1
Introduction

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 most of these books purport to be language-independent, they are unfortunately language-independent only in the sense of Henry Ford: Programmers can use any language they want, as long as it's imperative.† To rectify this imbalance, this book describes data structures from a functional point of view. We use Standard ML for all our examples, but the programs are easily translated into other functional languages such as Haskell or Lisp. We include Haskell versions of our programs in Appendix A.

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

† 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."

the point of view of designing and implementing efficient data structures, functional programming's stricture against destructive updates (i.e., 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 light of all these points, functional data structures sometimes seem like the dancing bear, of whom it is said, "the amazing thing is not that [he] dances so well, but that [he] dances at all!" In practice, however, the situation is not nearly so bleak. As we shall see, 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 sometimes 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 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 shall see, 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. We achieve this by extending Standard ML with lazy evaluation primitives as described in Chapter 4.

1.3 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 often 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.†

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 *function* or *operator* for the former.

1.4 Approach

Rather than attempting to catalog 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 sequences, heaps (priority queues), and search structures. Once you understand the techniques involved, you can easily adapt existing data structures to your particular needs, or even design new data structures from scratch.

1.5 Overview

This book is structured in three parts. The first part (Chapters 2 and 3) serves as an introduction to functional data structures.

- Chapter 2 describes how functional data structures achieve persistence.
- Chapter 3 examines three familiar data structures—leftist heaps, binomial heaps, and red-black trees—and shows how they can be implemented in Standard ML.

The second part (Chapters 4–7) concerns the relationship between lazy evaluation and amortization.

- Chapter 4 sets the stage by briefly reviewing the basic concepts of lazy evaluation and introducing the notation we use for describing lazy computations in Standard ML.
- Chapter 5 reviews the basic techniques of amortization and explains why these techniques are not appropriate for analyzing persistent data structures.

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

- Chapter 6 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 7 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.

The third part of the book (Chapters 8–11) explores a handful of general techniques for designing functional data structures.

- Chapter 8 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. Combining lazy rebuilding with the scheduling techniques of Chapter 7 often restores the worst-case bounds.
- Chapter 9 explores *numerical representations*, which are 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 10 examines *data-structural bootstrapping* [Buc93]. This technique comes in three flavors: *structural decomposition*, in which unbounded solutions are bootstrapped from bounded solutions; *structural abstraction*, in which efficient solutions are bootstrapped from inefficient solutions; and *bootstrapping to aggregate types*, in which implementations with atomic elements are bootstrapped to implementations with aggregate elements.
- Chapter 11 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 often recover the worst-case bounds using scheduling.

Finally, Appendix A includes Haskell translations of most of the implementations in this book.

2
Persistence

A distinctive property of functional data structures is that they are always *persistent*—updating a functional data structure does not destroy the existing version, but rather creates a new version that coexists with the old one. Persistence is achieved by *copying* the affected nodes of a data structure and making all changes in the copy rather than in the original. Because nodes are never modified directly, all nodes that are unaffected by an update can be *shared* between the old and new versions of the data structure without worrying that a change in one version will inadvertently be visible to the other.

In this chapter, we examine the details of copying and sharing for two simple data structures: lists and binary search trees.

2.1 Lists

We begin with simple linked lists, which are common in imperative programming and ubiquitous in functional programming. The core functions supported by lists are essentially those of the stack abstraction, which is described as a Standard ML signature in Figure 2.1. Lists and stacks can be implemented trivially using either the built-in type of lists (Figure 2.2) or a custom datatype (Figure 2.3).

Remark The signature in Figure 2.1 uses list nomenclature (cons, head, tail) rather than stack nomenclature (push, top, pop), because we regard stacks as an instance of the general class of sequences. Other instances include *queues*, *double-ended queues*, and *catenable lists*. We use consistent naming conventions for functions in all of these abstractions, so that different implementations can be substituted for each other with a minimum of fuss. ◇

Another common function on lists that we might consider adding to this signature is ++, which catenates (i.e., appends) two lists. In an imperative setting,

```
signature STACK =
sig
  type α Stack

  val empty   : α Stack
  val isEmpty : α Stack → bool

  val cons    : α × α Stack → α Stack
  val head    : α Stack → α          (* raises EMPTY if stack is empty *)
  val tail    : α Stack → α Stack    (* raises EMPTY if stack is empty *)
end
```

Figure 2.1. Signature for stacks.

```
structure List : STACK =
struct
  type α Stack = α list

  val empty = [ ]
  fun isEmpty s = null s

  fun cons (x, s) = x :: s
  fun head s = hd s
  fun tail s = tl s
end
```

Figure 2.2. Implementation of stacks using the built-in type of lists.

```
structure CustomStack : STACK =
struct
  datatype α Stack = NIL | CONS of α × α Stack

  val empty = NIL
  fun isEmpty NIL = true | isEmpty _ = false

  fun cons (x, s) = CONS (x, s)
  fun head NIL = raise EMPTY
    | head (CONS (x, s)) = x
  fun tail NIL = raise EMPTY
    | tail (CONS (x, s)) = s
end
```

Figure 2.3. Implementation of stacks using a custom datatype.

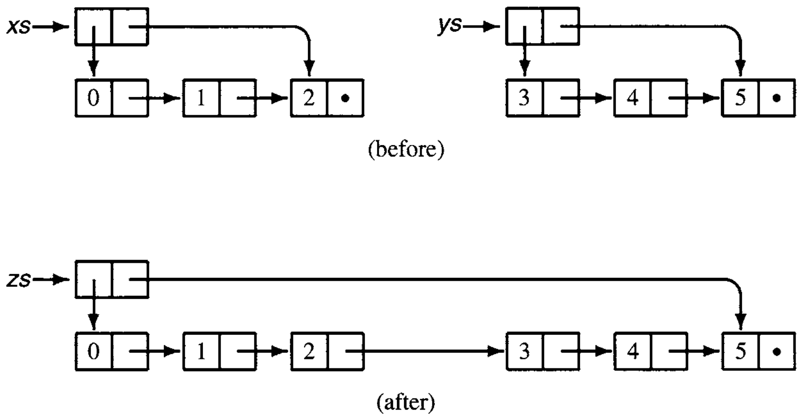

Figure 2.4. Executing *zs* = *xs* ++ *ys* in an imperative setting. Note that this operation destroys the argument lists, *xs* and *ys*.

this function can easily be supported in $O(1)$ time by maintaining pointers to both the first and last cell in each list. Then ++ simply modifies the last cell of the first list to point to the first cell of the second list. The result of this operation is shown pictorially in Figure 2.4. Note that this operation *destroys* both of its arguments—after executing *zs* = *xs* ++ *ys*, neither *xs* nor *ys* can be used again.

In a functional setting, we cannot destructively modify the last cell of the first list in this way. Instead, we *copy* the cell and modify the tail pointer of the copy. Then we copy the second-to-last cell and modify its tail to point to the copy of the last cell. We continue in this fashion until we have copied the entire list. This process can be implemented generically as

```
fun xs ++ ys = if isEmpty xs then ys else cons (head xs, tail xs ++ ys)
```

If we have access to the underlying representation (say, Standard ML's built-in lists), then we can rewrite this function using pattern matching as

```
fun [] ++ ys = ys
  | (x :: xs) ++ ys = x :: (xs ++ ys)
```

Figure 2.5 illustrates the result of catenating two lists. Note that after the oper-

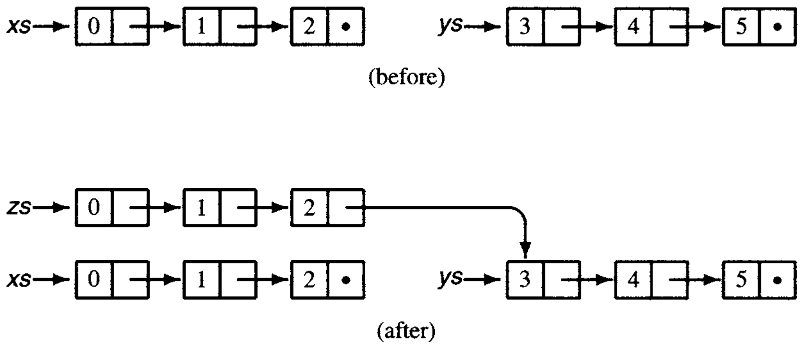

Figure 2.5. Executing *zs* = *xs* ++ *ys* in a functional setting. Notice that the argument lists, *xs* and *ys*, are unaffected by the operation.

ation, we are free to continue using the old lists, *xs* and *ys*, as well as the new list, *zs*. Thus, we get persistence, but at the cost of $O(n)$ copying.†

Although this is undeniably a lot of copying, notice that we did not have to copy the second list, *ys*. Instead, these nodes are shared between *ys* and *zs*. Another function that illustrates these twin concepts of copying and sharing is update, which changes the value of a node at a given index in the list. This function can be implemented as

```
fun update ([], i, y) = raise SUBSCRIPT
  | update (x :: xs, 0, y) = y :: xs
  | update (x :: xs, i, y) = x :: update (xs, i−1, y)
```

Here we do not copy the entire argument list. Rather, we copy only the node to be modified (node *i*) and all those nodes that contain direct or indirect pointers to node *i*. In other words, to modify a single node, we copy all the nodes on the path from the root to the node in question. All nodes that are not on this path are shared between the original version and the updated version. Figure 2.6 shows the results of updating the third node of a five-node list; the first three nodes are copied and the last two nodes are shared.

Remark This style of programming is greatly simplified by automatic garbage collection. It is crucial to reclaim the space of copies that are no longer needed, but the pervasive sharing of nodes makes manual garbage collection awkward.

† In Chapters 10 and 11, we will see how to support ++ in $O(1)$ time without sacrificing persistence.

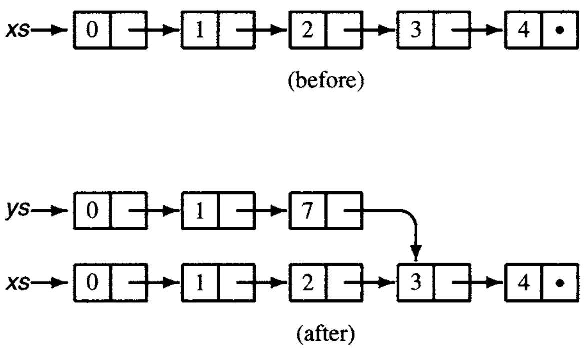

Figure 2.6. Executing *ys* = update(*xs*, 2, 7). Note the sharing between *xs* and *ys*.

Exercise 2.1 Write a function suffixes of type α list $\to$ α list list that takes a list *xs* and returns a list of all the suffixes of *xs* in decreasing order of length. For example,

suffixes [1,2,3,4] = [[1,2,3,4], [2,3,4], [3,4], [4], [ ]]

Show that the resulting list of suffixes can be generated in $O(n)$ time and represented in $O(n)$ space.

2.2 Binary Search Trees

More complicated patterns of sharing are possible when there is more than one pointer field per node. Binary search trees provide a good example of this kind of sharing.

Binary search trees are binary trees with elements stored at the interior nodes in *symmetric order*, meaning that the element at any given node is greater than each element in its left subtree and less than each element in its right subtree. We represent binary search trees in Standard ML with the following type:

```
datatype Tree = E | T of Tree × Elem × Tree
```

where Elem is some fixed type of totally-ordered elements.

Remark Binary search trees are not polymorphic in the type of elements because they cannot accept arbitrary types as elements—only types that are equipped with a total ordering relation are suitable. However, this does not mean that we must re-implement binary search trees for each different element