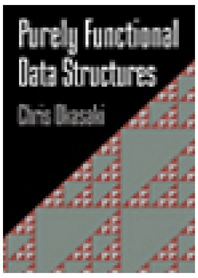


Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

10 - Data-Structural Bootstrapping pp. 141-170

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.011

Cambridge University Press

10
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 these two techniques, along with a third technique for bootstrapping data structures with aggregate elements from data structures with atomic elements.

10.1 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 α List is also α 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 = NIL' | CONS' 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 α Seq so this datatype is non-uniform.

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
  | sizeL (CONS (x, xs)) = 1 + sizeL xs
fun sizeS NIL' = 0
  | sizeS (CONS' (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.

10.1.1 Non-Uniform Recursion and Standard ML

Unfortunately, we usually cannot implement structural decomposition directly in Standard ML. Although Standard ML allows the definition of non-uniform recursive datatypes. the type system disallows most of the interesting functions on such types. For instance, consider the sizeS function on sequences. This function 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 outer sizeS has type α Seq $\rightarrow$ int but the inner sizeS has type $(\alpha \times \alpha)$ Seq $\rightarrow$ int.

It is always 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 rewritten

```
datatype α EP = ELEM of α | PAIR of α EP × α EP
datatype α Seq = NIL′ | CONS′ of α EP × α Seq
```

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

Since it is always possible to convert non-uniform types to uniform types, structural decomposition really refers more to how we think about a datatype than to how it is implemented. For example, consider the revised definition of α Seq above. The α EP type is isomorphic to binary leaf trees, so the revised version of α Seq is equivalent to α Tree list. However, we would tend to think of a list of trees differently than we would think of a sequence of pairs — some algorithms will seem simpler or more natural for one of the representations, and some for the other. We will see some examples of this in the next section.

There are also several pragmatic reasons to prefer the non-uniform definition of α Seq over the uniform one. First, it is more concise; there is one type instead of two, and there is no need to manually insert ELEM and PAIR constructors everywhere. Second, depending on the language implementation, it may be more efficient; there is no need to pattern match against ELEM and PAIR constructors, nor to build run-time representations of these constructors in memory. Third, and most importantly, it allows the type system to catch many more programmer errors. The type in the non-uniform definition ensures that the outermost CONS′ constructor contains a single element, the second a pair of elements, the third a pair of pairs of elements, and so on. The type in the uniform definition ensures neither that pairs are balanced nor that the nesting depth of pairs increases by one per level. Instead, these restrictions must be established by the programmer as system invariants. But if the programmer accidentally violates these invariants — 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. It can always be converted back into legal Standard ML using the kinds of coercions described on the previous page.

10.1.2 Binary Random-Access Lists Revisited

For all of its virtues, the α Seq type that we have been discussing is useless for representing sequences. The problem is that it can only represent sequences with $2^k - 1$ elements. Thinking in terms of numerical representations, the CONS′ constructor gives us a way to write one bits, but there is no way to write zero bits. This is easily corrected by adding another constructor to the type. We also rename the CONS′ constructor to emphasize the analogy to binary numbers.

```
datatype α Seq = NIL | ZERO of (α × α) Seq | ONE of α × (α × α) Seq
```

Now, we can represent the sequence 0...10 as

```
ONE (0, ONE ((1,2), ZERO (ONE ((((3,4),(5,6)),((7,8),(9,10))), NIL))))
```

The size of this sequence is eleven, written `1101` in binary.

The pairs in this type are always balanced. In fact, another way to think of pairs of elements or pairs of pairs of elements, etc., is as complete binary leaf trees. Thus, this type is essentially equivalent to the type of binary random-access lists from Section 9.2.1, but with the invariants of that structure made manifest.

Let's reimplement the functions on binary random-access lists, this time thinking in terms of elements and sequences of pairs rather than lists of complete binary leaf trees. The functions all still run in $O(\log n)$ time, but, as we will see, this new way of thinking yields algorithms that are usually both shorter and easier to understand.

We begin with the cons function. The first two clauses are easy.

```
fun cons (x, NIL) = ONE (x, NIL)
  | cons (x, ZERO ps) = ONE (x, ps)
```

To add a new element to a sequence of the form ONE (*y*, *ps*), we pair the new element with the existing element and add the pair to the sequence of pairs.

```
fun cons (x, ONE (y, ps)) = ZERO (cons ((x, y), ps))
```

This is where we need polymorphic recursion—the outer cons has type

$$\alpha \times \alpha \text{ Seq} \to \alpha \text{ Seq}$$

while the inner cons has type

$$(\alpha \times \alpha) \times (\alpha \times \alpha) \text{ Seq} \to (\alpha \times \alpha) \text{ Seq}.$$

We implement the head and tail functions in terms of an auxiliary function uncons that deconstructs a sequence into its first element and the remaining sequence.

```
fun head xs = let val (x, _) = uncons xs in x end
fun tail xs = let val (_, xs') = uncons xs in xs' end
```

We obtain the uncons function by reading each of the clauses for cons backwards.

```
fun uncons (ONE (x, NIL)) = (x, NIL)
  | uncons (ONE (x, ps)) = (x, ZERO ps)
  | uncons (ZERO ps) = let val ((x, y), ps') = uncons ps
                       in (x, ONE (y, ps')) end
```

Next, consider the lookup function. Given a sequence ONE (*x*, *ps*), we either return *x* or repeat the query on ZERO *ps*.

```
fun lookup (0, ONE (x, ps)) = x
  | lookup (i, ONE (x, ps)) = lookup (i−1, ZERO ps)
```

To lookup the element at index i in a sequence of pairs, we lookup the pair at index $\lfloor i/2 \rfloor$ and then extract the appropriate element from that pair.

```
fun lookup (i, ZERO ps) = let val (x, y) = lookup (i div 2, ps)
                          in if i mod 2 = 0 then x else y end
```

Finally, we turn to the update function. The clauses for the ONE constructor are simply

```
fun update (0, e, ONE (x, ps)) = ONE (e, ps)
  | update (i, e, ONE (x, ps)) = cons (x, update (i−1, e, ZERO ps))
```

However, in trying to update an element in a sequence of pairs, we run into a slight problem. We need to update the pair at index $\lfloor i/2 \rfloor$, but to construct the new pair, we need the other element from the old pair. Thus, we precede the update with a lookup.

```
fun update (i, e, ZERO ps) =
      let val (x, y) = lookup (i div 2, ps)
          val p = if i mod 2 = 0 then (e, y) else (x, e)
      in ZERO (update (i−1, p, ps)) end
```

Exercise 10.1 Prove that this version of update runs in $O(\log^2 n)$ time. ◇

To restore the $O(\log n)$ bound on the update function, we must eliminate the call to lookup. But then how do we get the other element from which to construct the new pair? Well, if we cannot bring Mohammed to the mountain, then we must send the mountain to Mohammed. That is, instead of fetching the old pair and constructing the new pair locally, we send a function to construct the new pair from the old pair wherever the old pair is found. We use a helper function fupdate that takes a function to apply to the ith element of a sequence. Then update is simply

```
fun update (i, y, xs) = fupdate (fn x ⇒ y, i, xs)
```

The key step in fupdate is promoting a function *f* on elements to a function *f'* that takes a pair and applies *f* to either the first or second element of the pair, depending on the parity of *i*.

```
fun f' (x, y) = if i mod 2 = 0 then (f x, y) else (x, f y)
```

Given this definition, the rest of fupdate is straightforward.

```
fun fupdate (f, 0, ONE (x, ps)) = ONE (f x, ps)
  | fupdate (f, i, ONE (x, ps)) = cons (x, fupdate (f, i-1, ZERO ps))
  | fupdate (f, i, ZERO ps) =
      let fun f' (x, y) = if i mod 2 = 0 then (f x, y) else (x, f y)
      in ZERO (fupdate (f', i div 2, ps)) end
```

The complete implementation is shown in Figure 10.1.

Comparing Figures 10.1 and 9.6, we see that this implementation is significantly more concise and that the individual functions are significantly simpler, with the possible exception of update. (And even update is simpler if you are comfortable with higher-order functions.) These benefits arise from recasting the data structure as a non-uniform type that directly reflects the desired invariants.

Exercise 10.2 Reimplement AltBinaryRandomAccessList so that cons, head, and tail all run in $O(1)$ amortized time, using the type

```
datatype α RList =
        NIL
      | ONE of α × (α × α) RList susp
      | TWO of α × α × (α × α) RList susp
      | THREE of α × α × α × (α × α) RList susp
```

10.1.3 Bootstrapped Queues

Consider the use of ++ in the banker's queues of Section 6.3.2. During a rotation, the front stream *f* is replaced by *f* ++ reverse *r*. After a series of rotations,

```
structure AltBinaryRandomAccessList : RANDOMACCESSLIST =
  (* assumes polymorphic recursion! *)
struct
  datatype α RList =
       NIL | ZERO of (α × α) RList | ONE of α × (α × α) RList

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

  fun cons (x, NIL) = ONE (x, NIL)
    | cons (x, ZERO ps) = ONE (x, ps)
    | cons (x, ONE (y, ps)) = ZERO (cons ((x, y), ps))

  fun uncons NIL = raise EMPTY
    | uncons (ONE (x, NIL)) = (x, NIL)
    | uncons (ONE (x, ps)) = (x, ZERO ps)
    | uncons (ZERO ps) = let val ((x, y), ps') = uncons ps
                         in (x, ONE (y, ps')) end

  fun head xs = let val (x, _) = uncons xs in x end
  fun tail xs = let val (_, xs') = uncons xs in xs' end

  fun lookup (i, NIL) = raise SUBSCRIPT
    | lookup (0, ONE (x, ps)) = x
    | lookup (i, ONE (x, ps)) = lookup (i−1, ZERO ps)
    | lookup (i, ZERO ps) = let val (x, y) = lookup (i div 2, ps)
                            in if i mod 2 = 0 then x else y end

  fun fupdate (f, i, NIL) = raise SUBSCRIPT
    | fupdate (f, 0, ONE (x, ps)) = ONE (f x, ps)
    | fupdate (f, i, ONE (x, ps)) = cons (x, fupdate (f, i−1, ZERO ps))
    | fupdate (f, i, ZERO ps) =
        let fun f' (x, y) = if i mod 2 = 0 then (f x, y) else (x, f y)
        in ZERO (fupdate (f', i div 2, ps)) end

  fun update (i, y, xs) = fupdate (fn x ⇒ y, i, xs)
end
```

Figure 10.1. An alternative implementation of binary random-access lists.

the front stream has the form

$$((f \mathbin{+\!\!+} \text{reverse } r_1) \mathbin{+\!\!+} \text{reverse } r_2) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \text{reverse } r_k$$

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 $\mathbin{+\!\!+}$), and the elements of r_i will be processed $k - i + 1$ times (once by reverse and once for each following $\mathbin{+\!\!+}$). 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 the front stream has the described form, we decompose it into two parts: *f* and the collection $m = \{\text{reverse } r_1, \ldots, \text{reverse } r_k\}$. Then we can represent *f* as a list and each reverse r_i as a suspended list. We also change the rear stream *r* to a list. These changes eliminate the vast majority of suspensions and avoid 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 represent empty queues with a special constructor.† The new representation is therefore

```
datatype α Queue =
      E | Q of int × α list × α list susp Queue × int × α list
```

The first integer, *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). The second integer, *lenr*, is as usual the length of *r*. The usual balance invariant becomes *lenr* ≤ *lenfm*. In addition, we require that *f* 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 (E, x) = Q (1, [x], E, 0, [ ])
  | snoc (Q (lenfm, f, m, lenr, r), x) = checkQ (lenfm, f, m, lenr+1, x :: r)
fun head (Q (lenfm, x :: f', m, lenr, r)) = x
fun tail (Q (lenfm, x :: f', m, lenr, r)) = checkQ (lenfm−1, f', m, lenr, r)
```

The interesting cases are in the helper function checkQ. If *r* is too long, checkQ creates a suspension to reverse *r* and adds the suspension to *m*. After checking the length of *r*, checkQ invokes a second 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 checkF (lenfm, [ ], E, lenr, r) = E
  | checkF (lenfm, [ ], m, lenr, r) =
        Q (lenfm, force (head m), tail m, lenr, r)
  | checkF q = Q q
fun checkQ (q as (lenfm, f, m, lenr, r)) =
        if lenr ≤ lenfm then checkF q
        else checkF (lenfm+lenr, f, snoc (m, $rev r), 0, [ ])
```

† 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 =
        E | Q of int × α list × α list susp Queue × int × α list

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

  fun checkQ (q as (lenfm, f, m, lenr, r)) =
        if lenr ≤ lenfm then checkF q
        else checkF (lenfm+lenr, f, snoc (m, $rev r), 0, [ ])
  and checkF (lenfm, [ ], E, lenr, r) = E
    | checkF (lenfm, [ ], m, lenr, r) =
        Q (lenfm, force (head m), tail m, lenr, r)
    | checkF q = Q q

  and snoc (E, x) = Q (1, [x], E, 0, [ ])
    | snoc (Q (lenfm, f, m, lenr, r), x) = checkQ (lenfm, f, m, lenr+1, x :: r)
  and head E = raise EMPTY
    | head (Q (lenfm, x :: f', m, lenr, r)) = x
  and tail E = raise EMPTY
    | tail (Q (lenfm, x :: f', m, lenr, r)) = checkQ (lenfm−1, f', m, lenr, r)
end
```

Figure 10.2. Bootstrapped queues based on structural decomposition.

Note that checkQ and checkF call snoc and tail, which in turn call checkQ. These functions must therefore all be defined mutually recursively. The complete implementation appears in Figure 10.2.

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 times of snoc and tail are not constant! Note that snoc calls checkQ, 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 depth of the entire queue is at most $O(\log^* n)$. 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 (from checkQ) and tail (from checkF). Note that, when this happens, tail is called on the result of the snoc. Now, the snoc might recursively call itself and the tail might again recursively call both snoc and tail. However, from Exercise 10.3, we know that the snoc and tail never both recursively call snoc. Therefore, both snoc and tail are 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 four simply as lists, then queues with fewer than about four billion elements have at most three levels.

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.

Exercise 10.3 Consider the expression tail (snoc (q, x)). Show that the calls to tail and snoc will never both recursively call snoc.

Exercise 10.4 Implement these queues without polymorphic recursion using the types

```
datatype α EL = ELEM of α | LIST of α EL list susp
datatype α Queue = E | Q of int × α EL list × α Queue × int × α EL list
```

Exercise 10.5 Another way to eliminate the need for polymorphic recursion is to represent the middle using some other implementation of queues. Then the type of bootstrapped queues is

```
datatype α Queue =
      E | Q of int × α list × α list susp PrimQ.Queue × int × α list
```

where PrimQ is the other implementation of queues.

(a) Implement this variation of bootstrapped queues as a functor of the form

```
functor BootstrapQueue (PrimQ : QUEUE) : QUEUE = ...
```

(b) Prove that if PrimQ is instantiated to some implementation of real-time queues, then all operations on the resulting bootstrapped queues take $O(1)$ amortized time.

10.2 Structural Abstraction

The second class 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 α C is a collection type with elements of type α , and that this type supports an efficient insert function, with signature

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

Call α C the *primitive* type. From this type, we wish to derive a new datatype, α B, called the *bootstrapped* type, such that α 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 subscript to distinguish functions on the bootstrapped type from functions on the primitive type.) The bootstrapped type should also 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 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, things are not quite that simple. Based on the above description, we might attempt to define α B as

datatype α B = B **of** (α B) C

This definition can be viewed as specifying an isomorphism

$$\alpha\ \mathsf{B} \cong (\alpha\ \mathsf{B})\ \mathsf{C}$$

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

$$\alpha\ \mathsf{B} \cong (\alpha\ \mathsf{B})\ \mathsf{C} \cong ((\alpha\ \mathsf{B})\ \mathsf{C})\ \mathsf{C} \cong \cdots \cong ((\cdots \mathsf{C})\ \mathsf{C})\ \mathsf{C}$$

The type α has disappeared, so there is no way to actually store an element in this collection! We can solve this problem 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 unitB 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 = E | B of α × (α B) C
```

Remark Actually, we 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 = E | NE of B+
```

Unfortunately, definitions of this form lead to more verbose code, so we stick with the earlier less precise, but more concise, definition. ◇

Now, we can refine the above templates for insert$_B$ and join$_B$ as

```
fun insertB (x, E) = B (x, empty)
  | insertB (x, B (y, c)) = B (x, insert (unitB y, c))

fun joinB (b, E) = b
  | joinB (E, b) = b
  | joinB (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.

```
signature CATENABLELIST =
sig
  type α Cat

  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 10.3. Signature for catenable lists.

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 primitive collection of type (α B) C, which must then be converted into a bootstrapped collection of type α 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.

10.2.1 Lists With Efficient Catenation

The first data structure we will implement using structural abstraction is catenable lists, as specified by the signature in Figure 10.3. 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* ++ 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

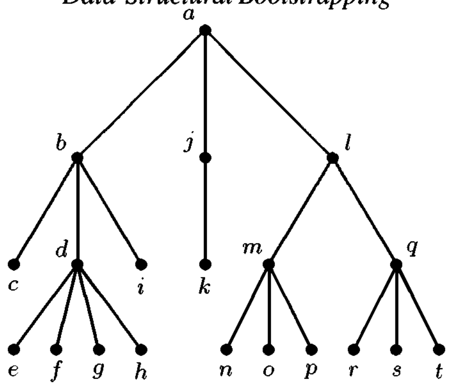


Figure 10.4. A tree representing the list $a \dots t$.

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 α Cat = E | C of α × α 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 list to be easily accessible, we 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 10.4.

Now, head is simply

```
fun head (C (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.

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

The helper function link adds its second argument to the child queue of its first argument.

```
fun link (C (x, q), ys) = C (x, Q.snoc (q, ys))
```

cons and snoc simply call ++.

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

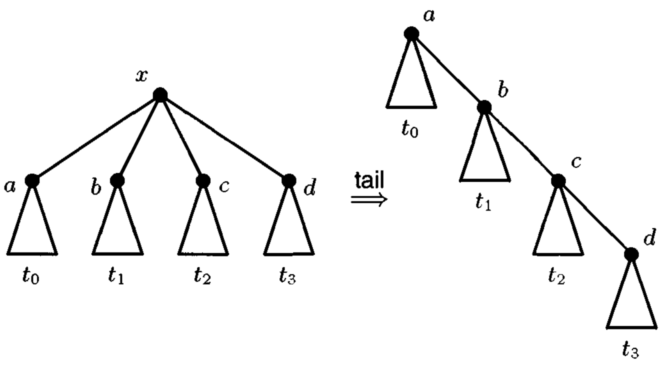


Figure 10.5. Illustration of the tail operation.

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 E. Otherwise we link all the children together.

```
fun tail (C (x, q)) = if Q.isEmpty q then E 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 10.5, will duplicate the least 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. We hope to reduce this to $O(1)$ amortized time, but to achieve this in the face of persistence, we must somehow incorporate lazy evaluation into the design. 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, the queues must contain tree suspensions rather than trees, so we redefine the type as

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

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

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

  fun link (C (x, q), s) = C (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 ++ E = xs
    | E ++ xs = xs
    | xs ++ ys = link (xs, $ys)
  fun cons (x, xs) = C (x, Q.empty) ++ xs
  fun snoc (xs, x) = xs ++ C (x, Q.empty)

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

Figure 10.6. Catenable lists.

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

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

The complete implementation is shown in Figure 10.6.

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 maintain this invariant by incrementing the number of debits on a node only when we also increment its degree.

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

$$D_t(i) \leq i + \mathit{depth}_t(i)$$

where $\mathit{depth}_t(i)$ is the length of the path from the root of t 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 to become the new root.

Theorem 10.1 *$+\!\!+$ 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 $i < n$

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

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

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

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

(tail) Let $t' =$ 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 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 10.5) while the cumulative debits of each node in t_j increase by j. Thus,

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

Discharging the first two debits restores the invariant, bringing the total to 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.

Exercise 10.6 Write a function flatten of type α Cat list $\rightarrow$ α Cat that catenates all the elements in a list of catenable lists. Show that your function runs in $O(1 + e)$ amortized time, where e is the number of empty catenable lists in the list.

10.2.2 Heaps With Efficient Merging

Next, we use structural abstraction on heaps to obtain an efficient merge operation.

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 9.3.2 are one such implementation; the scheduled binomial heaps of Section 7.3 are another. 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 = E | H of α × (α Heap) PrimH.Heap
```

where PrimH is the implementation of primitive heaps. The element stored at any given H 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). We can think of this type as a multiary tree in which the children of each node are stored in primitive heaps.

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

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

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

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

Finally, we consider deleteMin, defined as

```
fun deleteMin (H (x, p)) =
      if PrimH.isEmpty p then E
      else let val (H (y, p1)) = PrimH.findMin p
               val p2 = PrimH.deleteMin p
           in H (y, PrimH.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, findMin runs in $O(1)$ worst-case time regardless of the underlying implementation of primitive heaps. insert and merge depend only on PrimH.insert. Since we have assumed that PrimH.insert runs in $O(1)$ worst-case time, so do insert and merge. Finally, deleteMin calls PrimH.findMin, PrimH.deleteMin, and PrimH.merge. Since each of these runs in $O(\log n)$ worst-case time, so does deleteMin.

Remark We can also bootstrap heaps with amortized bounds. For example, bootstrapping the lazy binomial heaps of Section 6.4.1 produces an implementation that supports findMin in $O(1)$ worst-case time, insert and merge in $O(1)$ amortized time, and deleteMin in $O(\log n)$ amortized time. ◇

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. 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 (Element : ORDERED)
                              : HEAP where type Elem.T = Element.T)
                  (Element : ORDERED) : HEAP = ...
```

The Bootstrap functor takes the MakeH functor as an argument. The MakeH functor takes the ORDERED structure Element, which defines the element type and the comparison function, and returns a HEAP structure. Given MakeH, Bootstrap returns a functor that takes an ORDERED structure Element and returns a HEAP structure.

Remark The **where type** 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 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 BootstrappedElem 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 BootstrappedElem =
  struct
    datatype T = E | H of Elem.T × PrimH.Heap
    fun leq (H (x, _), H (y, _)) = Elem.leq (x, y)
    ... similar definitions for eq and lt...
  end
and PrimH = MakeH (BootstrappedElem)
```

```
functor Bootstrap (functor MakeH (Element : ORDERED)
                                    : HEAP where type Elem.T = Element.T)
                        (Element : ORDERED) : HEAP =
struct
  structure Elem = Element

  (* recursive structures not supported in Standard ML! *)
  structure rec BootstrappedElem =
    struct
      datatype T = E | H of Elem.T × PrimH.Heap
      fun leq (H (x, _), H (y, _)) = Elem.leq (x, y)
      ...similar definitions for eq and lt...
    end
  and PrimH = MakeH (BootstrappedElem)

  open BootstrappedElem   (* expose E and H constructors *)

  type Heap = BootstrappedElem.T

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

  fun merge (E, h) = h
    | merge (h, E) = h
    | merge (h1 as H (x, p1), h2 as H (y, p2)) =
        if Elem.leq (x, y) then H (x, PrimH.insert (h2, p1))
        else H (y, PrimH.insert (h1, p2))
  fun insert (x, h) = merge (H (x, PrimH.empty), h)

  fun findMin E = raise EMPTY
    | findMin (H (x, _)) = x
  fun deleteMin E = raise EMPTY
    | deleteMin (H (x, p)) =
        if PrimH.isEmpty p then E
        else let val (H (y, p1)) = PrimH.findMin p
                 val p2 = PrimH.deleteMin p
             in H (y, PrimH.merge (p1, p2)) end
end
```

Figure 10.7. Bootstrapped heaps.

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 10.7.

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 9.3.2, are well-behaved in this respect, and the recursive pattern embodied by the Bootstrap

```
signature HEAPWITHINFO =
sig
  structure Priority : ORDERED

  type α Heap

  val empty     : α Heap
  val isEmpty   : α Heap → bool

  val insert    : Priority.T × α × α Heap → α Heap
  val merge     : α Heap × α Heap → α Heap

  val findMin   : α Heap → Priority.T × α
  val deleteMin : α Heap → α Heap
      (* findMin and deleteMin raise EMPTY if heap is empty *)
end
```

Figure 10.8. Alternate signature for heaps.

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 or LazyBinomialHeap, and then eliminating BootstrappedElem and PrimH as separate structures. The recursion on structures then reduces to recursion on datatypes, which is supported by Standard ML.

Exercise 10.7 Inlining the LazyBinomialHeap functor of Section 6.4.1 as described above yields the types

```
datatype Tree = Node of int × Heap × Tree list
datatype Heap = E | NE of Elem.T × Tree list susp
```

Complete this implementation of bootstrapped heaps.

Exercise 10.8 Elements in a heap frequently contain other information besides the priority. For these kinds of elements, it is often more convenient to use heaps that separate the priority from the rest of the element. Figure 10.8 gives an alternate signature for this kind of heap.

(a) Adapt either LazyBinomialHeap or SkewBinomialHeap to this new signature.
(b) Rewrite the Bootstrap functor as

```
functor Bootstrap (PrimH : HEAPWITHINFO) : HEAPWITHINFO = ...
```

You will need neither higher-order functors nor recursive structures.

```
signature FINITEMAP =
sig
  type Key
  type α Map

  val empty  : α Map
  val bind   : Key × α × α Map → α Map
  val lookup : Key × α Map → α  (* raise NOTFOUND if key is not found *)
end
```

Figure 10.9. Signature for finite maps.

10.3 Bootstrapping To Aggregate Types

We have now seen several examples where collections of aggregate data (e.g., heaps of heaps) were useful in implementing collections of non-aggregate data (e.g., heaps of elements). However, collections of aggregate data are often useful in their own right. As a simple example, strings (i.e., sequences of characters) are frequently used as the element type of sets or the key type of finite maps. In this section, we illustrate bootstrapping finite maps defined over some simple type to finite maps defined over lists or even trees of that type.

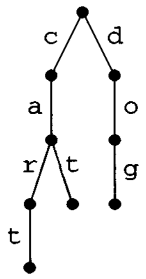
10.3.1 Tries

Binary search trees work well when comparisons on the key or element type are cheap. This is true for simple types like integers or characters, but may not be true for aggregate types like strings. For example, consider representing a phone book using a binary search tree. A query for `"Smith, Joan"` might involve multiple comparisons against entries for `"Smith, John"`, each of which inspects the first ten characters of both strings before returning.

A better solution for aggregate types such as strings is to choose a representation that takes advantage of the structure of that type. One such representation is *tries*, also known as a *digital search trees*. In this section, we will use tries to implement the FINITEMAP abstraction, shown in Figure 10.9.

In the following discussion, we assume that keys are strings, represented as lists of characters. We will often refer to characters as the *base type*. The ideas can easily be adapted to other sequence representations and other base types.

Now, a trie is a multiway tree where each edge is labelled with a character. Edges leaving the root of a trie represent the first character of a string, edges leaving children of the root represent the second character, and so on. To find the node associated with a given string, start at the root and follow the edges labelled by the characters of the string, in order. For example, the trie representing the strings `"cat"`, `"dog"`, `"car"`, and `"cart"` might be drawn



Note that entering a string into a trie also enters all the prefixes of that string into the trie. Only some of these prefixes constitute valid entries. In this example, `"c"`, `"ca"`, and `"car"` are all prefixes of `"cart"` but only `"car"` is valid. We therefore mark each node as either valid or invalid. For finite maps, we accomplish this with the built-in option datatype

```
datatype α option = NONE | SOME of α
```

If a given node is invalid, we mark it with NONE. If the node is valid, and the corresponding string is mapped to the value *x*, then we mark it with SOME *x*.

The critical remaining question is how to represent the edges leaving a node. Ordinarily, we would represent the children of a multiway node as a list of trees, but here we also need to represent the edge labels. Depending on the choice of base type and the expected density of the trie, we might represent the edges leaving a node as a vector, an association list, a binary search tree, or even, if the base type is itself a list or a string, another trie! But all of these are just finite maps from edges labels to tries. We abstract away from the particular representation of these edge maps by assuming that we are given a structure M implementing finite maps over the base type. Then the representation of a trie is simply

```
datatype α Map = TRIE of α option × α Map M.Map
```

The empty trie is represented by a single invalid node with no children.

```
val empty = TRIE (NONE, M.empty)
```

To lookup a string, we lookup each character in the appropriate edge map. When we reach the final node, we check whether it is valid or invalid.

```
fun lookup ([ ], TRIE (NONE, m)) = raise NOTFOUND
  | lookup ([ ], TRIE (SOME x, m)) = x
  | lookup (k :: ks, TRIE (v, m)) = lookup (ks, M.lookup (k, m))
```

```
functor Trie (M : FINITEMAP) : FINITEMAP =
struct
  type Key = M.Key list

  datatype α Map = TRIE of α option × α Map M.Map

  val empty = TRIE (NONE, M.empty)

  fun lookup ([ ], TRIE (NONE, m)) = raise NOTFOUND
    | lookup ([ ], TRIE (SOME x, m)) = x
    | lookup (k :: ks, TRIE (v, m)) = lookup (ks, M.lookup (k, m))

  fun bind ([ ], x, TRIE (_, m)) = TRIE (SOME x, m)
    | bind (k :: ks, x, TRIE (v, m)) =
        let val t = M.lookup (k, m) handle NOTFOUND ⇒ empty
            val t' = bind (ks, x, t)
        in TRIE (v, M.bind (k, t', m)) end
end
```

Figure 10.10. A simple implementation of tries.

Note that if a given string is not in the trie, then we may not even reach the final node. For example, if we were to lookup `"dark"` in our example trie, then the lookup of `d` would succeed but the lookup of `a` would fail. In that case, M.lookup would raise the NOTFOUND exception. This is also the appropriate response for lookup so we simply propagate the exception.

Remark This property of unsuccessful searches explains why tries can be even faster than hashing. An unsuccessful search in a trie might exit after examining only a few characters, whereas an unsuccessful search in a hash table must examine the entire string just to compute the hash function! ◇

The bind function is similar to the lookup function, except that we do not allow the call to M.lookup to fail. We force it to succeed by substituting the empty node whenever it raises the NOTFOUND exception.

```
fun bind ([ ], x, TRIE (_, m)) = TRIE (SOME x, m)
  | bind (k :: ks, x, TRIE (v, m)) =
      let val t = M.lookup (k, m) handle NOTFOUND ⇒ empty
          val t' = bind (ks, x, t)
      in TRIE (v, M.bind (k, t', m)) end
```

The complete implementation is shown in Figure 10.10.

Exercise 10.9 Very often, the set of keys to be stored in a trie has the property that no key is a proper prefix of another. For example, the keys might all be the same length, or the keys might all end in a unique character that occurs in no other position. Reimplement tries under this assumption, using the type

```
datatype α Map = Entry of α | TRIE of α Map M.Map
```

Exercise 10.10 Tries frequently contain long paths of nodes that each have only a single child. A common optimization is to collapse such paths into a single node. We accomplish this by storing with every node a substring that is the longest common prefix of every key in that subtrie. The type of tries is then

```
datatype α Map = TRIE of M.Key list × α option × α Map M.Map
```

Reimplement tries using this type. You should maintain the invariant that no node is both invalid and an only child. You may assume that the structure M provides an isEmpty function.

Exercise 10.11 (Schwenke [Sch97]) Another common data structure that involves multiple layers of finite maps is the *hash table*. Complete the following implementation of abstract hash tables.

```
functor HashTable (structure Approx : FINITEMAP
                   structure Exact  : FINITEMAP
                   val hash : Exact.Key → Approx.Key) : FINITEMAP =
struct
   type Key = Exact.Key
   type α Map = α Exact.Map Approx.Map
   ...
   fun lookup (k, m) = Exact.lookup (k, Approx.lookup (hash k, m))
   ...
end
```

The advantage of this representation is that Approx can use an efficient key type (such as integers) and Exact can use a trivial implementation (such as association lists).

10.3.2 Generalized Tries

The idea of tries can also be generalized to other aggregate types, such as trees [CM95]. First, consider how the edge maps of the previous section reflect the type of the cons constructor. The edge maps are represented by the type α Map M.Map. The outer map indexes the first field of the cons constructor and the inner map indexes the second field of the cons constructor. Looking up the head of a cons cell in the outer map returns the inner map in which to lookup the tail of the cons cell.

We can generalize this scheme to binary trees, which have three fields, by adding a third map layer. For example, given binary trees of type

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

we can represent the edge maps in tries over these trees as α Map Map M.Map. The outer map indexes the first field of the T constructor, the middle map indexes the second field, and the inner map indexes the third field. Looking up the element at a given node in the outer map returns the middle map in which to lookup the left subtree. That lookup, in turn, returns the inner map in which to lookup the right subtree.

More formally, we represent tries over binary trees as

```
datatype α Map = TRIE of α option × α Map Map M.Map
```

Notice that this is a non-uniform recursive type, so we will need polymorphic recursion in the functions over this type.

Now, the lookup function performs three lookups for each T constructor, corresponding to the three fields of the constructor. When it reaches the final node, it checks whether the node is valid.

```
fun lookup (E, TRIE (NONE, m)) = raise NOTFOUND
  | lookup (E, TRIE (SOME x, m)) = x
  | lookup (T (k, a, b), TRIE (v, m)) =
      lookup (b, lookup (a, M.lookup (k, m)))
```

The bind function is similar. It is shown in Figure 10.11, which summarizes the entire implementation of tries over trees.

Exercise 10.12 Reimplement the TrieOfTrees functor without polymorphic recursion using the types

```
datatype α Map = TRIE of α EM option × α Map M.Map
and α EM = ELEM of α | MAP of α Map
```

Exercise 10.13 Implement tries whose keys are multiway trees of type

```
datatype α Tree = T of α × α Tree list
```

◇

With these examples, we can generalize the notion of tries to any recursive type involving products and sums. We need only a few simple rules about how to construct a finite map for a structured type given finite maps for its component parts. Let α Map_τ be the type of finite maps over type τ.

For products, we already know what to do; to lookup a pair in a trie, we first lookup the first element of the pair and get back a map in which to lookup the second element. Thus,

$$\tau = \tau_1 \times \tau_2 \Rightarrow \alpha\ \text{Map}_\tau = \alpha\ \text{Map}_{\tau_2}\ \text{Map}_{\tau_1}$$

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

functor TrieOfTrees (M : FINITEMAP) : FINITEMAP =
  (* assumes polymorphic recursion! *)
struct
  type Key = M.Key Tree

  datatype α Map = TRIE of α option × α Map Map M.Map

  val empty = TRIE (NONE, M.empty)

  fun lookup (E, TRIE (NONE, m)) = raise NOTFOUND
    | lookup (E, TRIE (SOME x, m)) = x
    | lookup (T (k, a, b), TRIE (v, m)) =
        lookup (b, lookup (a, M.lookup (k, m)))

  fun bind (E, x, TRIE (_, m)) = TRIE (SOME x, m)
    | bind (T (k, a, b), x, TRIE (v, m)) =
        let val tt = M.lookup (k, m) handle NOTFOUND ⇒ empty
            val t = lookup (a, tt) handle NOTFOUND ⇒ empty
            val t' = bind (b, x, t)
            val tt' = bind (a, t', tt)
        in TRIE (v, M.bind (k, tt', m)) end
end
```

Figure 10.11. Generalized Tries.

Now, what about sums? Recall the types of trees and tries over trees:

```
datatype α Tree = E | T of α × α Tree × α Tree
datatype α Map = TRIE of α option × α Map Map M.Map
```

Obviously the type α Map Map M.Map corresponds to the T constructor, but what corresponds to the E constructor? Well, the α option type is really nothing more or less than a very efficient implementation of finite maps over the unit type, which is essentially equivalent to the missing body of the E constructor. From this, we infer the general rule for sums:

$$\tau = \tau_1 + \tau_2 \Rightarrow \alpha\ \mathsf{Map}_\tau = \alpha\ \mathsf{Map}_{\tau_1} \times \alpha\ \mathsf{Map}_{\tau_2}$$

Exercise 10.14 Complete the following functors that implement the above rules for products and sums.

```
functor ProductMap (M₁ : FINITEMAP) (M₂ : FINITEMAP) : FINITEMAP =
struct
    type Key = M₁.Key × M₂.Key
    ...
end
```

```
datatype (α, β) Sum = LEFT of α | RIGHT of β
functor SumMap (M₁ : FINITEMAP) (M₂ : FINITEMAP) : FINITEMAP =
struct
    type Key = (M₁.Key, M₂.Key) Sum
    ...
end
```

Exercise 10.15 Given a structure M that implements finite maps over the type Id of identifiers, implement tries over the type Exp of lambda expressions, where

```
datatype Exp = VAR of Id | LAM of Id × Exp | APP of Exp × Exp
```

You may find it helpful to extend the type of tries with a separate constructor for the empty map.

10.4 Chapter Notes

Data-Structural Bootstrapping Buchsbaum and colleagues 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 of the head or tail functions.

Myers [Mye84] described a representation based on AVL trees that supports all relevant list functions in $O(\log n)$ time. Tarjan and colleagues [DST94, BT95, KT95] investigated a series of sub-logarithmic implementations, culminating in a implementation that supports catenation and all other usual list functions in $O(1)$ worst-case time. The implementation of catenable lists in Section 10.2.1 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 (see Section 6.5), 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 8.4.3. However, such an implementation would be both complicated and slow. Brodal and Okasaki simplify this implementation in [BO96], using skew binomial heaps (Section 9.3.2) and structural abstraction (Section 10.2.2).

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 sidesteps this problem by allowing polymorphic recursion whenever the programmer provides an explicit type signature.