this definition to make sense, every queue must contain tree suspensions rather than trees, so we redefine the datatype as

```
datatype α Cat = Empty | Cat of α × α Cat susp Q.Queue
```

To conform to this new datatype, ++ must spuriously suspend its second argument.

```
fun xs ++ Empty = xs
  | Empty ++ xs = xs
  | xs ++ ys = link (xs, $ys)
```

The complete implementation is shown in Figure 7.5.

$head$ clearly runs in $O(1)$ worst-case time, while $cons$ and $snoc$ have the same time requirements as ++. We now prove that ++ and $tail$ run in $O(1)$ amortized time using the banker's method. The unshared cost of each is $O(1)$, so we must merely show that each discharges only $O(1)$ debits.

Let $d_t(i)$ be the number of debits on the ith node of tree t and let $D_t(i) = \sum_{j=0}^{i} d_t(j)$ be the cumulative number of debits on all nodes of t up to and including node i. Finally, let D_t be the total number debits on all nodes in t (i.e., $D_t = D_t(|t| - 1)$). We maintain two invariants on debits.

First, we require that the number of debits on any node be bounded by the degree of the node (i.e., $d_t(i) \leq degree_t(i)$). Since the sum of degrees of all nodes in a non-empty tree is one less than the size of the tree, this implies that the total number of debits in a tree is bounded by the size of the tree (i.e., $D_t < |t|$). We will maintain this invariant by incrementing the number of debits on a node only when we also increment its degree.

Second, we insist that the $D_t(i)$ be bounded by some linear function on i. The particular linear function we choose is

$$D_t(i) \leq i + depth_t(i)$$

```
functor CatenableList (structure Q : QUEUE) : CATENABLELIST =
struct
  datatype α Cat = Empty | Cat of α × α Cat susp Q.Queue

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true
    | isEmpty (Cat q) = false

  fun link (Cat (x, q), s) = Cat (x, Q.snoc (q, s))
  fun linkAll q = let val $t = Q.head q
                      val q' = Q.tail q
                  in if Q.isEmpty q' then t else link (t, $linkAll q') end

  fun xs ++ Empty = xs
    | Empty ++ xs = xs
    | xs ++ ys = link (xs, $ys)
  fun cons (x, xs) = Cat (x, Q.empty) ++ xs
  fun snoc (xs, x) = xs ++ Cat (x, Q.empty)

  fun head Empty = raise EMPTY
    | head (Cat (x, _)) = x
  fun tail Empty = raise EMPTY
    | tail (Cat (x, q)) = if Q.isEmpty q then Empty else linkAll q
end
```

Figure 7.5: Catenable lists.

where $depth_t(i)$ is the length of the path in t from the root to node i. This invariant is called the ***left-linear debit invariant***. Notice that the left-linear debit invariant guarantees that $d_t(0) = D_t(0) \leq 0 + 0 = 0$, so all debits on a node have been discharged by the time it reaches the root. (In fact, the root is not even suspended!) The only time we actually force a suspension is when the suspended node is about become the new root.

Theorem 7.1 *$+\!\!+$ and $tail$ maintain both debit invariants by discharging one and three debits, respectively.*

Proof: ($+\!\!+$) The only debit created by $+\!\!+$ is for the trivial suspension of its second argument. Since we are not increasing the degree of this node, we immediately discharge the new debit. Now, assume that t_1 and t_2 are non-empty and let $t = t_1 +\!\!+ t_2$. Let $n = |t_1|$. Note that the index, depth, and cumulative debits of each node in t_1 are unaffected by the catenation, so for $i < n$

$$\begin{array}{rcl} D_t(i) & = & D_{t_1}(i) \\ & \leq & i + \mathit{depth}_{t_1}(i) \\ & = & i + \mathit{depth}_t(i) \end{array}$$

The nodes in t_2 increase in index by n, increase in depth by one, and accumulate the total debits of t_1, so

$$\begin{array}{rcl} D_t(n+i) & = & D_{t_1} + D_{t_2}(i) \\ & < & n + D_{t_2}(i) \\ & \leq & n + i + \mathit{depth}_{t_2}(i) \\ & = & n + i + \mathit{depth}_t(n+i) - 1 \\ & < & (n+i) + \mathit{depth}_t(n+i) \end{array}$$

Thus, we do not need to discharge any further debits to maintain the left-linear debit invariant.

(tail) Let $t' = \mathit{tail}\ t$. After discarding the root of t, we link the children $t_0 \ldots t_{m-1}$ from right to left. Let t'_j be the partial result of linking $t_j \ldots t_{m-1}$. Then $t' = t'_0$. Since every link except the outermost is suspended, we assign a single debit to the root of each t_j, $0 < j < m-1$. Note that the degree of each of these nodes increases by one. We also assign a debit to the root of t'_{m-1} because the last call to $\mathit{linkAll}$ is suspended even though it does not call link. Since the degree of this node does not change, we immediately discharge this final debit.

Now, suppose the ith node of t appears in t_j. By the left-linear debit invariant, we know that $D_t(i) < i + \mathit{depth}_t(i)$, but consider how each of these quantities changes with the tail. i decreases by one because the first element is discarded. The depth of each node in t_j increases by $j-1$ (see Figure 7.4) while the cumulative debits of each node in t_j increase by j. Thus,

$$\begin{array}{rcl} D_{t'}(i-1) & = & D_t(i) + j \\ & \leq & i + \mathit{depth}_t(i) + j \\ & = & i + (\mathit{depth}_{t'}(i-1) - (j-1)) + j \\ & = & (i-1) + \mathit{depth}_{t'}(i-1) + 2 \end{array}$$

Discharging the first two debits restores the invariant, for a total of three debits. □

Hint to Practitioners: Given a good implementation of queues, this is the fastest known implementation of persistent catenable lists, especially for applications that use persistence heavily.

7.2.2 Heaps With Efficient Merging

Next, we apply structural abstraction to heaps to obtain an efficient merge operation. This section reflects joint work with Gerth Brodal.

Assume that we have an implementation of heaps that supports $insert$ in $O(1)$ worst-case time and $merge$, $findMin$, and $deleteMin$ in $O(\log n)$ worst-case time. The skew binomial heaps of Section 6.4.2 are one such implementation. Using structural abstraction, we improve the running time of both $findMin$ and $merge$ to $O(1)$ worst-case time.

For now, assume that the type of heaps is polymorphic in the type of elements, and that, for any type of elements, we magically know the right comparison function to use. Later we will account for the fact that both the type of elements and the comparison function on those elements are fixed at functor-application time.

Under the above assumption, the type of bootstrapped heaps can be given as

```
datatype α Heap = Empty | Heap of α × (α Heap) H.Heap
```

where H is the implementation of primitive heaps. The element stored at any given $Heap$ node will be the minimum element in the subtree rooted at that node. The elements of the primitive heaps are themselves bootstrapped heaps. Within the primitive heaps, bootstrapped heaps are ordered with respect to their minimum elements (i.e., their roots).

Since the minimum element is stored at the root, $findMin$ is simply

```
fun findMin (Heap (x, _)) = x
```

To $merge$ two bootstrapped heaps, we insert the heap with the larger root into the heap with the smaller root.

```
fun merge (Empty, h) = h
  | merge (h, Empty) = h
  | merge (h₁ as Heap (x, p₁), h₂ as Heap (y, p₂)) =
      if x < y then Heap (x, H.insert (h₂, p₁)) else Heap (y, H.insert (h₁, p₂))
```

(In the comparison $x < y$, we assume that $<$ is the right comparison function for these elements.) Now, $insert$ is defined in terms of $merge$.

```
fun insert (x, h) = merge (Heap (x, H.empty), h)
```

Finally, we consider $deleteMin$, defined as

```
fun deleteMin (Heap (x, p)) =
      if H.isEmpty p then Empty
      else let val (Heap (y, p₁)) = H.findMin p
               val p₂ = H.deleteMin p
           in Heap (y, H.merge (p₁, p₂)) end
```

After discarding the root, we first check if the primitive heap p is empty. If it is, then the new heap is empty. Otherwise, we find and remove the minimum element in p, which is the bootstrapped heap with the overall minimum element; this element becomes the new root. Finally, we merge p_1 and p_2 to obtain the new primitive heap.

The analysis of these heaps is simple. Clearly, $\mathit{findMin}$ runs in $O(1)$ worst-case time regardless of the underlying implementation of primitive heaps. insert and merge depend only on $H.\mathit{insert}$. Since we have assumed that $H.\mathit{insert}$ runs in $O(1)$ worst-case time, so do insert and merge. Finally, $\mathit{deleteMin}$ calls $H.\mathit{findMin}$, $H.\mathit{deleteMin}$, and $H.\mathit{merge}$. Since each of these runs in $O(\log n)$ worst-case time, so does $\mathit{deleteMin}$.

Until now, we have assumed that heaps are polymorphic, but in fact the HEAP signature specifies that heaps are monomorphic — both the type of elements and the comparison function on those elements are fixed at functor-application time. The implementation of a heap is a functor that is parameterized by the element type and the comparison function. Therefore, the functor that we use to bootstrap heaps maps heap functors to heap functors, rather than heap structures to heap structures. Using higher-order functors [MT94], this can be expressed as

```
functor Bootstrap (functor MakeH (structure E : ORDERED) : sig
                                                              include HEAP
                                                              sharing Elem = E
                                                           end)
                  (structure E : ORDERED) : HEAP = ...
```

The $\mathit{Bootstrap}$ functor takes the MakeH functor as an argument. The MakeH functor takes the ORDERED structure E, which contains the element type and the comparison function, and returns a HEAP structure. Given MakeH, $\mathit{Bootstrap}$ returns a functor that takes an ORDERED structure E and returns a HEAP structure.

Remark: The sharing constraint in the signature for the MakeH functor is necessary to ensure that the functor returns a heap structure with the desired element type. This kind of sharing constraint is extremely common with higher-order functors. ◇

Now, to create a structure of primitive heaps with bootstrapped heaps as elements, we apply MakeH to the ORDERED structure $\mathit{BootstrappedH}$ that defines the type of bootstrapped heaps and a comparison function that orders two bootstrapped heaps by their minimum elements. (The ordering relation is undefined on empty bootstrapped heaps.) This is expressed by the following mutually recursive structure declarations.

```
structure rec BootstrappedH =
   struct
      datatype T = Empty | Heap of Elem.T × H.Heap
      fun leq (Heap (x, _), Heap (y, _)) = Elem.leq (x, y)
   end
and H = MakeH (structure E = BootstrappedH)
```

where $Elem$ is the ORDERED structure specifying the true elements of the bootstrapped heap. The complete implementation of the $Bootstrap$ functor is shown in Figure 7.6.

Remark: Standard ML does not support recursive structure declarations, and for good reason — this declaration does not make sense for $MakeH$ functors that have effects. However, the $MakeH$ functors to which we might consider applying $Bootstrap$, such as $SkewBinomialHeap$ from Section 6.4.2, are well-behaved in this respect, and the recursive pattern embodied by the $Bootstrap$ functor does make sense for these functors. It is unfortunate that Standard ML does not allow us to express bootstrapping in this fashion.

We can still implement bootstrapped heaps in Standard ML by inlining a particular choice for $MakeH$, such as $SkewBinomialHeap$, and then eliminating $BootstrappedH$ and H as separate structures. The recursion on structures then reduces to recursion on datatypes, which is supported by Standard ML. $\diamond$

7.3 Related Work

Data-Structural Bootstrapping Buchsbaum *et al.* identified data-structural bootstrapping as a general data structure design technique in [Buc93, BT95, BST95]. Structural decomposition and structural abstraction had previously been used in [Die82] and [DST94], respectively.

Catenable Lists Although it is relatively easy to design alternative representations of persistent lists that support efficient catenation (see, for example, [Hug86]), such alternative representations seem almost inevitably to sacrifice efficiency on the $head$ and/or $tail$ functions.

Myers [Mye84] described a representation based on AVL trees that supports all relevant list functions in $O(\log n)$ time.

Driscoll, Sleator, and Tarjan achieved the first sub-logarithmic implementation in [DST94]. They represent catenable lists as n-ary trees with the elements at the leaves. To keep the leftmost leaves near the root, they use a restructuring operation known as $pull$ that removes the first grandchild of the root and reattaches it directly to the root. Unfortunately, catenation breaks all useful invariants based on this restructuring heuristic, so they are forced to develop quite a bit of machinery to support catenation. The resulting implementation supports catenation in $O(\log \log k)$ worst-case time, where k is the number of list operations (note that k may be much smaller than n), and all other functions in $O(1)$ worst-case time.

Buchsbaum and Tarjan [BT95] use structural decomposition to recursively decompose catenable deques of size n into catenable deques of size $O(\log n)$. They use the $pull$ operation of Driscoll, Sleator, and Tarjan to keep their tree balanced (i.e., of depth $O(\log n)$), and then use the smaller deques to represent the left and right spines of each subtree. This yields an

```
functor Bootstrap (functor MakeH (structure E : ORDERED) : sig
                                                              include HEAP
                                                              sharing Elem = E
                                                           end)
                  (structure E : ORDERED) : HEAP =
struct
  structure Elem = E

  (* recursive structures not supported in SML! *)
  structure rec BootstrappedH =
    struct
      datatype T = Empty | Heap of Elem.T × H.Heap
      fun leq (Heap ($x$, _), Heap ($y$, _)) = Elem.leq ($x$, $y$)
    end
  and H = MakeH (structure E = BootstrappedH)

  open BootstrappedH   (* expose Empty and Heap constructors *)

  type Heap = BootstrappedH.T

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true
    | isEmpty (Heap _) = false

  fun merge (Empty, $h$) = $h$
    | merge ($h$, Empty) = $h$
    | merge ($h_1$ as Heap ($x$, $p_1$), $h_2$ as Heap ($y$, $p_2$)) =
        if Elem.leq ($x$, $y$) then Heap ($x$, H.insert ($h_2$, $p_1$)) else Heap ($y$, H.insert ($h_1$, $p_2$))
  fun insert ($x$, $h$) = merge (Heap ($x$, H.empty), $h$)

  fun findMin Empty = raise EMPTY
    | findMin (Heap ($x$, _)) = $x$
  fun deleteMin Empty = raise EMPTY
    | deleteMin (Heap ($x$, $p$)) =
        if H.isEmpty $p$ then Empty
        else let val (Heap ($y$, $p_1$)) = H.findMin $p$
                 val $p_2$ = H.deleteMin $p$
             in Heap ($y$, H.merge ($p_1$, $p_2$)) end
end
```

Figure 7.6: Bootstrapped heaps.

implementation that supports deletion of the first or last element in $O(\log^* k)$ worst-case time, and all other deque functions, including catenation, in $O(1)$ worst-case time.

Kaplan and Tarjan [KT95] finally achieved an implementation that supports catenation and all other usual list functions in $O(1)$ worst-case time. Their data structure is based on the technique of recursive slowdown. We will describe recursive slowdown in greater detail in Chapter 8.

The implementation of catenable lists in Section 7.2.1 first appeared in [Oka95a]. It is much simpler than Kaplan and Tarjan's, but yields amortized bounds rather than worst-case bounds.

Mergeable Heaps Many imperative implementations support $insert$, $merge$, and $findMin$ in $O(1)$ amortized time, and $deleteMin$ in $O(\log n)$ amortized time, including binomial queues [KL93], Fibonacci heaps [FT87], relaxed heaps [DGST88], V-heaps [Pet87], bottom-up skew heaps [ST86b], and pairing heaps [FSST86]. However, of these, only pairing heaps appear to retain their amortized efficiency when combined with lazy evaluation in a persistent setting [Oka96a], and, unfortunately, the bounds for pairing heaps have only been conjectured, not proved.

Brodal [Bro95, Bro96] achieves equivalent worst-case bounds. His original data structure [Bro95] can be implemented purely functionally (and thus made persistent) by combining the recursive-slowdown technique of Kaplan and Tarjan [KT95] with a purely functional implementation of real-time deques, such as the real-time deques of Section 5.4.3. However, such an implementation would be both complicated and slow. Brodal and Okasaki simplify this implementation in [BO96], using skew binomial heaps (Section 6.4.2) and structural abstraction (Section 7.2.2).

Polymorphic Recursion Several attempts have been made to extend Standard ML with polymorphic recursion, such as [Myc84, Hen93, KTU93]. One complication is that type inference is undecidable in the presence of polymorphic recursion [Hen93, KTU93], even though it is tractable in practice. Haskell 1.3 [P$^+$96] sidesteps this problem by allowing polymorphic recursion whenever the programmer provides an explicit type signature.

Chapter 8

Implicit Recursive Slowdown

Implicit recursive slowdown is a lazy variant of the recursive-slowdown technique of Kaplan and Tarjan [KT95]. We first review recursive slowdown, and then show how lazy evaluation can significantly simplify this technique. Finally, we illustrate implicit recursive slowdown with implementations of queues and catenable deques.

8.1 Recursive Slowdown

The simplest illustration of recursive slowdown is a variant of binary numbers that can be incremented in $O(1)$ worst-case time. (We have already seen several such variants, including skew binary numbers and segmented binary numbers.) As always, the trick is to avoid cascades of carries. In recursive slowdown, we allow digits to be `0`, `1`, or `2`. `2`s exist only temporarily and represent a carry in progress. To increment a number, we first increment the first digit, which is guaranteed not to be `2`. We then find the first non-`1` digit. If it is `0`, we do nothing, but if it is `2`, we convert it to `0` and increment the following digit, which is also guaranteed not to be `2`. Changing a `2` to a `0` and incrementing the following digit corresponds to executing a single carry step.

It is easy to show that following the above rules maintains the invariant that the first `2` is preceded by at least one `0` (and any number of `1`s) and that any pair of `2`s is separated by at least one `0` (and any number of `1`s). This invariant guarantees that we never attempt to increment a digit that is already `2`.

Since we want the increment function to run in $O(1)$ worst-case time, we cannot afford to scan the digits to find the first non-`1` digit. Instead, we choose a representation that groups contiguous blocks of `1`s together.

```
datatype Digit = Zero | Ones of int | Two
type Nat = Digit list
```

The integer associated with $Ones$ is the size of the block. Now the first non-`1` digit is either the first element of the $Digit\ list$ or the second element if the first element is a $Ones$ block.

To increment a number, we first blindly increment the first digit, which is either `0` or `1`. If it is `0`, it becomes `1` (and possibly joins an existing block of `1`s). If it is `1`, it becomes `2` (possibly eliminating an existing block of `1`s). This is achieved by the following function:

```
**fun** simpleInc [ ] = [Ones 1]
  | simpleInc (Zero :: $ds$) = ones (1, $ds$)
  | simpleInc (Ones $k$ :: $ds$) = Two :: ones ($k-1$, $ds$)
```

where the $ones$ pseudo-constructor discards empty blocks and combines adjacent blocks of `1`s.

```
**fun** ones (0, $ds$) = $ds$
  | ones ($k_1$, Ones $k_2$ :: $ds$) = Ones ($k_1$+$k_2$) :: $ds$
  | ones ($k$, $ds$) = Ones $k$ :: $ds$
```

The $fixup$ function finds the first non-`1` digit, and if it is `2`, converts it to `0` and blindly increments the following digit.

```
**fun** fixup (Two :: $ds$) = Zero :: simpleInc $ds$
  | fixup (Ones $k$ :: Two :: $ds$) = Ones $k$ :: Zero :: simpleInc $ds$
  | fixup $ds$ = $ds$
```

Finally, inc calls $simpleInc$, followed by $fixup$.

```
**fun** inc $ds$ = fixup (simpleInc $ds$)
```

Remark: Actually, in a functional language, inc would typically be implemented using function composition, as in

```
**val** inc = fixup ∘ simpleInc
```

$\circ$ is a higher-order operator that takes two functions and returns a function such that $(f \circ g)\ x = f\ (g\ x)$. ◇

This implementation can serve as a template for many other data structures. Such a data structure comprises a sequence of levels, where each level can be classified as *green*, *yellow*, or *red*. Each color corresponds to a digit in the above implementation, with *green*=`0`, *yellow*=`1`, and *red*=`2`. We maintain the invariants that the first red level is preceded by at least one green level, and that any two red levels are separated by at least one green level. An operation on any given object may degrade the first level from green to yellow, or from yellow to red, but never from green to red. A $fixup$ procedure then checks if the first non-yellow level is red, and if so converts it to green, possibly degrading the following level from green to yellow, or from yellow to red. Consecutive yellow levels are grouped in a block to support efficient access to the first non-yellow level. Kaplan and Tarjan [KT95] describe two implementations based on this template: real-time deques and real-time catenable lists.

8.2 Implicit Recursive Slowdown

The essence of the recursive-slowdown implementation of binary numbers is a method for executing carries incrementally. By now we have seen many examples of incremental functions implemented with lazy evaluation. By combining the ideas of recursive slowdown with lazy evaluation, we obtain a new technique, called *implicit recursive slowdown*, that is significantly simpler than the original.

Consider the following, straightforward implementation of binary numbers as streams of `0`s and `1`s:

```
datatype Digit = Zero | One
type Nat = Digit Stream

fun inc ($Nil) = $Cons (One, $Nil)
  | inc ($Cons (Zero, ds)) = $Cons (One, ds)
  | inc ($Cons (One, ds)) = $Cons (Zero, inc ds)
```

This is exactly the same as the original presentation of binary numbers in Chapter 6, except with streams instead of lists.

Remark: This implementation is less lazy than it could be. It forces its argument immediately, and then creates a suspension of the result. A reasonable alternative would be to also suspend forcing the argument, as in

```
fun inc' ds = $case force ds of
                 Nil ⇒ Cons (One, $Nil)
               | Cons (Zero, ds') ⇒ Cons (One, ds')
               | Cons (One, ds') ⇒ Cons (Zero, inc' ds')
```

However, in this chapter, we will often need to force one level ahead of the current level, so we stick with the first implementation. ◇

Theorem 8.1 *inc runs in $O(1)$ amortized time.*

Proof: We use the banker's method. By inspection, the unshared cost of inc is $O(1)$. Therefore, to show that inc runs in $O(1)$ amortized time, we must merely show that inc discharges only $O(1)$ debits per operation. In fact, we show that inc discharges only two debits.

Each suspension except the outermost is the tail of some digit. We allow the tail of a `0` to retain a single debit, but require that the tail of a `1` be fully paid off. In addition, the outermost suspension may not have any debits.

We argue by *debit passing*. Whenever a suspension has more debits than it is allowed, we pass those debits to the enclosing suspension, which is the tail of the previous digit. We discharge debits whenever they reach the outermost suspension. Debit passing is safe because earlier tails must be forced before later tails can be forced. Passing responsibility for discharging debits from a later tail to an earlier tail ensures that those debits will be discharged before the earlier tail is forced, and hence before the later tail can be forced. We show by induction on the depth of recursion that, after any cascade of incs, the outermost suspension always has two debits that must be discharged.

First, consider a call to inc that changes a `0` to a `1` (i.e., the final call in a cascade). We begin by creating a debit to cover the cost of the new suspension. In addition, the new suspension receives a debit from the current digit's tail, since that tail's debit allowance has dropped from one to zero. Altogether, the new suspension has been charged two debits.

Next, consider a call to inc that changes a `1` to a `0` and recurses. Again, we begin by creating a debit to cover the cost of the new suspension. When forced, this suspension will force the current digit's tail, but that is okay since the tail of a `1` has no debits. Finally, the new suspension receives a single debit from the recursive call to inc, since that suspension (which is the tail of a `0`) is allowed one debit, but, by the inductive hypothesis, has been charged two debits. Again, the new suspension has been charged a total of two debits. □

As with recursive slowdown, this very simple implementation can serve as a template for many other data structures. Such a data structure consists of a lazy sequence of levels (digits), where each level can be classified as *green* (`0`) or *yellow* (`1`). An operation on an object begins at the outer level and only occasionally propagates to the next level. In particular, an operation on a green level never propagates to the next level but may degrade the level from green to yellow. Operations on yellow levels may (lazily) propagate to the next level, but only after upgrading the current level to green. For example, with binary numbers, incrementing a `0` produces a `1` and stops. Incrementing a `1` recurses to the next level, but produces a `0` at the current level.

The intuition behind this framework is that successive operations at a given level cannot both propagate to the next level; there is a delay of at least one operation when the level is changed from green to yellow. Hence, every other operation may affect the second level, but only every fourth operation may affect the third level, and so on. Intuitively, then, the amortized cost of a single operation is approximately $O(1 + 1/2 + 1/4 + 1/8 + \cdots) = O(1)$. Unfortunately, this clean intuitive picture is complicated by persistence. However, the above proof can be generalized to apply to any problem in this framework.

Clearly, implicit recursive slowdown is much simpler than recursive slowdown. We have eliminated the headache of grouping yellow levels into blocks, and have also eliminated explicit representations of red levels. In a sense, red levels are still present, but they are represented *implicitly* as suspended computations that have not yet been executed. However, recursive slowdown has the advantage that it naturally yields data structures with worst-case bounds, whereas implicit recursive slowdown naturally yields data structures with amortized bounds. If desired, we can often regain worst-case bounds using the scheduling techniques of Chapter 4. We illustrate the use of scheduling on binary numbers.

We extend the type of binary numbers with a schedule of type $Digit\ Stream\ list$. The elements of this list will be suspended calls to $lazyInc$, where $lazyInc$ is just the inc function defined above.

```
fun lazyInc ($Nil) = $Cons (One, $Nil)
  | lazyInc ($Cons (Zero, $ds$)) = $Cons (One, $ds$)
  | lazyInc ($Cons (One, $ds$)) = $Cons (Zero, lazyInc $ds$)
```

The initial schedule is empty.

```
type Nat = Digit Stream $\times$ Digit Stream list
val zero = ($Nil, [ ])
```

To execute a suspension, we simply inspect the first digit of a stream. If it is `0`, then there is another recursive call to $lazyInc$, so we put the remaining stream back in the schedule. If it is `1`, then this call to $lazyInc$ terminates, so we discard the rest of the stream.

```
fun exec [ ] = [ ]
  | exec (($Cons (One, _)) :: $sched$) = $sched$
  | exec (($Cons (Zero, $ds$)) :: $sched$) = $ds$ :: $sched$
```

Altogether, inc calls $lazyInc$, places the resulting stream on the schedule, and then executes two suspensions from the schedule.

```
fun inc ($ds$, $sched$) =
      let val $ds'$ = lazyInc $ds$
      in ($ds'$, exec (exec ($ds'$ :: $sched$))) end
```

To show that inc runs in $O(1)$ worst-case time, we prove that, whenever $exec$ executes a suspension of the form $lazyInc\ ds$, ds has already been forced and memoized. Define the *range* of a call to $lazyInc$ to be the set of indices of all digits altered by that $lazyInc$. Note that digits for any given range form a (possibly empty) sequence of `0`s followed by a `1`. We say two ranges overlap if their intersection is non-empty. At any given moment, all unevaluated suspensions in a digit stream are at indices in the range of some suspension in the schedule. Therefore, we can show that ds has already been executed whenever we execute a suspension of the form $lazyInc\ ds$ by proving that no two suspensions in the schedule have overlapping ranges.

In fact, we prove a slightly stronger result. Define a *completed* `0` to be a `0` whose suspension has already been forced and memoized.

Theorem 8.2 *inc maintains the invariant that every digit stream contains at least two completed `0`s prior to the first range in the schedule, and at least one completed `0` between every two adjacent ranges in the schedule.*

Proof: Consider the situation just before a call to inc. Let r_1 and r_2 be the first two ranges in the schedule. Let z_0 and z_1 be the two completed `0`s before r_1, and let z_2 be the completed `0` between r_1 and r_2. Now, before executing two suspensions from the schedule, inc first adds a new range r_0 to the front of the schedule. Note that r_0 terminates in a `1` that replaces z_0. Let m be the number of `0`s in r_0. There are three cases.

- $m = 0$. The only digit in r_0 is a `1`, so r_0 is eliminated by executing a single suspension. Executing the second suspension forces the first digit of r_1. If this digit is `0`, then it becomes the second completed `0` (along with z_1) before the first range. If this digit is `1`, then r_1 is eliminated and r_2 becomes the new first range. The two completed zeros prior to r_2 are z_1 and z_2.

- $m = 1$. The first two digits of the old digit stream were `1` and `0` (z_0), but they are replaced with `0` and `1`. Executing two suspensions evaluates and memoizes both of these digits, and eliminates r_0. The leading `0` replaces z_0 as one of the two completed `0`s before the first range.

- $m \geq 2$. The first two digits of r_0 are both `0`s. They are both completed by executing the first two suspensions, and become the two completed `0`s before the new first range (the rest of r_0). z_1 becomes the single completed zero between r_0 and r_1.

$\square$

8.3 Supporting a Decrement Function

We have now presented several implementations of an increment function, but of course such a function is useless without some other operations on binary numbers, such as addition and comparisons. These operations typically have an $O(\log n)$ cost, since they must inspect every digit. In the lazy implementation (without scheduling), a digit stream contains at most $O(\log n)$ debits, so discharging those debits does not increase the asymptotic complexity of these operations.

But something interesting happens when we consider the decrement function.

```
fun dec ($Cons (One, $Nil)) = $Nil
  | dec ($Cons (One, ds)) = $Cons (Zero, ds)
  | dec ($Cons (Zero, ds)) = $Cons (One, dec ds)
```

Since this function follows exactly the same pattern as inc, but with the roles of 0 and 1 reversed, we would expect that a similar proof would yield a similar bound. And, in fact, it does provided we do not use *both* increments and decrements. However, if we use both functions, then at least one must be charged $O(\log n)$ amortized time. Simply consider a sequence of increments and decrements that cycle between $2^k - 1$ and 2^k. In that case, every operation touches every digit.

But didn't we prove that both functions run in $O(1)$ amortized time? What went wrong? The problem is that the two proofs require contradictory debit invariants. To prove that inc runs in $O(1)$ amortized time, we require that the tail of a 0 has one debit and the tail of a 1 has zero debits. To prove that dec runs in $O(1)$ amortized time, we require that the tail of a 1 has one debit and the tail of a 0 has zero debits. Put another way, inc needs the green digit to be smaller than the yellow digit while dec needs the green digit to be larger than the yellow digit. We cannot satisfy both requirements simultaneously in this representation.

However, we can achieve $O(1)$ amortized bounds for both operations at the same time by changing the implementation slightly. For increments, we want the largest digit to be yellow, with a smaller green digit. For decrements, we want the smallest digit to be yellow, with a larger green digit. We can satisfy both requirements by allowing digits to be 1, 2, or 3, where 2 is green and 1 and 3 are yellow.

This observation leads immediately to the following implementation:

```
datatype Digit = One | Two | Three
datatype Nat = Digit Stream

fun inc ($Nil) = $Cons (One, $Nil)
  | inc ($Cons (One, ds)) = $Cons (Two, ds)
  | inc ($Cons (Two, ds)) = $Cons (Three, ds)
  | inc ($Cons (Three, ds)) = $Cons (Two, inc ds)

fun dec ($Cons (One, $Nil)) = $Nil
  | dec ($Cons (One, ds)) = $Cons (Two, dec ds)
  | dec ($Cons (Two, ds)) = $Cons (One, ds)
  | dec ($Cons (Three, ds)) = $Cons (Two, ds)
```

Now it is simple to show that both functions run in $O(1)$ amortized time using a proof in which the tail of every green digit has one debit and the tail of every yellow digit has zero debits.

8.4 Queues and Deques

As our first substantial example of implicit recursive slowdown, we present an implementation of queues that also integrates aspects of numerical representations and structural decomposition.

A queue is either *shallow* or *deep*. A shallow queue contains either zero or one elements. A deep queue is decomposed into three segments: a *front*, containing either one or two elements; a *rear*, containing either zero or one elements; and a *middle*, which is a suspended queue of pairs.

```
datatype $\alpha$ ZeroOne = Zero | One of $\alpha$
datatype $\alpha$ OneTwo = One$'$ of $\alpha$ | Two$'$ of $\alpha \times \alpha$
datatype $\alpha$ Queue = Shallow of $\alpha$ ZeroOne
                        | Deep of {F : $\alpha$ OneTwo, M : ($\alpha \times \alpha$) Queue susp, R : $\alpha$ ZeroOne}
```

To add an element to a deep queue using $snoc$, we look at R. If it is `0`, then we add the element to R. If it is `1`, then we pair the new element with the existing element, and add the pair to M, resetting R to `0`. We also need a few special cases for adding an element to a shallow queue.

```
fun snoc (Shallow Zero, $y$) = Shallow (One $y$)
  | snoc (Shallow (One $x$), $y$) = Deep {F = Two$'$ ($x$, $y$), M = $empty, R = Zero}
  | snoc (Deep {F = $f$, M = $m$, R = Zero}, $y$) = Deep {F = $f$, M = $m$, R = One $y$}
  | snoc (Deep {F = $f$, M = $$q$, R = One $x$}, $y$) =
      Deep {F = $f$, M = $snoc ($q$, ($x$, $y$)), R = Zero}
```

Note that in the final clause of $snoc$, we force M earlier than we need to. Instead, we could write this clause as

```
  | snoc (Deep {F = $f$, M = $m$, R = One $x$}, $y$) =
      Deep {F = $f$, M = $snoc (force $m$, ($x$, $y$)), R = Zero}
```

However, this change has no effect on the running time.

To remove an element from a deep queue using $tail$, we look at F. If it is `2`, then we simply remove the element, setting F to `1`. If it is `1`, then we “borrow” a pair from M, and set F to `2`. Again, there are several special cases dealing with shallow queues.

```
fun tail (Shallow (One $x$)) = Shallow Zero
  | tail (Deep {F = Two$'$ ($x$, $y$), M = $m$, R = $r$}) = Deep {F = One$'$ $y$, M = $m$, R = $r$}
  | tail (Deep {F = One$'$ $x$, M = $$q$, R = $r$}) =
      if isEmpty $q$ then Shallow $r$
      else let val ($y$, $z$) = head $q$
           in Deep {F = Two ($y$, $z$), M = $tail $q$, R = $r$} end
```

```
structure ImplicitQueue : QUEUE =     (* assumes polymorphic recursion! *)
struct
  datatype $\alpha$ ZeroOne = Zero | One of $\alpha$
  datatype $\alpha$ OneTwo = One′ of $\alpha$ | Two′ of $\alpha \times \alpha$
  datatype $\alpha$ Queue = Shallow of $\alpha$ ZeroOne
                    | Deep of {F : $\alpha$ OneTwo, M : ($\alpha \times \alpha$) Queue susp, R : $\alpha$ ZeroOne}

  exception EMPTY

  val empty = Shallow Zero
  fun isEmpty (Shallow Zero) = true
    | isEmpty _ = false

  fun snoc (Shallow Zero, $y$) = Shallow (One $y$)
    | snoc (Shallow (One $x$), $y$) = Deep {F = Two′ ($x$, $y$), M = $empty, R = Zero}
    | snoc (Deep {F = $f$, M = $m$, R = Zero}, $y$) = Deep {F = $f$, M = $m$, R = One $y$}
    | snoc (Deep {F = $f$, M = $$q$, R = One $x$}, $y$) = Deep {F = $f$, M = $snoc ($q$, ($x$, $y$)), R = Zero}
  fun head (Shallow Zero) = raise EMPTY
    | head (Shallow (One $x$)) = $x$
    | head (Deep {F = One′ $x$, ...}) = $x$
    | head (Deep {F = Two′ ($x$, $y$), ...}) = $x$
  fun tail (Shallow Zero) = raise EMPTY
    | tail (Shallow (One $x$)) = Shallow Zero
    | tail (Deep {F = Two′ ($x$, $y$), M = $m$, R = $r$}) = Deep {F = One′ $y$, M = $m$, R = $r$}
    | tail (Deep {F = One′ $x$, M = $$q$, R = $r$}) =
        if isEmpty $q$ then Shallow $r$
        else let val ($y$, $z$) = head $q$
             in Deep {F = Two ($y$, $z$), M = $tail $q$, R = $r$} end
end
```

Figure 8.1: Queues based on implicit recursive slowdown.

Note that in the last clause of $tail$, we have choice but to force M since we must test whether M is empty, and if not, query its first pair. However, we can delay the recursive call to $tail$. The complete code appears in Figure 8.1.

Remark: This implementation highlights a third simplification that implicit recursive slowdown offers as compared to ordinary recursive slowdown, along with not explicitly representing red nodes and not grouping yellow nodes into blocks. Whereas this implementation limits F to contain one or two elements and R to contain zero or one elements, an implementation based on ordinary recursive slowdown would allow both F and R to contain from zero to three elements. For F, `0` is red, `1` is yellow, and `2` and `3` are green. For R, `3` is red, `2` is yellow, and `1` and `0` are green. We expect the addition of a red digit, but the extra green digit in each case is surprising. It arises because, under recursive slowdown, when we convert either F or R from red to green by doing a "carry" or "borrow", we must ensure that the other is also green by doing a second "carry" or "borrow", if necessary. So, for instance, when we convert F from red to green, if R is `3` (red), then we move two elements to the middle, changing R to `1`. If R is `2` (yellow), then again we move two elements to the middle, changing R to `0`. Without the second green digit, there would be no way to convert a yellow node to a green node. $\diamond$

To analyze this implementation, we assign debits to every suspension, each of which is the middle field of some deep queue. We adopt a debit invariant that allows each suspension a number of debits governed by the colors of the front and rear fields. F is green if it is `2` and yellow if it is `1`. R is green if it is `0` and yellow if it is `1`. M may have two debits if both F and R are green, one debit if one of F and R is green, and zero debits if both F and R are yellow.

Theorem 8.3 *$snoc$ and $tail$ run in $O(1)$ amortized time.*

Proof: The unshared cost of each function is $O(1)$, so we must merely show that both functions discharge no more than $O(1)$ debits. The analysis of both functions is identical, so we describe only the $tail$ function.

We argue by debit passing. Each cascade of $tail$s ends in a call to $tail$ that changes F from `2` to `1`. (For simplicity of presentation, we ignore the possibility of shallow queues). This decreases the debit allowance of M by one, so we pass the excess debit to the enclosing suspension.

Every intermediate call to $tail$ changes F from `1` to `2` and recurses. There are two subcases:

- R is `0`. M has one debit, which must be discharged before M can be forced. We pass this debit to the enclosing suspension. We create one debit to cover the unshared cost of the suspended recursive call. In addition, this suspension is passed one debit by the recursive call. Since this suspension has a debit allowance of two, we are done.
- R is `1`. M has zero debits, so we can force it for free. We create one debit to cover the unshared cost of the suspended recursive call. In addition, this suspension is passed one debit by the recursive call. Since this suspension has a debit allowance of one, we keep one debit and pass the other to the enclosing suspension.

Every call to $tail$ passes one debit to its enclosing suspension, except the outermost call, which has no enclosing suspension. That call simply discharges its excess debit. $\square$

Remark: In practice, these queues are slower than the implementations in Chapters 3, 4, and 7. However, like many other numerical representations, these queues have the advantage of supporting random access efficiently. In particular, we can $lookup$ or $update$ the ith element in $O(\log i)$ time. As with the numerical representations in Chapter 6, these queues contain a logarithmic number of trees of logarithmic depth. Random access is a two stage process of finding the right tree and then finding the right element.

In the implementation as presented, the presence of trees is somewhat obscured by the use of structural decomposition. However, recall that the first level contains elements, the second level contains pairs of elements, the third level contains pairs of pairs of elements, and so on. These are just complete binary leaf trees. $\diamond$

Finally, we show how to modify this implementation of queues to support double-ended queues. To support deques, we must be able to insert or remove elements from either the front or rear. This is analogous to supporting both increments and decrements for binary numbers. We saw in Section 8.3 that this could be accomplished by allowing digits to range over `1`, `2`, and `3`. Thus, to implement deques, we modify the earlier implementation to allow both the front and rear fields of a deep queue to contain one, two, or three elements. This implementation is shown in Figure 8.2. The analysis is almost identical to that of queues, except that `1`s and `3`s are yellow, and `2`s are green.

We can also easily implement several forms of restricted deques, including

- *Output-restricted deques*, which support insertions on both sides, but removals only from the front. We allow the front field to contain one, two, or three elements, but the rear field to contain only zero or one elements.
- *Input-restricted deques*, which support removals from both sides, but insertions only at the front. We allow the front field to contain one, two, or three elements, but the rear field to contain only one or two elements.

8.5 Catenable Double-Ended Queues

Finally, we use implicit recursive slowdown to implement catenable double-ended queues, with the signature shown in Figure 8.3. We first describe a relatively simple implementation that supports $+\!\!+$ in $O(\log n)$ amortized time and all other operations in $O(1)$ amortized time. We then describe a much more complicated implementation that improves the running time of $+\!\!+$ to $O(1)$.

Consider the following representation for catenable double-ended queues, or *c-deques*. A c-deque is either *shallow* or *deep*. A shallow c-deque is simply an ordinary deque, such as those presented in Chapter 5 or in the previous section. A deep c-deque is decomposed into three segments: a *front*, a *middle*, and a *rear*. The front and rear are both ordinary deques containing two or more elements each. The middle is a c-deque of ordinary deques, each containing two

```
structure ImplicitDeque : DEQUE =    (* assumes polymorphic recursion! *)
struct
  datatype α D = Zero | One of α | Two of of α × α | Three of α × α × α
  datatype α Queue = Shallow of α D
                   | Deep of {F : α D, M : (α × α) Queue susp, R : α D}

  exception EMPTY

  val empty = Shallow Zero
  fun isEmpty (Shallow Zero) = true
    | isEmpty _ = false

  fun dcons (x, Zero) = One x                    fun dsnoc (Zero, x) = One x
    | dcons (x, One a) = Two (x, a)                | dsnoc (One a, x) = Two (a, x)
    | dcons (x, Two (a, b)) = Three (x, a, b)      | dsnoc (Two (a, b), x) = Three (a, b, x)
  fun dhead Zero = raise EMPTY                   fun dlast Zero = raise EMPTY
    | dhead (One a) = a                            | dlast (One a) = a
    | dhead (Two (a, b)) = a                       | dlast (Two (a, b)) = b
    | dhead (Three (a, b, c)) = a                  | dlast (Three (a, b, c)) = c
  fun dtail Zero = raise EMPTY                   fun dinit Zero = raise EMPTY
    | dtail (One a) = Zero                         | dinit (One a) = Zero
    | dtail (Two (a, b)) = One b                   | dinit (Two (a, b)) = One a
    | dtail (Three (a, b, c)) = Two (b, c)         | dinit (Three (a, b, c)) = Two (a, b)

  fun cons (x, Shallow (Three (a, b, c))) = Deep {F = Two (x, a), M = $empty, R = Two (b, c)}
    | cons (x, Shallow d) = Shallow (dcons (x, d))
    | cons (x, Deep {F = Three (a, b, c), M = m, R = r}) =
        Deep {F = Two (x, a), M = $cons ((b, c), force m), R = r}
    | cons (x, Deep {F = f, M = m, R = r}) = Deep {F = dcons (x, f), M = m, R = r}
  fun head (Shallow d) = dhead d
    | head (Deep {F = f, ...}) = dhead f
  fun tail (Shallow d) = Shallow (dtail d)
    | tail (Deep {F = One a, M = $ps, R = r}) =
        if isEmpty ps then Shallow r
        else let val (b, c) = head ps
             in Deep {F = Two (b, c), M = $tail ps, R = r} end
    | tail (Deep {F = f, M = m, R = r}) = Deep {F = dtail f, M = m, R = r}

  ...snoc, last, and init defined symmetrically...
end
```

Figure 8.2: Double-ended queues based on implicit recursive slowdown.

```
signature CATENABLEDEQUE =
sig
  type α Cat

  exception EMPTY

  val empty   : α Cat
  val isEmpty : α Cat → bool

  val cons    : α × α Cat → α Cat
  val head    : α Cat → α          (* raises EMPTY if deque is empty *)
  val tail    : α Cat → α Cat      (* raises EMPTY if deque is empty *)

  val snoc    : α Cat × α → α Cat
  val last    : α Cat → α          (* raises EMPTY if deque is empty *)
  val init    : α Cat → α Cat      (* raises EMPTY if deque is empty *)

  val ++      : α Cat × α Cat → α Cat
end
```

Figure 8.3: Signature for catenable double-ended queues.

or more elements. We assume that D is an implementation of deques satisfying the signature DEQUE.

```
datatype α Cat = Shallow of α D.Queue
               | Deep of {F : α D.Queue, M : α D.Queue Cat susp, R : α D.Queue}
```

Note that this definition assumes polymorphic recursion.

To insert an element at either end, we simply insert the element into either the front deque or the rear deque. For instance, $cons$ is implemented as

```
fun cons (x, Shallow d) = Shallow (D.cons (x, d))
  | cons (x, Deep {F = f, M = m, R = r}) = Deep {F = D.cons (x, f), M = m, R = r}
```

To remove an element from either end, we remove an element from either the front deque or the rear deque. If this drops the length of that deque below two, then we remove the next deque from the middle, add the one remaining element from the old deque, and install the result as the new front or rear. With the addition of the remaining element from the old deque, the new deque now contains at least three elements, so the next operation on that deque will not propagate to the next level. For example, the code for $tail$ is

```
fun tail (Shallow d) = Shallow (D.tail d)
  | tail (Deep {F = f, M = m, R = r}) =
        **if** D.size $f > 2$ **then** Deep $\{$F = D.tail $f$, M = $m$, R = $r\}$
        **else if** isEmpty (force $m$) **then** Shallow $r$
        **else** Deep $\{$F = D.cons (D.last $f$, head (force $m$)), M = $tail (force $m$), R = $r\}$
```

It is simple to see that the proof techniques of this chapter will yield $O(1)$ amortized time bounds on each of these functions.

But what about catenation? To catenate two deep c-deques c_1 and c_2, we retain the front of c_1 as the new front, the rear of c_2 as the new rear, and combine the remaining segments into the new middle by inserting the rear of c_1 into the middle of c_1, and the front of c_2 into the middle of c_2, and then catenating the results.

```
    fun (Deep {F = f₁, M = m₁, R = r₁}) ++ (Deep {F = f₂, M = m₂, R = r₂}) =
        Deep {F = f₁, M = $(snoc (force m₁, r₁) ++ cons (f₂, force m₂)), R = r₂}
```

(Of course, there are also cases where c_1 and/or c_2 are shallow.) Note that $+\!\!+$ recurses to the depth of the shallower c-deque. Furthermore, $+\!\!+$ creates $O(1)$ debits per level, which must be immediately discharged to restore the debit invariant required by the $tail$ function. Therefore, $+\!\!+$ runs in $O(\min(\log n_1, \log n_2))$ amortized time, where n_i is the size of c_i.

The complete code for this implementation of c-deques appears in Figure 8.4.

To improve the running time of $+\!\!+$ to $O(1)$ we modify the representation of c-deques so that $+\!\!+$ does not recurse. The key is to enable $+\!\!+$ at one level to call only $cons$ and $snoc$ at the next level. Instead of a front, a middle, and a rear, we expand deep c-deques to contain five segments: a *front* (F), an *antemedial* (A), a *middle* (M), a *postmedial* (B), and a *rear* (R). F, M, and R are all ordinary deques; F and R contain three or more elements each, and M contains two or more elements. A and B are c-deques of *compound elements*. A degenerate compound element is simply an ordinary deque containing two or more elements. A full compound element has three segments: a *front* (F), a *middle* (C), and a *rear* (R), where F and R are ordinary deques containing at least two elements each, and C is a c-deque of compound elements. This datatype can be written in Standard ML (with polymorphic recursion) as

```
    datatype α Cat = Shallow of α D.Queue
                   | Deep of {F : α D.Queue (* ≥ 3 *),
                              A : α CmpdElem Cat susp,
                              M : α D.Queue (* ≥ 2 *),
                              B : α CmpdElem Cat susp,
                              R : α D.Queue (* ≥ 3 *)}
    and α CmpdElem = Simple of α D.Queue (* ≥ 2 *)
                   | CE of {F : α D.Queue (* ≥ 2 *),
                            C : α CmpdElem Cat susp,
                            R : α D.Queue (* ≥ 2 *)}
```

```
functor SimpleCatenableDeque (structure D : DEQUE) : CATENABLEDEQUE =
  (* assumes polymorphic recursion! *)
struct
  datatype α Cat = Shallow of α D.Queue
                 | Deep of {F : α D.Queue, M : α D.Queue Cat susp, R : α D.Queue}

  exception EMPTY

  val empty = Shallow D.empty
  fun isEmpty (Shallow d) = D.isEmpty d
    | isEmpty _ = false

  fun cons (x, Shallow d) = Shallow (D.cons (x, d))
    | cons (x, Deep {F = f, M = m, R = r}) = Deep {F = D.cons (x, f), M = m, R = r}
  fun head (Shallow d) = if D.isEmpty d then raise EMPTY else D.head d
    | head (Deep {F = f, ...}) = D.head f
  fun tail (Shallow d) = if D.isEmpty d then raise EMPTY else Shallow (D.tail d)
    | tail (Deep {F = f, M = m, R = r}) =
        if D.size f > 2 then Deep {F = D.tail f, M = m, R = r}
        else if isEmpty (force m) then Shallow r
        else Deep {F = D.cons (D.last f, head (force m)), M = $tail (force m), R = r}

  ...snoc, last, and init defined symmetrically...

  fun shortAppendL (d1, d2) = if D.isEmpty d1 then d2 else D.cons (D.head d1, d2)
  fun shortAppendR (d1, d2) = if D.isEmpty d2 then d1 else D.snoc (d1, D.last d2)

  fun (Shallow d1) ++ (Shallow d2) =
        if D.size d1 < 2 then Shallow (shortAppendL (d1, d2))
        else if D.size d2 < 2 then Shallow (shortAppendR (d1, d2))
        else Deep {F = d1, M = $empty, R = d2}
    | (Shallow d) ++ (Deep {F = f, M = m, R = r}) =
        if D.size d < 2 then Deep {F = shortAppendL (d, f), M = m, R = r}
        else Deep {F = d, M = $cons (f, force m), R = r}
    | (Deep {F = f, M = m, R = r}) ++ (Shallow d) =
        if D.size d < 2 then Deep {F = f, M = m, R = shortAppendR (r, d)}
        else Deep {F = f, M = $snoc (force m, r), R = d}
    | (Deep {F = f1, M = m1, R = r1}) ++ (Deep {F = f2, M = m2, R = r2}) =
        Deep {F = f1, M = $(snoc (force m1, r1) ++ cons (f2, force m2)), R = r2}
end
```

Figure 8.4: Simple catenable deques.

Now, given two deep c-deques $c_1 = \langle F_1, A_1, M_1, B_1, R_1 \rangle$ and $c_2 = \langle F_2, A_2, M_2, B_2, R_2 \rangle$, we compute their catenation as follows: First, we retain F_1 as the front of the result, and R_2 as the rear of the result. Next, we build the new middle deque from the last element of R_1 and the first element of F_2. We then combine M_1, B_1, and the rest of R_1 into a compound element, which we $snoc$ onto A_1. This becomes the antemedial segment of the result. Finally, we combine the rest of F_2, A_2, and M_2 into a compound element, which we $cons$ onto B_2. This becomes the postmedial segment of the result. Altogether, this is implemented as

```
fun (Deep {F = $f_1$, A = $a_1$, M = $m_1$, B = $b_1$, R = $r_1$})
        ++ (Deep {F = $f_2$, A = $a_2$, M = $m_2$, B = $b_2$, R = $r_2$}) =
    let val ($r_1'$, $m$, $f_2'$) = share ($r_1$, $f_2$)
        val $a_1'$ = $snoc (force $a_1$, CE {F = $m_1$, A = $b_1$, R = $r_1'$})
        val $b_2'$ = $cons (CE {F = $f_2'$, A = $a_2$, R = $m_2$}, force $b_2$)
    in Deep {F = $f_1$, A = $a_1'$, M = $m$, B = $b_2'$, R = $r_2$} end
```

where

```
fun share ($f$, $r$) = (D.init $f$, D.cons (D.last $f$, D.cons (D.head $r$, D.empty)), D.tail $r$)
fun cons ($x$, Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}) =
    Deep {F = D.cons ($x$, $f$), A = $a$, M = $m$, B = $b$, R = $r$})
fun snoc (Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}, $x$) =
    Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = D.snoc ($r$, $x$)})
```

(For simplicity of presentation, we have ignored all cases involving shallow c-deques.)

Unfortunately, in this implementation, $tail$ and $init$ are downright messy. Since the two functions are symmetric, we describe only $tail$. Given some deep c-deque $c = \langle F, A, M, B, R \rangle$, there are six cases:

- $|F| > 3$.
- $|F| = 3$.
  - A is non-empty.
    - The first compound element of A is degenerate.
    - The first compound element of A is full.
  - A is empty and B is non-empty.
    - The first compound element of B is degenerate.
    - The first compound element of B is full.
  - A and B are both empty.

Here we describe the behavior of $tail\ c$ in the first three cases. The remaining cases are covered by the complete implementation in Figures 8.5 and 8.6. If $|F| > 3$ then we simply replace F with $D.tail\ F$. If $|F| = 3$, then removing an element from F would drop its length below the allowable minimum. Therefore, we remove a new front deque from A and combine it with the remaining two elements of the old F. The new F contains at least four elements, so the next call to $tail$ will fall into the $|F| > 3$ case.

When we remove the first compound element of A to find the new front deque, we get either a degenerate compound element or a full compound element. If we get a degenerate compound element (i.e., a simple deque), then the new value of A is **$**$tail$ ($force\ A$). If we get a full compound element $\langle F', C', R' \rangle$, then F' becomes the new F (along with the remaining elements of the old F), and the new value of A is

$(force C' ++ cons (Simple R', tail (force A)))

But note that the effect of the $cons$ and $tail$ is to replace the first element of A. We can do this directly, and avoid an unnecessary call to $tail$, using the function $replaceHead$.

```
fun replaceHead (x, Shallow d) = Shallow (D.cons (x, D.tail d))
  | replaceHead (x, Deep {F = f, A = a, M = m, B = b, R = r}) =
      Deep {F = D.cons (x, D.tail f), A = a, M = m, B = b, R = r})
```

The remaining cases of $tail$ are similar, each doing $O(1)$ work followed by at most one call to $tail$.

The $cons$, $snoc$, $head$, and $last$ functions make no use of lazy evaluation, and are easily seen to take $O(1)$ worst-case time. We analyze the remaining functions using the banker's method and debit passing.

As always, we assign debits to every suspension, each of which is the antemedial (A) or postmedial (B) segment of a deep c-deque, or the middle (C) of a compound element. Each C field is allowed four debits, but A and B fields may have from zero to five debits, based on the lengths of the F and R fields. A and B have a base allowance of zero debits. If F contains more than three elements, then the allowance for A increases by four debits and the allowance for B increases by one debit. Similarly, if R contains more than three elements, then the allowance for B increases by four debits and the allowance for A increases by one debit.

Theorem 8.4 *++, $tail$, and $init$ run in $O(1)$ amortized time.*

Proof: (++) The interesting case is catenating two deep c-deques $c_1 = \langle F_1, A_1, M_1, B_1, R_1 \rangle$ and $c_2 = \langle F_2, A_2, M_2, B_2, R_2 \rangle$. In that case, ++ does $O(1)$ unshared work and discharges at most four debits. First, we create two debits for the suspended $snoc$ and $cons$ onto A_1 and B_2, respectively. We always discharge these two debits. In addition, if B_1 or A_2 has five debits, then we must discharge one debit when that segment becomes the middle of a compound element.

```
functor ImplicitCatenableDeque (structure D : DEQUE) : CATENABLEDEQUE =
struct
  datatype $\alpha$ Cat = Shallow of $\alpha$ D.Queue
                  | Deep of {F : $\alpha$ D.Queue, A : $\alpha$ CmpdElem Cat susp, M : $\alpha$ D.Queue,
                             B : $\alpha$ CmpdElem Cat susp, R : $\alpha$ D.Queue}
  and $\alpha$ CmpdElem = Simple of $\alpha$ D.Queue
                   | CE of {F : $\alpha$ D.Queue, A : $\alpha$ CmpdElem Cat susp, R : $\alpha$ D.Queue}

  exception EMPTY

  val empty = Shallow D.empty
  fun isEmpty (Shallow $d$) = D.isEmpty $d$
    | isEmpty _ = false

  fun cons ($x$, Shallow $d$) = Shallow (D.cons ($x$, $d$))
    | cons ($x$, Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}) =
        Deep {F = D.cons ($x$, $f$), A = $a$, M = $m$, B = $b$, R = $r$})
  fun head (Shallow $d$) = if D.isEmpty $d$ then raise EMPTY else D.head $d$
    | head (Deep {F = $f$, . . . }) = D.head $f$

  . . . snoc and last defined symmetrically. . .

  fun share ($f$, $r$) = (D.init $f$, D.cons (D.last $f$, D.cons (D.head $r$, D.empty)), D.tail $r$)
  fun shortAppendL ($d_1$, $d_2$) =
        if D.isEmpty $d_1$ then $d_2$ else shortAppendL (D.init $d_1$, D.cons (D.last $d_1$, $d_2$))
  fun shortAppendR ($d_1$, $d_2$) =
        if D.isEmpty $d_2$ then $d_1$ else shortAppendR (D.snoc ($d_1$, D.head $d_2$), D.tail $d_2$)

  fun (Shallow $d_1$) ++ (Shallow $d_2$) =
        if D.size $d_1 < 4$ then Shallow (shortAppendL ($d_1$, $d_2$))
        else if D.size $d_2 < 4$ then Shallow (shortAppendR ($d_1$, $d_2$))
        else let val ($f$, $m$, $r$) = share ($d_1$, $d_2$)
             in Deep {F = $f$, A = $empty, M = $m$, B = $empty, R = $r$} end
    | (Shallow $d$) ++ (Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}) =
        if D.size $d < 3$ then Deep {F = shortAppendL ($d$, $f$), A = $a$, M = $m$, B = $b$, R = $r$}
        else Deep {F = $d$, A = $cons (Simple $f$, force $a$), M = $m$, B = $b$, R = $r$}
    | (Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}) ++ (Shallow $d$) =
        if D.size $d < 3$ then Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = shortAppendR ($r$, $d$)}
        else Deep {F = $f$, A = $a$, M = $m$, B = $snoc (force $b$, Simple $r$), R = $d$}
    | (Deep {F = $f_1$, A = $a_1$, M = $m_1$, B = $b_1$, R = $r_1$})
             ++ (Deep {F = $f_2$, A = $a_2$, M = $m_2$, B = $b_2$, R = $r_2$}) =
        let val ($r_1'$, $m$, $f_2'$) = share ($r_1$, $f_2$)
            val $a_1'$ = $snoc (force $a_1$, CE {F = $m_1$, A = $b_1$, R = $r_1'$})
            val $b_2'$ = $cons (CE {F = $f_2'$, A = $a_2$, R = $m_2$}, force $b_2$)
        in Deep {F = $f_1$, A = $a_1'$, M = $m$, B = $b_2'$, R = $r_2$} end
  . . .
```

Figure 8.5: Catenable deques using implicit recursive slowdown (part I).

```
  ...
  fun replaceHead ($x$, Shallow $d$) = Shallow (D.cons ($x$, D.tail $d$))
    | replaceHead ($x$, Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}) =
        Deep {F = D.cons ($x$, D.tail $f$), A = $a$, M = $m$, B = $b$, R = $r$})

  fun tail (Shallow $d$) = if D.isEmpty $d$ then raise EMPTY else Shallow (D.tail $d$)
    | tail (Deep {F = $f$, A = $a$, M = $m$, B = $b$, R = $r$}) =
        if D.size $f > 3$ then Deep {F = D.tail $f$, A = $a$, M = $m$, B = $b$, R = $r$}
        else if not (isEmpty (force $a$)) then
          case head (force $a$) of
            Simple $d$ ⇒
              let val $f'$ = shortAppendL (D.tail $f$, $d$)
              in Deep {F = $f'$, A = $tail (force $a$), M = $m$, B = $b$, R = $r$} end
          | CE {F = $f'$, A = $a'$, R = $r'$} ⇒
              let val $f''$ = shortAppendL (D.tail $f$, $f'$)
                  val $a''$ = $(force $a'$ ++ replaceHead (Simple $r'$, force $a$))
              in Deep {F = $f''$, A = $a''$, M = $m$, B = $b$, R = $r$} end
        else if not (isEmpty (force $b$)) then
          case head (force $b$) of
            Simple $d$ ⇒
              let val $f'$ = shortAppendL (D.tail $f$, $m$)
              in Deep {F = $f'$, A = $empty, M = $d$, B = $tail (force $b$), R = $r$} end
          | CE {F = $f'$, A = $a'$, R = $r'$} ⇒
              let val $f''$ = shortAppendL (D.tail $f$, $m$)
                  val $a''$ = $cons (Simple $f'$, force $a'$)
              in Deep {F = $f''$, A = $a''$, M = $r'$, B = $tail (force $b$), R = $r$} end
        else Shallow (shortAppendL (D.tail $f$, $m$)) ++ Shallow $r$

  ... replaceLast and init defined symmetrically...
end
```

Figure 8.6: Catenable deques using implicit recursive slowdown (part II).

Also, if F_1 has only three elements but F_2 has more than three elements, then we must discharge a debit from B_2 as it becomes the new B. Similarly for R_1 and R_2. However, note that if B_1 has five debits, then F_1 has more than three elements, and that if A_2 has five debits, then R_2 has more than three elements. Therefore, we must discharge at most four debits altogether, or at least pass those debits to an enclosing suspension.

($tail$ and $init$) Since $tail$ and $init$ are symmetric, we include the argument only for $tail$. By inspection, $tail$ does $O(1)$ unshared work, so we must show that it discharges only $O(1)$ debits. In fact, we show that it discharges at most five debits.

Since $tail$ can call itself recursively, we must account for a cascade of $tail$s. We argue by debit passing. Given some deep c-deque $c = \langle F, A, M, B, R \rangle$, there is one case for each case of $tail$.

If $|F| > 3$, then this is the end of a cascade. We create no new debits, but removing an element from F might decrease the allowance of A by four debits, and the allowance of B by one debit, so we pass these debits to the enclosing suspension.

If $|F| = 3$, then assume A is non-empty. (The cases where A is empty are similar.) If $|R| > 3$, then A might have one debit, which we pass to the enclosing suspension. Otherwise, A has no debits. If the head of A is a degenerate compound element (i.e., a simple deque of elements), then this becomes the new F along with the remaining elements of the old F. The new A is a suspension of the tail of the old A. This suspension receives at most five debits from the recursive call to $tail$. Since the new allowance of A is at least four debits, we pass at most one of these debits to the enclosing suspension, for a total of at most two debits. (Actually, the total is at most one debit since we pass one debit here exactly in the case that we did not have to pass one debit for the original A).

Otherwise, if the head of A is a full compound element $\langle F', C', R' \rangle$, then F' becomes the new F along with the remaining elements of the old F. The new A involves calls to $+\!\!+$ and $replaceHead$. The total number of debits on the new A is nine: four debits from C', four debits from the $+\!\!+$, and one newly created debit for the $replaceHead$. The allowance for the new A is either four or five, so we pass either five or four of these nine debits to the enclosing suspension. Since we pass four of these debits exactly in the case that we had to pass one debit from the original A, we always pass at most five debits. $\square$

8.6 Related Work

Recursive Slowdown Kaplan and Tarjan introduced recursive slowdown in [KT95], and used it again in [KT96b], but it is closely related to the regularity constraints of Guibas et al. [GMPR77]. Brodal [Bro95] used a similar technique to implement heaps.

Implicit Recursive Slowdown and Binomial Heaps Lazy implementations of binomial heaps [Kin94, Oka96b] can be viewed as using implicit recursive slowdown. Such implementations support $insert$ in $O(1)$ amortized time and all other operations in $O(\log n)$ amortized time. [Oka96b] extends a lazy implementation of binomial heaps with scheduling to improve these bounds to worst-case.

Catenable Deques Buchsbaum and Tarjan [BT95] presented a purely functional implementation of catenable deques that supports $tail$ and $init$ in $O(\log^* n)$ worst-case time and all other operations in $O(1)$ worst-case time. Our implementation improves that bound to $O(1)$ for all operations, although in the amortized rather than worst-case sense. Kaplan and Tarjan have independently developed a similar implementation with worst-case bounds [KT96a]. However, the details of their implementation are quite complicated.

Chapter 9

Conclusions

In the preceding chapters, we have described a framework for designing and analyzing functional amortized data structures (Chapter 3), a method for eliminating amortization from such data structures (Chapter 4), four general data structure design techniques (Chapters 5–8), and sixteen new implementations of specific data structures. We next step back and reflect on the significance of this work.

9.1 Functional Programming

Functional programming languages have historically suffered from the reputation of being slow. Regardless of the advances in compiler technology, functional programs will never be faster than their imperative counterparts as long as the algorithms available to functional programmers are significantly slower than those available to imperative programmers. This thesis provides numerous functional data structures that are asymptotically just as efficient as the best imperative implementations. More importantly, we also provide numerous design techniques so that functional programmers can create their own data structures, customized to their particular needs.

Our most significant contribution to the field of functional programming, however, is the new understanding of the relationship between amortization and lazy evaluation. In the one direction, the techniques of amortized analysis — suitably extended as in Chapter 3 — provide the first practical approach to estimating the complexity of lazy programs. Previously, functional programmers often had no better option than to pretend their lazy programs were actually strict.

In the other direction, lazy evaluation allows us to implement amortized data structures that are efficient even when used persistently. Amortized data structures are desirable because they are often both simpler and faster than their worst-case counterparts. Without exception, the amortized data structures described in this thesis are significantly simpler than competing worst-case designs.[1] Because of the overheads of lazy evaluation, however, our amortized data structures are not necessarily faster than their strict worst-case cousins. When used in a mostly single-threaded fashion, our implementations are often slower than competing implementations not based on memoization, because most of the time spent doing memoization is wasted. However, when persistence is used heavily, memoization more than pays for itself and our implementations fly.

In a followup to [Pip96], Bird, Jones, and de Moor [BJdM96] have recently exhibited a problem for which a lazy solution exists that is asymptotically superior to any possible strict solution. However, this result depends on several extremely restrictive assumptions. Our work suggests a promising approach towards removing these restrictions. What is required is an example of a data structure for which a lazy, amortized solution exists that is asymptotically superior to any possible strict, worst-case solution. Unfortunately, at this time, we know of no such data structure — for every lazy, amortized data structure we have developed, there is a strict, worst-case data structure with equivalent bounds, albeit one that is more complicated.

9.2 Persistent Data Structures

We have shown that memoization, in the form of lazy evaluation, can resolve the apparent conflict between amortization and persistence. We expect to see many persistent amortized data structures based on these ideas in the coming years.

We have also reinforced the observation that functional programming is an excellent medium for developing new persistent data structures, even when the target language is imperative. It is trivial to implement most functional data structures in an imperative language such as C, and such implementations suffer few of the complications and overheads associated with other methods for implementing persistent data structures, such as [DSST89] or [Die89]. Furthermore, unlike these other methods, functional programming has no problems with data structures that support combining functions such as list catenation. It is no surprise that the best persistent implementations of data structures such as catenable lists (Section 7.2.1) and catenable deques (Section 8.5) are all purely functional (see also [KT95, KT96a]).

9.3 Programming Language Design

Next, we briefly discuss the implications of this work on programming language design.

[1]As partial evidence for this fact, we note that only one of these implementations takes more than one page.

Order of Evaluation Most functional programming languages support either strict evaluation or lazy evaluation, but not both. Algorithmically, the two orders of evaluation fulfill complementary roles — strict evaluation is useful in implementing worst-case data structures and lazy evaluation is useful in implementing amortized data structures. Therefore, functional programming languages that purport to be general-purpose should support both. **$**-notation offers a lightweight syntax for integrating lazy evaluation into a predominantly strict language.

Polymorphic Recursion Data structures based on structural decomposition, such as those in Chapters 7 and 8, often obey invariants that can be precisely captured by non-uniform recursive datatypes. Unfortunately, processing such datatypes requires polymorphic recursion, which causes difficulties for type inference and hence is disallowed by most functional programming languages. We can usually sidestep this restriction by rewriting the datatypes to be uniform, but then the types fail to capture the desired invariants and the type system will not catch bugs involving violations of those invariants. All in all, we believe the compromise taken by Haskell 1.3 [P^{+}96] is best: allow polymorphic recursion in those cases where the programmer explicitly provides a type signature, and disallow it everywhere else.

Higher-order, Recursive Modules The bootstrapped heaps of Section 7.2.2 (see also [BO96]) demonstrate the usefulness of higher-order, recursive modules. In languages such as Standard ML that do not support higher-order, recursive modules, we can often sidestep this restriction by manually inlining the desired definitions for each instance of bootstrapping. Clearly, however, it would be cleaner, and much less error-prone, to provide a single module-to-module transformation that performs the bootstrapping. In the case of bootstrapped heaps, Simon Peyton Jones and Jan Nicklisch [private communication] have recently shown how to implement the desired recursion using constructor classes [Jon95].

Pattern Matching Ironically, pattern matching — one of the most popular features in functional programming languages — is also one of the biggest obstacles to the widespread use of efficient functional data structures. The problem is that pattern matching can only be performed on data structures whose representation is known, yet the basic software-engineering principle of abstraction tells us that the representation of non-trivial data structures should be hidden. The seductive allure of pattern matching leads many functional programmers to abandon sophisticated data structures in favor of simple, known representations such as lists, even when doing so causes an otherwise linear algorithm to explode to quadratic or even exponential time.

Views [Wad87] and their successors [BC93, PPN96] offer one way of reconciling the convenience of pattern matching with the desirability of data abstraction. In fact, **$**-patterns are just a special case of views. Unfortunately, views are not supported by any major functional programming language.

Implementation Finally, we note that functional catenable lists are an essential ingredient in the implementation of certain sophisticated control structures [FWFD88]. The advent of new, efficient implementations of catenable lists, both here and in [KT95], makes the efficient implementation of such control structures possible for the first time.

9.4 Open Problems

We conclude by describing some of the open problems related to this thesis.

- What are appropriate empirical measurements for persistent data structures? Standard benchmarks are misleading since they do not measure how well a data structure supports access to older versions. Unfortunately, the theory and practice of benchmarking persistent data structures is still in its infancy.

- For ephemeral data structures, the physicist's method is just as powerful as the banker's method. However, for persistent data structures, the physicist's method appears to be substantially weaker. Can the physicist's method, as described in Section 3.5, be improved and made more widely applicable?

- The catenable deques of Section 8.5 are substantially more complicated than the catenable lists of Section 7.2.1. Is there a simpler implementation of catenable deques closer in spirit to that of catenable lists?

- Finally, can scheduling be applied to these implementations of catenable lists and deques? In both cases, maintaining a schedule appears to take more than $O(1)$ time.

Appendix A

The Definition of Lazy Evaluation in Standard ML

The syntax and semantics of Standard ML are formally specified in *The Definition of Standard ML* [MTH90]. This appendix extends the *Definition* with the syntax and semantics of the lazy evaluation primitives (**$**-notation) described in Chapter 2. This appendix is designed to be read in conjunction with the *Definition*; it describes only the relevant changes and additions.

Paragraph headers such as **[2.8 Grammar (8,9)]** refer to sections within the *Definition*. The numbers in parentheses specify the relevant pages.

A.1 Syntax

[2.1 Reserved Words (3)] **$** is a reserved word and may not be used as an identifier.

[2.8 Grammar (8,9)] Add the following productions for expressions and patterns.

$$exp \ ::= \ \mathbf{\$}\, exp \qquad \text{and} \qquad pat \ ::= \ \mathbf{\$}\, pat$$

[Appendix B: Full Grammar (71–73)] Add the following productions for expressions and patterns.

$$exp \ ::= \ \mathbf{\$}\, exp \qquad \text{and} \qquad pat \ ::= \ \mathbf{\$}\, pat$$

These productions have lower precedence than any alternative form (i.e., appear last in the lists of alternatives).

A.2 Static Semantics

[4.4 Types and Type functions (18)] τ `susp` does not admit equality.

Remark: This is an arbitrary choice. Allowing an equality operator on suspensions that automatically forces the suspensions and compares the results would also be reasonable, but would be moderately complicated. ◇

[4.7 Non-expansive Expressions (20)] **$** expressions are non-expansive.

Remark: The dynamic evaluation of a **$** expression may in fact extend the domain of memory, but, for typechecking purposes, suspensions should be more like functions than references. ◇

[4.10 Inference Rules (24,29)] Add the following inference rules.

$$\frac{C \vdash \textit{exp} \Rightarrow \tau}{C \vdash \textbf{\$}\ \textit{exp} \Rightarrow \tau\ \text{susp}} \qquad \text{and} \qquad \frac{C \vdash \textit{pat} \Rightarrow \tau}{C \vdash \textbf{\$}\ \textit{pat} \Rightarrow \tau\ \text{susp}}$$

[4.11 Further Restrictions (30)] Because matching against a **$** pattern may have effects (in particular, may cause assignments), it is now more difficult to determine if matches involving both suspensions and references are *irredundant* and *exhaustive*. For example, the first function below is non-exhaustive even though the first and third clauses appear to cover all cases and the second is irredundant even though the first and fourth clauses appear to overlap.

```
fun f (ref true,  _) = 0
  | f (ref false, $0) = 1
  | f (ref false, _) = 2
```

```
fun f (ref true,  _) = 0
  | f (ref false, $0) = 1
  | f (ref false, _) = 2
  | f (ref true,  _) = 3
```

(Consider the execution of `f (r,$(r := true; 1))` where `r` initially equals `ref false`.)

[Appendix C: The Initial Static Basis (74,75)] Extend T_0 to include `susp`, which has arity 1 and does not admit equality.

Add `force` to VE_0 (Figure 23), where

$$\texttt{force} \mapsto \forall\, \texttt{'a.'a susp} \to \texttt{'a}$$

A.3 Dynamic Semantics

[6.3 Compound Objects (47)] Add the following definitions to Figure 13.

$$\begin{array}{rcl} (\mathit{exp}, E) & \in & \text{Thunk} = \text{Exp} \times \text{Env} \\ \mathit{mem} & \in & \text{Mem} = \text{Addr} \overset{\text{fin}}{\to} (\text{Val} \cup \text{Thunk}) \end{array}$$

Remark: Addresses and memory are overloaded to represent both references and suspensions. The values of both references and suspensions are addresses. Addresses representing references are always mapped to values, but addresses representing suspensions may be mapped to either thunks (if unevaluated) or values (if evaluated and memoized). The static semantics ensures that there will be no confusion about whether a value in memory represents a reference or a memoized suspension. ◇

[6.7 Inference Rules (52,55,56)] Add the following inference rule for suspending an expression.

$$\frac{a \notin \text{Dom}(\mathit{mem} \text{ of } s)}{s, E \vdash \textbf{\$}\ \mathit{exp} \Rightarrow a, s + \{a \mapsto (\mathit{exp}, E)\}}$$

Extend the signatures involving pattern rows and patterns to allow exceptions to be raised during pattern matching.

$$E, r \vdash \mathit{patrow} \Rightarrow \mathit{VE}/\text{FAIL}/p$$

$$E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}/p$$

Add the following inference rules for forcing a suspension.

$$\frac{s(a) = v \qquad s, E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}, s'}{s, E, a \vdash \textbf{\$}\ \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}, s'}$$

$$\frac{s(a) = (\mathit{exp}, E') \qquad s, E' \vdash \mathit{exp} \Rightarrow v, s' \qquad s' + \{a \mapsto v\}, E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}, s''}{s, E, a \vdash \textbf{\$}\ \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}, s''}$$

The first rule looks up a memoized value. The second rule evaluates a suspension and memoizes the result.

Finally, modify Rule 158 to reflect the fact that matching against a pattern may change the state.

$$\frac{s(a) = v \qquad s, E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}, s'}{s, E, a \vdash \texttt{ref}\ \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}, s'} \qquad (158)$$

Remark: The interaction between suspensions and exceptions is specified by the exception convention. If an exception is raised while forcing a suspension, the evaluation of that suspension is aborted and the result is not memoized. Forcing the suspension a second time will duplicate any side effects it may have. A reasonable alternative would be to memoize raised exceptions, so that forcing such a suspension a second time would simply reraise the memoized exception without duplicating any side effects. ◇

[Appendix D: The Initial Dynamic Basis (77,79)] Extend E_0'' with the following declaration:

fun force (**$***x*) = *x*

A.4 Recursion

This section details the changes necessary to support recursive suspensions.

[2.9 Syntactic Restrictions (9)] Lift the syntactic restriction on `rec` to allow value bindings of the form var = **$** *exp* within `rec`.

[6.7 Inference Rules (54)] Modify Rule 137 as follows.

$$\frac{s, E \vdash valbind \Rightarrow \mathit{VE}, s' \qquad \mathit{VE}' = \mathrm{Rec}\,\mathit{VE} \qquad s'' = \mathrm{SRec}(\mathit{VE}', s')}{s, E \vdash \texttt{rec}\ valbind \Rightarrow \mathit{VE}', s''} \qquad 137$$

where

$$\mathrm{SRec} : \mathrm{VarEnv} \times \mathrm{State} \rightarrow \mathrm{State}$$

and

- *ens* of $\mathrm{SRec}(\mathit{VE}, s) =$ *ens* of s
- $\mathrm{Dom}($*mem* of $\mathrm{SRec}(\mathit{VE}, s)) = \mathrm{Dom}($*mem* of $s)$
- If $a \notin \mathrm{Ran}(\mathit{VE})$, then $\mathrm{SRec}(\mathit{VE}, s)(a) = s(a)$
- If $a \in \mathrm{Ran}(\mathit{VE})$ and $s(a) = (\mathit{exp}, E)$, then $\mathrm{SRec}(\mathit{VE}, s)(a) = (\mathit{exp}, E + \mathit{VE})$

The SRec operator defines recursive suspensions by “tying the knot” through the memory. Note that in the definition of SRec, it will never be the case that $a \in \mathrm{Ran}(\mathit{VE})$ and $s(a) \notin$ Thunk, because the suspension could not have been forced yet.

Remark: In the presence of recursion, a suspension might be memoized more than once if evaluating its body somehow forces itself. Then, the inner evaluation might produce and memoize a value that is subsequently overwritten by the result of the outer evaluation. Note, however, that evaluating a suspension that forces itself will not terminate unless side effects are involved. If desired, the “blackhole” technique [Jon92] can be used to detect such circular suspensions and guarantee that a given suspension is only memoized once. $\diamond$

Bibliography

[Ada93] Stephen Adams. Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993. (p. 17)

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages*, pages 233–246, January 1995. (p. 10)

[AVL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962. English translation of Russian orginal appearing in *Doklady Akademia Nauk SSSR*, 146:263-266. (p. 49)

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. (p. 1)

[BAG92] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992. (p. 2)

[BC93] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993. (p. 129)

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957. (p. 12)

[BH89] Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Conference on Functional Programming Languages and Computer Architecture*, pages 157–165, September 1989. (p. 38)

[BJdM96] Richard S. Bird, Geraint Jones, and Oege de Moor. A lazy pure language versus impure Lisp. `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications/FP-1-96.html`, 1996. (p. 128)

[Blu86] Norbert Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4):1021–1024, November 1986. (p. 14)

[BO96] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6), December 1996. To appear. (pp. 82, 84, 103, 129)

[Bro78] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, August 1978. (pp. 64, 68, 73, 84)

[Bro95] Gerth Stølting Brodal. Fast meldable priority queues. In *Workshop on Algorithms and Data Structures*, volume 955 of *LNCS*, pages 282–290. Springer-Verlag, August 1995. (pp. 103, 124)

[Bro96] Gerth Stølting Brodal. Worst-case priority queues. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, January 1996. (p. 103)

[BST95] Adam L. Buchsbaum, Rajamani Sundar, and Robert E. Tarjan. Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues. *SIAM Journal on Computing*, 24(6):1190–1206, December 1995. (p. 101)

[BT95] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. *Journal of Algorithms*, 18(3):513–547, May 1995. (pp. 58, 101, 125)

[Buc93] Adam L. Buchsbaum. *Data-structural bootstrapping and catenable deques*. PhD thesis, Department of Computer Science, Princeton University, June 1993. (pp. 6, 85, 101)

[Bur82] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982. (pp. 16, 18, 37)

[But83] T. W. Butler. Computer response time and user performance. In *Conference on Human Factors in Computing Systems*, pages 58–62, December 1983. (p. 39)

[CG93] Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, June 1993. (pp. 55, 58)

[CMP88] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Scandinavian Workshop on Algorithm Theory*, volume 318 of *LNCS*, pages 1–13. Springer-Verlag, July 1988. (pp. 48, 82)

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, November 1988. (pp. 48, 103)

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, pages 122–127, May 1982. (p. 101)

[Die89] Paul F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *LNCS*, pages 67–74. Springer-Verlag, August 1989. (pp. 37, 128)

[DR91] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 78–88, January 1991. (p. 48)

[DR93] Paul F. Dietz and Rajeev Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications. In *Workshop on Algorithms and Data Structures*, volume 709 of *LNCS*, pages 289–301. Springer-Verlag, August 1993. (p. 48)

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *ACM Symposium on Theory of Computing*, pages 365–372, May 1987. (p. 58)

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989. (pp. 2, 12, 20, 37, 128)

[DST94] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, September 1994. (pp. 4, 37, 101)

[Fag96] Rolf Fagerberg. A generalization of binomial queues. *Information Processing Letters*, 57(2):109–114, January 1996. (p. 84)

[FMR72] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the ACM*, 19(4):590–607, October 1972. (p. 58)

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. (p. 103)

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. (pp. 12, 103)

[FW76] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, pages 257–281, July 1976. (p. 11)

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: a mathematical semantics for handling full functional jumps. In *ACM Conference on LISP and Functional Programming*, pages 52–62, July 1988. (p. 130)

[GMPR77] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *ACM Symposium on Theory of Computing*, pages 49–60, May 1977. (pp. 82, 84, 124)

[Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981. (pp. 16, 18, 37)

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *IEEE Symposium on Foundations of Computer Science*, pages 8–21, October 1978. (p. 49)

[GT86] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, April 1986. (p. 58)

[H$^+$92] Paul Hudak et al. Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices*, 27(5), May 1992. (p. 7)

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993. (p. 103)

[HJ94] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. …An experiment in software prototyping productivity, 1994. (p. 1)

[HM76] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *ACM Symposium on Principles of Programming Languages*, pages 95–103, January 1976. (p. 11)

[HM81] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, November 1981. (pp. 16, 18, 37, 41, 48, 51, 58)

[Hoo82] Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs*. PhD thesis, Department of Computer Science, Cornell University, August 1982. (Cornell TR 82-503). (p. 58)

[Hoo92a] Rob R. Hoogerwoord. A logarithmic implementation of flexible arrays. In *Conference on Mathematics of Program Construction*, volume 669 of *LNCS*, pages 191–207. Springer-Verlag, July 1992. (p. 84)

[Hoo92b] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992. (pp. 55, 58)

[HU73] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, December 1973. (p. 12)

[Hug85] John Hughes. Lazy memo functions. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer-Verlag, September 1985. (p. 11)

[Hug86] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, March 1986. (pp. 82, 101)

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989. (pp. 1, 58)

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992. (p. 135)

[Jon95] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995. (p. 129)

[Jos89] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68(1):105–111, October 1989. (p. 10)

[KD96] Anne Kaldewaij and Victor J. Dielissen. Leaf trees. *Science of Computer Programming*, 26(1–3):149–165, May 1996. (pp. 64, 66, 84)

[Kin94] David J. King. Functional binomial queues. In *Glasgow Workshop on Functional Programming*, pages 141–150, September 1994. (pp. 84, 125)

[KL93] Chan Meng Khoong and Hon Wai Leong. Double-ended binomial queues. In *International Symposium on Algorithms and Computation*, volume 762 of *LNCS*, pages 128–137. Springer-Verlag, December 1993. (p. 103)

[Knu73] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1973. (p. 62)

[KT95] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *ACM Symposium on Theory of Computing*, pages 93–102, May 1995. (pp. 6, 84, 103, 105, 107, 124, 128, 130, 142)

[KT96a] Haim Kaplan and Robert E. Tarjan. Purely functional lists with catenation via recursive slow-down. Draft revision of [KT95], August 1996. (pp. 125, 128)

[KT96b] Haim Kaplan and Robert E. Tarjan. Purely functional representations of catenable sorted lists. In *ACM Symposium on Theory of Computing*, pages 202–211, May 1996. (pp. 4, 82, 84, 124)

[KTU93] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993. (p. 103)

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965. (pp. 11, 58)

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 144–154, January 1993. (p. 10)

[LS81] Benton L. Leong and Joel I. Seiferas. New real-time simulations of multihead tape units. *Journal of the ACM*, 28(1):166–180, January 1981. (p. 58)

[Mic68] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968. (pp. 2, 11)

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*, pages 409–423, April 1994. (p. 100)

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990. (pp. 4, 131)

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Spring-Verlag, April 1984. (pp. 87, 103)

[Mye83] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983. (pp. 76, 82, 84)

[Mye84] Eugene W. Myers. Efficient applicative data types. In *ACM Symposium on Principles of Programming Languages*, pages 66–75, January 1984. (pp. 84, 101)

[Oka95a] Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995. (pp. 37, 103)

[Oka95b] Chris Okasaki. Purely functional random-access lists. In *Conference on Functional Programming Languages and Computer Architecture*, pages 86–95, June 1995. (pp. 76, 78, 84)

[Oka95c] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995. (pp. 37, 42, 43, 48, 58)

[Oka96a] Chris Okasaki. Functional data structures. In *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 131–158. Springer-Verlag, August 1996. (pp. 90, 103)

[Oka96b] Chris Okasaki. The role of lazy evaluation in amortized data structures. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–72, May 1996. (pp. 37, 48, 125)

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–81, January 1994. (p. 10)

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *LNCS*. Springer-Verlag, 1983. (pp. 6, 48, 49, 51, 58)

[P$^+$96] John Peterson et al. Haskell 1.3: A non-strict, purely functional language. `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`, May 1996. (pp. 103, 129)

[Pau91] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991. (p. 84)

[Pet87] Gary L. Peterson. A balanced tree scheme for meldable heaps with updates. Technical Report GIT-ICS-87-23, School of Information and Computer Science, Georgia Institute of Technology, 1987. (p. 103)

[Pip96] Nicholas Pippenger. Pure versus impure Lisp. In *ACM Symposium on Principles of Programming Languages*, pages 104–109, January 1996. (pp. 2, 128)

[PPN96] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *ACM SIGPLAN International Conference on Functional Programming*, pages 110–121, May 1996. (p. 129)

[Ram92] Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Times*. PhD thesis, Department of Computer Sciences, University of Rochester, October 1992. (pp. 4, 37, 39, 48)

[San90] David Sands. Complexity analysis for a lazy higher-order language. In *European Symposium on Programming*, volume 432 of *LNCS*, pages 361–376. Springer-Verlag, May 1990. (p. 38)

[San95] David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, August 1995. (p. 38)

[Sar86] Neil Sarnak. *Persistent Data Structures*. PhD thesis, Department of Computer Sciences, New York University, 1986. (p. 58)

[Sch92] Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis, Eindhoven University of Technology, September 1992. (p. 15)

[Sch93] Berry Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45(1):41–50, January 1993. (p. 37)

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, Massachusetts, 1986. (p. 82)

[SS90] Jörg-Rüdiger Sack and Thomas Strothotte. A characterization of heaps and its applications. *Information and Computation*, 86(1):69–86, May 1990. (p. 64)

[ST85] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985. (p. 21)

[ST86a] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986. (p. 68)

[ST86b] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986. (pp. 12, 21, 103)

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988. (p. 39)

[Sto70] Hans-Jörg Stoß. K-band simulation von k-Kopf-Turing-maschinen. *Computing*, 6(3):309–317, 1970. (p. 58)

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, 1983. (p. 37)

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985. (pp. 14, 15)

[Tsa85] Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1–3):173–194, October 1985. (p. 84)

[TvL84] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, April 1984. (pp. 12, 14)

[Vui74] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, December 1974. (p. 10)

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978. (pp. 61, 64, 68, 73, 84)

[W$^+$90] Pierre Weis et al. The CAML reference manual (version 2.6.1). Technical Report 121, INRIA-Rocquencourt, September 1990. (p. 12)

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lamda-Calculus*. PhD thesis, University of Oxford, September 1971. (p. 10)

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 307–313, January 1987. (p. 129)

[Wad88] Philip Wadler. Strictness analysis aids time analysis. In *ACM Symposium on Principles of Programming Languages*, pages 119–132, January 1988. (p. 38)

[WV86] Christopher Van Wyk and Jeffrey Scott Vitter. The complexity of hashing with lazy deletion. *Algorithmica*, 1(1):17–29, 1986. (p. 12)

Index