Contents

3.4 Binary Trees

There are no publicly available libraries of Immutable Data Structures implemented in imperative programming languages. This section describes the design of a general purpose Immutable Data Structure. This flexible design can be specialised to implement a variety of ADTs. The topology of the data structure is hidden from the program so it can be balanced independent of the ADT that it implements.

In a functional programming language immutable values are maintained in purely functional data structures, such as those described by Okasaki [Oka04]. Purely functional data structures might appear to be a starting point for developing Immutable Data Structures. However, functional programming languages permit the expression of a function in terms of immutable data, whereas the evaluation of a function typically relies on mutable data. In some cases a functional programming language compiler implements a purely functional data structure as a mutable structure. In a concurrent execution environment it is the *actual* immutability of values used during the execution that matters rather than the *appearance* of immutability presented to the programmer by the programming language.

The main contribution of this section is the development of a general Immutable Data Structure that can be used to maintain speculative and shared state. This section focuses on design flexibility and subsequent sections show how the structure can be specialised to conform to a variety of ADTs.

3.4.1 A flexible Immutable Data Structure design

A general purpose Immutable Data Structure should be flexible enough to present a variety of familiar ADTs to the program. Design flexibility is a vague term but we take it to mean three things. Firstly, we prefer the simplest most general solution. In practice, this means designs that contain no special cases. Secondly, we prefer to hide details of the data structure implementation from the application. In practice, this means making details of the topology inaccessible to the ADT. Thirdly, we delay performance optimisations until the final stages of implementation.

3.4.2 The Canonical Binary Tree

The proposed solution is an immutable binary tree that can be specialised to conform to a particular ADT. We call this structure the Canonical Binary Tree because it is an immutable binary tree reduced to the simplest and most significant form possible without loss of generality. This section states the decisions on which the design of the Canonical Binary Tree is based.

Why a binary tree?

A tree in which each node has many children can be shallower than a binary tree containing the same number of leaves. It is common for purely functional data structures to be based on shallow trees so that access times are minimised.

For example, the Clojure language implements a number of purely functional data structures internally. These structures are based on a 32 bit hash array mapped trie. Each node of the trie has up to 32 children so the structure is shallow and permits fast access. Bagwell describes the implementation of these data structures in detail [Bag01]. A hash array mapped trie is a complex structure optimised for good performance on modern computer hardware, but it is difficult to implement. Clojure offers just a few Immutable Data Structures as primitives and the language offers no control over the implementation of the underlying data structure. At the time of writing a new ADT is under development by the Clojure community. Hickey describes the performance benefits of the hash array mapped trie and the significant work involved in implementing ADTs based upon it [Hic11].

In a concurrent execution environment access time is not the most important design consideration and optimisation can be deferred to a later stage in the design process. The binary tree has the simplest possible structure and offers the greatest design and implementation flexibility.

Why associate values exclusively with leaves?

A Canonical Binary Tree contains both structural information and application values. Structural information is necessarily associated with the nodes but application values can be associated either exclusively with leaves or with both leaves and nodes, which we refer to as vertices. A tree that associates application values exclusively with leaves can hide its topology.

For example, the priority queue ADT associates a priority with an application value. It is common for a priority queue to be implemented as a binary heap in which both a priority and an application value are associated with each vertex. Both the ephemeral priority queue considered by Sedgewick [Sed98] and the purely functional priority queue considered by Okasaki [Oka04] associate an application value with a vertex because a vertex can be accessed more quickly than a leaf. When a priority queue is implemented by a binary heap the highest priority vertex can be accessed in $O(1)$ time and insertion into the queue takes $O(log_2(n))$ time. However, when a priority queue is implemented by a tree with application values associated exclusively with leaves the access time for all operations is $O(log_2(n))$.

The Canonical Binary Tree associates application values exclusively with leaves. All functions access leaves so the amortised access time is:

$$O(log_2(n))$$

This amortised access time is identical to that of an ephemeral binary tree with application values maintained exclusively by leaves.

Why separate keys and annotations?

A key is an argument to a function of a data structure, whereas a vertex annotation is a value used to navigate a path through the tree. Usually, annotations and keys are of the same type and annotations are accessible to the program.

The Canonical Binary Tree design separates the concepts of keys and annotations. Annotations are not accessible to programs so the topology of the tree can be altered independent of the ADT being implemented.

For example, an associative data structure in which all values are reachable, such as a map, is typically distinguished from one in which not all application values are reachable, such as a deque. However, the front and back functions of a deque can be regarded as a query function that takes as its access argument a binary key indicating which end of the queue it acts upon.

By separating the concept of the annotation from the key all ADTs can be regarded as associative. The Canonical Binary Tree treats all ADTs as associative and hides the details of the annotations from the calling program.

Why fix the comparison function?

The function that determines the annotation of a node given the annotations of its children is referred to as the annotator and the operation that determines which of the children of a node is on the path is called the comparison. The annotator function specialises the Canonical Binary Tree so that it conforms to a particular ADT. The Canonical Binary Tree uses the same comparison function for every ADT.

For example, a path through a Binary Search Tree can be determined by a comparison function that causes the right child of a node to be selected if the access argument is greater than its annotation. This causes an in-order traversal to return application values in ascending order of the access argument used to insert them. The order of the elements returned by an in-order traversal can be reversed either by using a different comparison function or by inserting the values using a different annotator.

The Canonical Binary Tree fixes the comparison function to reduce the amount of information that must be specified to specialise it to conform to a particular ADT.

Why maintain a sentinel leaf?

There is a distinction between a data structure that is empty and a data structure that does not exist. This is particularly important for data structures that are accessed concurrently.

An empty Canonical Binary Tree contains a sentinel leaf which is always present within the tree. We adopt the convention that the sentinel is always the right-most leaf of the tree.

Which access functions should the Canonical Binary Tree implement?

The Canonical Binary Tree implements only the access functions: $create()$, $insert()$, $query()$, $delete()$ and $empty()$. The interface functions required by common ADTs, such as $Top()$, $Front()$ etc. are implemented by wrapper functions.

The $create()$ function creates a new data structure containing only the sentinel. Its parameters specify the appropriate sentinel annotation for the ADT being implemented and an application value. The function allocates storage for the root and returns a reference to it. The root is initialised with a reference to the sentinel. References to the root and the sentinel are maintained by the program.

The *query*() function returns an application value. The function accepts: an access argument, a reference to the root and a reference to the sentinel as its parameters. It is ADT agnostic and does not require a specialising function as a parameter. Its access argument will always match a single leaf within the tree. When the tree is empty it returns the application value of the sentinel.

The *insert*() function always succeeds in inserting a leaf into the tree and has no return value. The function accepts: a specialising annotator function, an access argument, a reference to the root and a reference to the sentinel as its parameters.

The *delete*() removes a leaf from the tree unless it is empty and has no return value. The function accepts: a specialising annotator function, an access argument, a reference to the root and a reference to the sentinel as its parameters. The sentinel cannot be deleted and an instance of the Canonical Binary Tree persists until the program terminates, so there is no function to delete an entire data structure.

The *empty*() function is a macro that compares the address of the sentinel with the address of the root node, both of which are maintained by the program and passed as parameters.

3.4.3 Previous work

Sedgewick provides a comprehensive guide to important ephemeral data structures [Sed98]. Okasaki provides a comprehensive guide to purely functional data structures [Oka98].

Tarjan describes methods of amortised time analysis called the Banker's and Physicist's methods [Tar85]. Okasaki adapts these analyses to purely functional data structures [Oka98]. The Banker's method associates credits and debits with short and long paths in the data structure respectively. The analysis balances the debits and credits to determine the effective cost of an operation. The Physicist's method describes a function mapping each element in the data structure onto a real number called its potential. The analysis balances the positive and negative potential of accesses to particular elements to determine the effective charge of an operation. These analyses are more complicated than ours because the ADTs presented are tightly coupled to the data structures that implement them.

Okasaki focuses on the path copying technique and the diagrams in the book imply that the programmer should visualise path copying when thinking about the structures. However, in a functional programming language a data structure is specified at a high level of abstraction and how the language compiler implements the structure is not specified. In some cases path copying is used by the generated code but this is compiler dependent. A structure that appears immutable when described in a functional programming language might be compiled to a mutable structure to improve performance.

Moss describes a set of benchmark applications that can be used to assess the performance of purely functional data structures [Mos99].

Prior to this thesis there were no publicly available libraries of Immutable Data Structures implemented in an imperative programming language. We do not know of any previous attempts to produce such a library.

Persistent Data Structures implemented in an imperative programming language are typically bespoke solutions to problems in algebraic geometry or version control. Sarnak describes how a persistent data structure can be used to solve the planar point location problem in computational geometry [ST86]. Pluquet describes how to construct a partially persistent data structure in C++ to solve the same planar point location problem [PLMW08]. These persistent data structures use the fat node technique so they are not immutable.

Parrish describes a class based implementation of persistence in C++ [PDC+98]. The problem that Parrish addresses is one of transforming a general application class into a persistent class. The resulting data structure is immutable but new versions can only be created by copying the entire object.

The C++ STL contains several associative ADTs that are usually implemented by a balanced red-black tree [Jos99]. The STL separates the concerns of the ADT from those of the data structure that implements it. The STL separates the ADT from the balancing process. STL iterators separate the ADT from the process of traversing the tree. STL allocators separate the ADT from the memory management processes so the data structure implements a container.

Hinze describes how a similar separation of concerns can be applied to a purely functional data structure [HP05]. Hinze describes a general technique for creating Immutable Data Structures in a functional programming language. This technique has not previously been explored in the context of imperative programming. Hinze reduces the amortised access time of a binary tree by adding a central spine, to create a so-called finger tree. However, the spine is just an access time optimisation. Hinze describes how a specialising function can be used to make an immutable binary tree conform to a particular ADT. Hinze shows how monoid functions, which are associative functions with an identity, can be used to specialise a binary tree.

A finger tree is statically specialised to conform to a particular ADT, whereas the Canonical Binary Tree is dynamically specialised. The set of access functions associated with each structure implemented by a finger tree is ADT dependent, whereas the Canonical Binary Tree presents a basic set of functions that can be adapted to implement a particular ADT.

Hinze's design is based on an Immutable Data Structure that requires both a function to determine the annotation of a node given its children and a comparison operation to determine the path, whereas the Canonical Binary Tree requires only one specialising function.

Finally, Hinze does not make a distinction between an empty tree and a non-existent tree, whereas the Canonical Binary Tree maintains a sentinel to make this distinction.

Bibliography

[Bag01] Phil Bagwell. *Ideal Hash Trees*. PhD thesis, Department of Computer Science, Ecole Polytechnique Federale de Lausanne, 2001.

[Hic11] Rich Hickey. Clojure concurrency (video). http://blip.tv/file/812787, January 2011.

[HP05] R Hinze and R Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Prog.*, 16(02):197–217, 2005.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Mos99] Graeme E. Moss. Benchmarking purely functional data structures. *Journal of Functional Programming*, 11:525–556, 1999.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998.

[Oka04] Chris Okasaki. Purely functional structures. In *Handbook Of Data Structures And Applications*. Chapman & Hall/CRC, 2004.

[PDC$^+$98] Allen Parrish, Brandon Dixon, David Cordes, Susan Vrbsky, and John Lusth. Implementing persistent data structures using C++. *Softw. Pract. Exper.*, 28:1559–1579, December 1998.

[PLMW08] Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *ALENEX Algorithm Engineering and Experiments*, pages 37–48, 2008.

[Sed98] Robert Sedgewick. *Algorithms in C++, parts 1-4: fundamentals, data structure, sorting, searching, third edition.* Addison-Wesley Professional, 1998.

[ST86] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[Tar85] Tarjan, R. E. Amortized computational complexity. *SIAM J. Alg. and Discr. Meth.*, 6(2):306–318, 1985.