Abstract

This paper gives an introduction and short overview of various topics related to purely functional data structures. First the concept of persistent and purely functional data structures is introduced, followed by some examples of basic data structures, relevant in purely functional setting, such as lists, queues and trees. Various techniques for designing more efficient purely functional data structures based on lazy evaluation are then described. Two notable general-purpose functional data structures, zippers and finger trees, are presented next. Finally, a brief overview is given of verifying the correctness of purely functional data structures and the time and space complexity calculations.

1 Data Structures and Persistence

A data structure is a particular way of organizing data, usually stored in memory and designed for better algorithm efficiency[27]. The term data structure covers different distinct but related meanings, like:

- An abstract data type, that is a set of values and associated operations, specified independent of any particular implementation.
- A concrete realization or implementation of an abstract data type.
- An instance of a data type, also referred to as an object or version.

Initial instance of a data structure can be thought of as the version zero and every update operation then generates a new version of the data structure.

Data structure is called persistent if it supports access to all versions. It’s called partially persistent if all versions can be accessed, but only last version can be modified and fully persistent if every version can be both accessed and modified. Persistent data structures are observationally immutable, as operations do not visibly change the structure in-place, however there are data structures which are persistent but perform assignments. Persistence can be achieved by simply copying the whole structure but this is inefficient in time and space as most modifications are relatively small. Often some similarity between the new and old versions of a data structure can be exploited to make the operation more efficient.
Purely functional is a term used to describe algorithms, programming languages and data structures that exclude destructive modifications. So all purely functional data structures are automatically persistent.

Data structures that are not persistent are called ephemeral. Ephemeral data structures have only have single version available at a time and previous version is lost after each modification.

Many commonly used algorithms, especially graph algorithms rely heavily on destructive modifications for efficiency [19]. It has been showed that there are problems for which imperative languages can be asymptotically more efficient, than strict eagerly evaluated functional languages [28]. However this does not necessarily apply to non-strict languages with lazy evaluation and lazy evaluation has been proved to be more efficient than eager evaluation for some problems [2]. Also in a purely functional language a mutable memory can be simulated with a balanced binary tree, so the worst case slowdown is at most $O(\log n)$. The interesting questions in the research of purely functional data structures are often related to constructing purely functional data structures that give better efficiency and are less complex to implement and reason about. It turns out that many common data structures have purely functional counterparts that are simple to understand and asymptotically as efficient as their imperative versions. The techniques for constructing and analyzing such data structures has been explored in a prominent book [24] and articles by Okasaki and others.

2 Examples of Purely Functional Data Structures

Next we will look at some basic purely functional data structures. The code examples are written in Haskell and are based on the implementations given in [25].

2.1 Linked List

The simplest example of a purely functional data structures is probably a singly-linked list (or cons-based list), shown on Figure 1.

When catenating two lists (++), then in the imperative setting the operator could run in $O(1)$ by maintaining pointers to both first and last element in both lists and then modifying the last cell of the first list to point at the first cell of the second list. This however destroys the argument lists.

In the functional setting destructive modification is not allowed so instead we copy the entire first argument list. This takes $O(n)$ but we get persistence as both argument lists and result list are still available. The second argument list is not copied and is instead shared by the between the argument list and resulting list. This technique is called tail-sharing. A similar technique based on sharing and used commonly for implementing persistent tree-like data structures is called path-copying. In path-copying the path from the place where
Figure 1: Singly linked list

modification occurred to the root of the tree is copied together with references to the unchanged subtrees. [7]

2.2 Red-Black Tree

Red-black tree is a type of self-balancing binary search tree, satisfying following constraints:

- No red node has a red child
- Every path from root to empty node contains the same number of black nodes

This guarantees the maximum depth of a node in a red black tree of size to be $O(lg n)$.

The implementation of a red-black tree is given in Figure 2 and included here mainly as an example of the compactness and clarity that often characterizes functional data structures in functional setting, as the imperative implementation of a red-black tree is usually substantially longer and more complex.

Here `insert` is used to insert a node to the tree by searching top-down for the correct position and then constructing the balanced tree bottom-up by re-balancing if necessary. Balancing is done in `balance` by rearranging the constraint violation of a black node with red child that has a red child into a red node with two black children. Balancing is then continued up to the root.

In addition to lists, which can be also thought of as unary trees, tree data structures are the most important data structures in functional setting. Similar red-black tree implementations are the basis for Scala’s immutable TreeSet and TreeMap[31] and Clojure’s PersistentTreeMap[12]. Most purely functional data structures, including such commonly used data structures as random access lists and maps, are usually implemented using some form of balanced trees.
data Color = R | B
data RBTree a = E | T Color (RBTree a) a (RBTree a)

balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance c a x b = T c a x b

insert x s = T B a y b where
ins E = T R E x E
ins s@(T color a y b) | x < y = balance color (ins a) y b
| x > y = balance color a y (ins b)
| otherwise = s
T _ a y b = ins s

Figure 2: Red-black tree (without the delete node operation)

internally. Examples include Scala’s Vectors [31] and Haskell’s Data.Sequence [32] and Data.Map [33] data structures.

2.3 Queue

Queue is another commonly used data structure, which has the elements ordered according to the first in first out (FIFO) rule. Queues can be naively implemented as ordinary lists, but this means \( O(n) \) cost for accessing the rear of the queue, where \( n \) is the length of the queue. So purely functional queues are usually implemented instead using a pair of lists \( f \) and \( r \), where \( f \) contains the front elements of the queue in the correct order and \( r \) contains the rear elements of the queue in reverse order [13]. Such queue, also called batched queue, can be seen on Figure 2. Elements are added to \( r \) and removed from \( f \). When the front list becomes empty, the rear list is reversed and will become the new front list while the new rear list will be empty.

If used as an ephemeral data structure then the less frequent but expensive reversal would be balanced by more often executed constant-time \texttt{head} and \texttt{tail} operations giving \( O(1) \) amortized access to both front and rear of the queue, but for persistent data structure this means that the complexity degenerates to \( O(n) \). In the next section we will look at ways to ensure better amortized cost for purely functional data structures.

3 Amortized analysis

Data structures with good worst-case bounds are often complex to construct and can give overly pessimistic bounds for sequences of operations as the interactions
between the operations of a data structure is ignored. Amortized analysis \[30\] has been developed to take such interactions into account and give a worst-case bounds that are often more realistic.

3.1 Traditional amortization

Amortized analysis is a technique for analyzing the running time of an algorithm. It is useful in describing the asymptotic complexity over a sequence of operations. It gives an upper bound for the average performance of each operation in the worst case, meaning that a single expensive operation can have worse cost than the bound, but the average cost in any valid sequence will always be within the bound. Two main approaches for amortized analysis are the accounting method and potential method.

- **Accounting method (or banker’s method)**

  In the accounting method, different charges are assigned to different individual operations, with the aim of assigning the total charge as close as possible to the total cost of the sequence of operations. If the actual cost of an operation is smaller than the amortized cost, then the difference is stored as credit and if the actual cost is larger than amortized cost, then the previously allocated credit are used to pay for the difference. The goal is to show that the costs for each operations will be paid so that the total credit will never go negative. This means that the total charge is an upper bound of the actual costs for a sequence of operations.

- **Potential method (or physicist’s method)**
In the potential method a potential function is defined that maps the whole data structure onto a non-negative potential. The amortized cost for an operation is equal to the actual cost plus the change in potential due to the operation. So an operation whose actual cost is less than its amortized cost increases potential while operation whose actual cost is greater than its amortized cost decreases potential.

3.2 Amortization and persistence

Okasaki shows that the traditional potential and accounting methods break down in persistent setting where data structure can have multiple futures [24]. The traditional methods work by accumulating savings for future use and these savings can be only spent once. With persistence the expensive operations can be called arbitrarily often. In accounting method this means that credits can be spent and in potential method potential can be decreased multiple times.

For example let \( f(x) \) be some expensive function application. If each call to \( f(x) \) takes the same amount of time then the amortized bounds will degrade to the worst-case bounds. To overcome this it must be guaranteed that if the first call of \( f(x) \) is expensive then the following calls are not. With \textit{call-by-value} (strict evaluation) and \textit{call-by-name} (lazy evaluation without memoization) this is not possible as each application of \( f \) to \( x \) takes the same amount of time. This means that with such evaluation rules it’s not even possible to describe amortized data structures. In case of the \textit{call-by-need} (lazy evaluation with memoization), the first application of \( f \) to \( x \) is expensive, but after that the result is memoized and each successive call will get get the memoized result directly. This means that if we only care about the worst-case bounds for a sequence and not for individual operations then utilizing lazy evaluation and memoization can result in purely functional data structures with good amortized bounds.

Okasaki develops a framework for analyzing lazy data structures and then modifies the banker’s and physicist’s method providing both techniques for analyzing persistent amortized data structures and practical techniques for analyzing non-trivial lazy programs.

- \textit{Persistent accounting method}

Using lazy evaluation with memoization, it’s possible to modify the accounting method so that the amortized bound are valid even for persistent data structures. Instead of keeping the credits from cheap operations, debits are generated during the suspension of an expensive operation, with each debit representing a constant amount of suspended work. Each operation can discharge a number of debits proportional to its amortized cost. The amortized cost of an operation is the unshared cost of the operation, that is the actual time to execute the operation assuming all suspensions

\footnote{Kaplan and Tarjan however consider memoization to be a side-effect and data structures that use lazy evaluation with memoization not to be “purely functional” [18].}
have been forced and memoized, plus the number of discharged debits. To prove an amortized bound, it’s necessary to show that whenever a location is accessed (which can force the evaluation of a suspension), all debits associated with that location have already been paid off.

- **Persistent potential method**

  Potential method can be also adapted for persistent data structures using lazy evaluation and memoization. The potential function representing lower bound on accumulated savings is replaced with a function representing an upper bound on the accumulated debt. Amortized cost of an operation can be, similarly to accounting method, viewed as the unshared cost of the operation plus the number of discharged debits.

  The main difference between persistent accounting method and potential method is that for accounting method it is allowed to force a suspension as soon as debits for that suspension have been paid off, but in potential method we can force a shared suspension only when the entire accumulated debt has been reduced to zero. The potential method appears thus less powerful, however when it applies it tends to be much simpler.

  Implementation of banker’s queue in Figure 4 is similar to that of the batched queue given in Figure 3, but for banker’s queues it is ensured that the rear never grows bigger than the front list. The debt of reverse is the length of the rear list |r|, so we have to discharge at least |r| debits before the suspension is forced. But to force the suspension of reverse we have to process through f first. By guaranteeing that |f| ≥ |g|, we can discharge credit with tail for each element in f, so that when the suspension of reverse is forced |f| ≥ |r| debits have been discharged. This ensures O(1) amortized access to both ends of the queue regardless of how the queue is used. Note that this method does not work for batched queue as we cannot guarantee to have paid off the debt when suspension is forced. Also it doesn’t work for strict evaluation or lazy evaluation with memoization as the expensive call forcing reverse is then evaluated again each time with O(n) cost. Benchmarks showing the performance differences of naive queue, batched queue and banker’s queue implementations in is given in [22]. However the batched queue approach is shown to be sufficient for most cases.

### 3.3 Eliminating Amortization

Often, such as for real-time systems or interactive systems, predictability and consistency of operations are more important than raw speed, meaning that worst-case data structure are preferable to amortized data structure even if amortized data structure would be simpler and faster. A technique called scheduling can be used to convert amortized data structures to worst-case data structures by systematically forcing lazy components so that no execution of suspension will be too expensive [24]. In case of the amortized banker’s queue in Figure 4 the reverse operation, during the rotation of lists, is paid for with
data BankersQueue a = Queue Int [a] Int [a]
empty = Queue 0 [] 0 []
isEmpty (Queue _ [] _ _) = True
isEmpty _ = False

head (Queue _ [] _ _) = error "empty queue"
head (Queue (_:_) _ _ _) = _
tail (Queue _ [] _ _) = error "empty queue"
tail (Queue lenf (_:f) lenr r) = queue (lenf-1) f lenr r
snoc (Queue lenf f lenr r) x = queue lenf f (lenr+1) (x:r)

queue lenf f lenr r
| lenr <= lenf = Queue lenf f lenr r
| otherwise = Queue (lenf+lenr) (f++reverse r) 0 []

Figure 4: Amortized banker’s queue using the accounting method

previous discharges, but the reverse itself is still an expensive monolithic operation with cost proportional to the length of the rear list. To get worst-case bounds the reversing of r has to be performed incrementally, before it is needed, that is before l becomes empty. As the append (++) is already incremental rotation can made incremental by executing one step of the reverse for every step of the append. The function queue then executes the next suspension and maintains the invariant that |s| = |f| − |r| (which also guarantees that |f| ≥ |r|, so the maintaining lengths is no more needed) The implementation of real-time queue is an be seen in Figure 4 The function rotation combines append with reverse and accumulates the partially reversed list in a. Such queue can be shown to support O(1) time per operation in the worst-case.

4 Design Techniques

In the following section we describe some general techniques for designing functional data structures. The focus is on achieving good amortized bounds for lazy data structures and each of the techniques are described in detail in [24].

4.1 Rebuilding

Many data structures conform to some balance invariants for efficient access. Familiar example is that of binary search trees, which have O(log n) worst-case complexity for balanced trees while having only O(n) for unbalanced trees. Restoring perfect balance after every update is often too expensive. If update
data RealTimeQueue a = Queue [a] [a] [a]

empty = Queue [] [] []

isEmpty (Queue [] _ _) = True
isEmpty _ = False

head (Queue [] _ _) = error "empty queue"
head (Queue (x:_) _ _) = x

tail (Queue [] _ _) = error "empty queue"
tail (Queue (x:fs) rs ss) = queue fs rs ss

snoc (Queue fs rs ss) x = queue fs (x:rs) ss

queue fs rs (s:ss) = Queue fs rs ss
queue fs rs [] = let fs' = rotate fs rs [] in Queue fs' [] fs'

rotate [] [r] a = r:a
rotate (f:fs) (r:rs) a = f:rotate fs rs (r:a)

Figure 5: Real-time queue with scheduling

does not change the balance too much, then it is often possible to postpone re-balancing for many updates and then re-balance the whole data structure. Such technique is called batched rebuilding. It gives good amortized bounds if the rebuilding does not happen too often and the updates do not degrade the performance of later operations too much. As an example of batched rebuilding consider the implementation of queue given in Figure 3. The rebuilding transformation reverses the rear list into the front list, so the queue will be in desired state. This would give $O(1)$ amortized efficiency for ephemeral data structures. Unfortunately this is not for persistent data structures, where the expensive re-balancing operation can be executed arbitrarily often.

Global rebuilding is a technique to achieve worst-case bounds by eliminating amortization from batched rebuilding [26]. The idea is to execute rebuilding incrementally by executing a few steps of rebuilding for each normal operation. Global rebuilding maintains two copies of a data structure, the primary, working copy and the secondary copy which is being built incrementally. All operations are executed on the working copy until the secondary copy is completed. Then the secondary copy becomes new working copy and old working copy is discarded. Keeping the second copy up to date adds more complexity and global rebuilding is often quite complicated. Unlike batched rebuilding, global rebuilding also works in a purely functional setting and several data structures, such as the real time queues of Hood and Melville [13], have been implemented using this technique.

Lazy rebuilding is a lazy variant of global rebuilding. It is considerably sim-
pler but unlike global rebuilding that gives worst-case bounds, lazy rebuilding gives only amortized bounds. However, worst-case bounds can be recovered using scheduling. Lazy rebuilding with scheduling can be viewed as a simple instance of global rebuilding. The implementation of real-time queues given in Figure 5 is an example of combining scheduling with lazy rebuilding.

4.2 Numerical Representation

Consider two simple implementations of lists and natural numbers given in Figure 6.

```haskell
data List a = Nil | Cons a (List a)

tail' (Cons x xs) = xs
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

```haskell
data Nat = Zero | Succ Nat

pred' (Succ x) = x
plus Zero y = y
plus (Succ x) y = Succ (plus x y)
```

Figure 6: Similarity of structure in list and natural number definitions

Other than lists containing elements while natural number do not, the definitions look almost identical. Functions on a list are similar to arithmetic operations on natural numbers. Inserting an element is similar to incrementing numbers, deleting an element resembles decrementing a number and appending two lists is like adding two numbers. Data structures that are designed based on this analogy are called numerical representations. The central idea behind this approach is that such data structures inherit the useful characteristics of the corresponding numerical systems.

A familiar data structure using this property is binomial heap. Binomial heap is a collection of heap ordered trees with at most one tree for each order. The number of nodes in heap uniquely determines the number and order of binomial trees in heap: each binomial tree corresponds to one digit in the binary representation of n. This gives the binomial heap a useful property: merging of two binomial queues is done by adding the binomial trees similar to how the bits are combined when adding two binary numbers.

In [24] Okasaki gives an overview of designing efficient purely functional random-access lists and binomial heaps that take advantage of the properties of different numerical systems like segmented binary numbers and skew binary numbers.
4.3 Bootstrapping Data Structures

Data-structural bootstrapping refers to two techniques used for building data structures with more desirable properties using some existing simpler data structures [4].

First technique is called structural decomposition and this means bootstrapping complete data structures from incomplete data structures. Structural decomposition involves taking some implementation that can handle objects only up to bounded size (possibly zero) and extending it to handle unbounded objects.

The second technique is structural abstraction and this involves bootstrapping efficient data structures from inefficient ones. Structural abstraction is typically used to extend an implementation of collections, like lists or heaps, with an efficient join operation to combine the two collections. Designing efficient insert function, for adding single element, is often easier than designing efficient join. Structural abstraction creates collections containing other collections as elements, so that they can be joined by just inserting one collection into the other.

5 Notable Purely Functional Data Structures

5.1 Zipper

Zipper is a data structure, proposed by Huet [15], that allows to efficiently edit tree-shaped data in a purely functional setting. Tree-like data structures are usually represented as a node which recursively points to its child trees with the root node also referring to the whole tree. Using zipper the tree is instead represented as a pair of a subtree and the rest of the three as the one-hole context for this subtree. Zipper can be thought of as a cursor into a data structure, that supports navigation and changing some node deep inside the tree. This allows localized changes to take \(O(1)\) instead of \(O(lg \ n)\). It’s useful whenever there is a need for a focal point for editing: a cursor in a text editor, current directory for a file system etc.

While the initial zipper was constructed for a particular data structure, it was a surprising discovery that the zipper for any suitably regular data structure can be obtained by taking the derivative of a data structure using the rules of differentiation familiar from mathematical analysis [21].

5.2 Finger Trees

Another interesting data structure is the 2-3 finger tree. 2-3 finger trees are a functional representation of persistent sequences, that support amortized constant time access to both ends and and concatenation and splitting in time logarithmic in the size of the smaller piece [14].

Sequence can be represented by an ordinary balanced tree, however the add and remove from both ends of such structure typically take time logarithmic in
the size of the tree, but for an efficient implementation of sequence it is important to have constant time access to both ends. The problem seems similar to what zippers are designed to solve and in fact the concept of 2-3 finger trees is very similar to zippers. Finger is defined as a structure providing efficient access to nodes of a tree, near the distinguished location[11]. For efficient access to the ends of a sequence, such fingers should be placed to the left and right ends of a tree. As a starting point let’s have some ordinary 2-3 tree having all of the leaves at the same level representing a sequence. If this tree is then picked up from the leftmost and rightmost internal nodes and let other nodes to drop lower until they hang by the fingers, we get a structure where we have references to the fingers representing the ends of the sequence and rest of the tree dangling down. Each pair of nodes forming central spine of the tree is then merged into a single deep node. At each level of depth \( n \) of the resulting finger tree, there are left and right fingers called digits that represent ordinary 2-3 trees with depth \( n \) and at center a subtree, which is another finger tree. This describes the basic structure of finger trees. A Finger Tree can be parameterized with a monoid, and using different monoids will result in different behaviors for the tree. This lets Finger Trees simulate other data structures. Another property of 2-3 Finger trees is can be parametrized with a monoid and by using different monoids different behaviour can be achieved for the tree. The tree and it’s nodes are annotated with a measure, which can be the size of the tree when implementing random-access sequences or priority when implementing max-priority queues etc. This means that finger trees provide a general purpose data structure that can be used in implementing efficiently various different data structures such as sequence, priority queue, search tree, priority search queue and more.

Haskell’s Data.Sequence [32] is implemented using the 2-3 finger trees.

6 Cyclic Data Structures

Functional programming languages are well equipped for manipulating tree-like data structures. Such data structures are declaratively described by algebraic data types and related functions that use pattern matching on these types. However the data is often better described using cyclic data structures like doubly linked list or cyclic graphs and unfortunately the solutions for manipulating cyclic data structures are not as good. Traditionally the graph is represented by adjacency list table and when traversing the graph a list of visited nodes should be kept to avoid visiting a node multiple times and ending up in an endless loop[19]. In impure languages like ML mutable references are often used, while in non-strict languages like Haskell cyclic structures can be represented using recursive data structures. For example a cyclic list with elements 1, 2 and 3 can be defined in Haskell by:

\[
\text{clist} = 1:2:3: \text{clist}
\]

However such representation is not distinguishable from infinite list and does not allow explicit observation or manipulation of the cycle. Various improvements
have been proposed for supporting cyclic data structures in functional languages such as using recursive binders [9], graph constructors [8] or nested data types [10] to model cycles and sharing.

7 Correctness and Verification

Verifying the correctness of data structures and the time and space complexity calculations are another important area of research in data structures. Types are traditionally used to contribute to the correctness of programs, so one natural direction is to study more expressive types such as phantom types [20], nested datatypes [3] and generalized abstract data types [5]. Using extensions to Haskell’s type system like phantom types, nested datatypes and existential types the red-black tree presented in was implemented with type-level correctness guarantees for the color invariants [16]. With the recent developments and success of proof assistants like Coq and Agda, a popular approach is to implement data structures in such a proof assistant to verify its properties. The correctness of the original implementation of finger trees as well as a certified implementation of random-access sequences built on top of finger trees was given in [29] using the Coq proof assistant. The time analysis for lazy programs, based on the Okasaki’s accounting method, was formalized into a library using the Agda proof assistant, where the type system was used for tracking the time complexity of functions which were then combined together [6].

8 Summary

Unlike the overwhelming amount of literature available for ephemeral data structures, resources describing purely functional data structures are rather scarce. There is a single influential book by Okasaki, but there has not been any other comprehensive overviews or books in this area. Major part of this work introduced the concepts of purely functional data structures, provided some simple implementations and described techniques to design efficient data structures. The efficiency of the data structures and techniques covered depend largely on lazy evaluation with memoization. Most of this is based on the book by Okasaki. As can be seen, achieving good performance for purely functional data structures demands sometimes quite sophisticated approach to designing the data structures. On the other hand, it is often possible to design purely functional data structures with performance comparable to ephemeral data structures, while retaining the benefits of guaranteed persistence and immutability.

The final part of this work gives a brief introduction of two general-purpose purely functional data structures, zippers and 2-3 finger trees, as well as of studies related to cyclic data structures in functional setting and the formal verification of the correctness of data structures and time-space complexity calculations.

An area that could be expanded more, is to describe the benefits and appli-
cations of the purely functional data structures, such as the inherent referential transparency and immutability that are useful for both in formal analysis of data structures as well as for parallel computing.

Some of the more interesting data structures not covered in this overview are tries and radix trees in purely functional setting and their applications like the hash array mapped tries [1], which is used as a basis for the hash map implementations of Clojure and Scala.

References


