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

2 - Persistence pp. 7-16

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.003

Cambridge University Press

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.

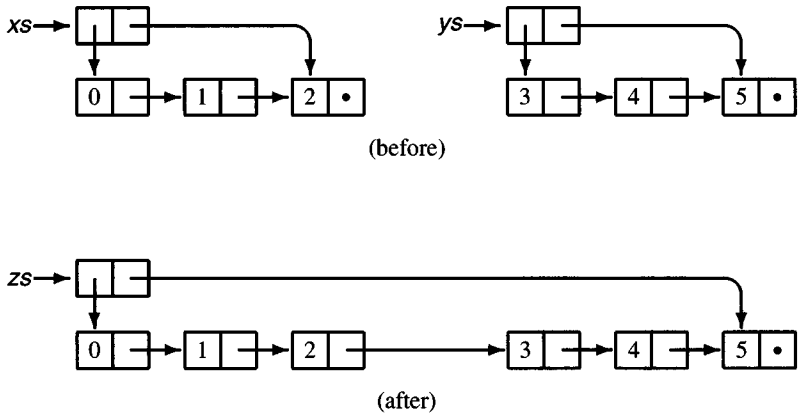

Figure 2.4. Executing *zs* = *xs* ++ *ys* in an imperative setting. Note that this operation destroys the argument lists, *xs* and *ys*.

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-

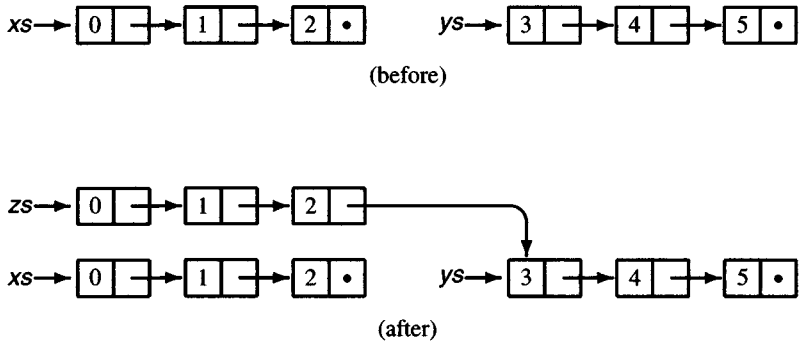

Figure 2.5. Executing *zs* = *xs* ++ *ys* in a functional setting. Notice that the argument lists, *xs* and *ys*, are unaffected by the operation.

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.

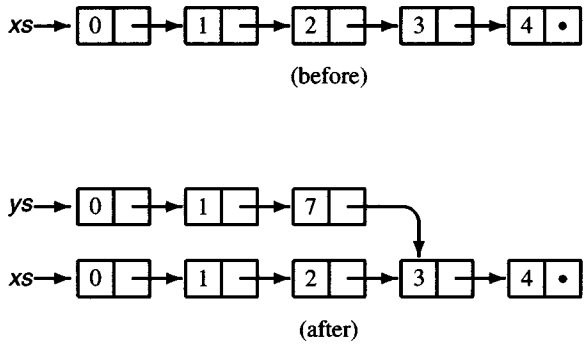

Figure 2.6. Executing *ys* = update(*xs*, 2, 7). Note the sharing between *xs* and *ys*.

Exercise 2.1 Write a function suffixes of type α list $\rightarrow$ 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 $\times$ Elem $\times$ 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

```
signature SET =
sig
  type Elem
  type Set
  val empty  : Set
  val insert : Elem × Set → Set
  val member : Elem × Set → bool
end
```

Figure 2.7. Signature for sets.

type. Instead, we make the type of elements and its attendant comparison functions parameters of the *functor* that implements binary search trees (see Figure 2.9). ◇

We will use this representation to implement sets. However, it can easily be adapted to support other abstractions (e.g., finite maps) or fancier functions (e.g., find the ith smallest element) by augmenting the T constructor with extra fields.

Figure 2.7 describes a minimal signature for sets. This signature contains a value for the empty set and functions for inserting a new element and testing for membership. A more realistic implementation would probably include many additional functions, such as deleting an element or enumerating all elements.

The member function searches a tree by comparing the query element with the element at the root. If the query element is smaller than the root element, then we recursively search the left subtree. If the query element is larger than the root element, then we recursively search the right subtree. Otherwise the query element is equal to the element at the root, so we return true. If we ever reach the empty node, then the query element is not an element of the set, so we return false. This strategy is implemented as follows:

```
fun member (x, E) = false
  | member (x, T (a, y, b)) =
      if x < y then member (x, a)
      else if x > y then member (x, b)
      else true
```

Remark For simplicity, we have assumed that the comparison functions are named $<$ and $>$. However, when these functions are passed as parameters to a functor, as they will be in Figure 2.9, it is often more convenient to use names such as lt or leq, and reserve $<$ and $>$ for comparing integers and other primitive types. ◇

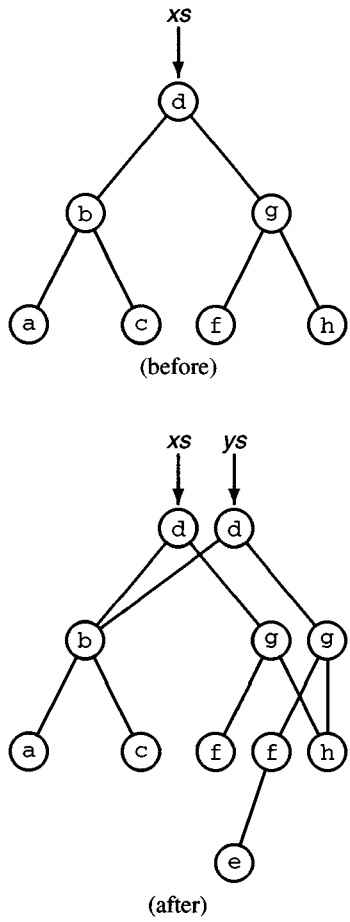

Figure 2.8. Execution of *ys* = insert ("e", *xs*). Once again, notice the sharing between *xs* and *ys*.

The insert function searches the tree using the same strategy as member, except that it copies every node along the way. When it finally reaches an empty node, it replaces the empty node with a node containing the new element.

```
fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
      if x < y then T (insert (x, a), y, b)
      else if x > y then T (a, y, insert (x, b))
      else s
```

Figure 2.8 illustrates a typical insertion. Every node that is copied shares one

```
signature ORDERED =
  (* a totally ordered type and its comparison functions *)
sig
  type T
  val eq : T × T → bool
  val lt : T × T → bool
  val leq : T × T → bool
end

functor UnbalancedSet (Element : ORDERED) : SET =
struct
  type Elem = Element.T
  datatype Tree = E | T of Tree × Elem × Tree
  type Set = Tree

  val empty = E

  fun member (x, E) = false
    | member (x, T (a, y, b)) =
        if Element.lt (x, y) then member (x, a)
        else if Element.lt (y, x) then member (x, b)
        else true

  fun insert (x, E) = T (E, x, E)
    | insert (x, s as T (a, y, b)) =
        if Element.lt (x, y) then T (insert (x, a), y, b)
        else if Element.lt (y, x) then T (a, y, insert (x, b))
        else s
end
```

Figure 2.9. Implementation of binary search trees as a Standard ML functor.

subtree with the original tree—the subtree that was not on the search path. For most trees, this search path contains only a tiny fraction of the nodes in the tree. The vast majority of nodes reside in the shared subtrees.

Figure 2.9 shows how binary search trees might be implemented as a Standard ML functor. This functor takes the element type and its associated comparison functions as parameters. Because these same parameters will often be used by other functors as well (see, for example, Exercise 2.6), we package them in a structure matching the ORDERED signature.

Exercise 2.2 (Andersson [And91]) In the worst case, member performs approximately $2d$ comparisons, where d is the depth of the tree. Rewrite member to take no more than $d + 1$ comparisons by keeping track of a candidate element that *might* be equal to the query element (say, the last element for which

$<$ returned false or $\leq$ returned true) and checking for equality only when you hit the bottom of the tree.

Exercise 2.3 Inserting an existing element into a binary search tree copies the entire search path even though the copied nodes are indistinguishable from the originals. Rewrite insert using exceptions to avoid this copying. Establish only one handler per insertion rather than one handler per iteration.

Exercise 2.4 Combine the ideas of the previous two exercises to obtain a version of insert that performs no unnecessary copying and uses no more than $d + 1$ comparisons.

Exercise 2.5 Sharing can also be useful within a single object, not just between objects. For example, if the two subtrees of a given node are identical, then they can be represented by the same tree.

(a) Using this idea, write a function complete of type Elem $\times$ int $\rightarrow$ Tree where complete (*x*, *d*) creates a complete binary tree of depth *d* with *x* stored in every node. (Of course, this function makes no sense for the set abstraction, but it can be useful as an auxiliary function for other abstractions, such as bags.) This function should run in $O(d)$ time.

(b) Extend this function to create balanced trees of arbitrary size. These trees will not always be complete binary trees, but should be as balanced as possible: for any given node, the two subtrees should differ in size by at most one. This function should run in $O(\log n)$ time. (Hint: use a helper function create2 that, given a size *m*, creates a pair of trees, one of size *m* and one of size *m*+1.)

Exercise 2.6 Adapt the UnbalancedSet functor to support finite maps rather than sets. Figure 2.10 gives a minimal signature for finite maps. (Note that the NOTFOUND exception is not predefined in Standard ML—you will have to define it yourself. Although this exception could be made part of the FINITEMAP signature, with every implementation defining its own NOTFOUND exception, it is convenient for all finite maps to use the same exception.)

2.3 Chapter Notes

Myers [Mye82, Mye84] used copying and sharing to implement persistent binary search trees (in his case, AVL trees). Sarnak and Tarjan [ST86a] coined the term *path copying* for the general technique of implementing persistent

```
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 2.10. Signature for finite maps.

data structures by copying all affected nodes. Other general techniques for implementing persistent data structures have been proposed by Driscoll, Sarnak, Sleator, and Tarjan [DSST89] and Dietz [Die89], but these techniques are not purely functional.