Contents

4.2 Persistent Data Structures

Linearizability is a desirable property for the objects that encapsulate shared state in a concurrent system. However, we have proposed that shared state should be maintained in Immutable Data Structures. This section considers how the access functions of an Immutable Data Structure can be made linearizable. Linearizability endows the functions that access shared state with intuitive concurrent semantics.

A persistent data structure permits access to past versions but in the context of serial execution Immutable Data Structures do not. During concurrent execution access to a past version occurs when the root of the structure is modified by a function executing on another processor. This causes the version being accessed to become a past version. The tardiness of a function permits access to a past version. An Immutable Data Structure that permits access to past versions is persistent. This section considers how access to past versions can be controlled to ensure the property of linearizability.

The main contribution of this section is the implementation of Immutable Data Structures as linearizable objects. This section focuses on mechanisms to restrict access to past versions of an Immutable Data Structure.

4.2.1 Accessing Previous Versions

The property of immutability permits a separation of concerns about the integrity of a data structure from concerns about the semantic order of the functions acting on it. A Canonical Binary Tree preserves its structural invariants during concurrent operation. However, it does not provide any mechanism for ensuring the semantic ordering or the invariants of the functions acting upon it.

For example, consider a set implemented by a Canonical Binary Tree. A function inserts a value into the set if and only if the value is not already present. Structural integrity can be described by a set of invariants related to the connectedness of the tree. During concurrent execution the structure of the binary tree is guaranteed because its structural invariants are the pre and post conditions of the path copy operation which implements the function. However, the uniqueness of values in the set is guaranteed by pre and post conditions of the function. The semantic invariants are guaranteed by the ADT and the structural invariants are guaranteed by the data structure.

A function can ensure both the structural and the semantic invariants of a data structure by enforcing mutual exclusion. The use of mutual exclusion is so pervasive that programmers do not usually distinguish between the semantic and structural invariants.

A proof of the linearizability of an object typically requires the identification of the linearization point of each method acting on the object [HW90]. The linearization point of a method is some moment in time between the invocation of the method and its response. Prior to this moment in time the pre-conditions of the method are true and after it the post-conditions are true. To be linearizable the execution of the method must appear to take place atomically at its linearization point. An Immutable Data Structure has both structural and semantic invariants that can be considered separately.

For example, two processors might attempt to insert the same value into an Immutable Data Structure concurrently. Both of the operations complete eventually and the structural invariants of the data structure are preserved. Each of the instances of the function appears to modify the structure at a unique moment in time so the data structure has a linearization point and is structurally linearizable.

Now consider the semantic order of operations on the set. If two instances of a function attempt to insert the same value concurrently both succeed. The resulting data structure contains a duplicate value that violates the invariants of the function. There is no instant in time when the operation can be considered to have taken place. Consequently, there is no semantic linearization point so the Immutable Data Structure is not semantically linearizable.

To imbue an Immutable Data Structure with the desirable properties of a linearizable object all of its access functions must be linearizable. Structural linearizability is the concern of the Canonical Binary Tree and the semantic linearizability is the concern of the ADT. The access functions of an Immutable Data Structure must have both semantic and structural linearization points in order that the Immutable Data Structure is linearizable.

4.2.2 Persistence

An Immutable Data Structure can be made structurally linearizable by requiring that the root is modified by an atomic instruction. Immutable Data Structures can be made semantically linearizable by recording the root at the moment of functional invocation and validating that the root has the same value at the moment of response. If the value of the root has changed then the function is invalid.

The root can be modified while a function is active. When this occurs two functions can be reading different versions of the same data structure concurrently. The version referenced by the current value of the root is regarded as the current version. The version that was referenced by the root at some point in the past can be regarded as a past version. An Immutable Data Structure that permits access to past versions is called a persistent data structure.

An Immutable Data Structure in which all access functions record the root when they start and validate it before replacement is ephemeral. Functions acting on versions other than the current version are not successful. An ephemeral Immutable Data Structure is semantically linearizable.

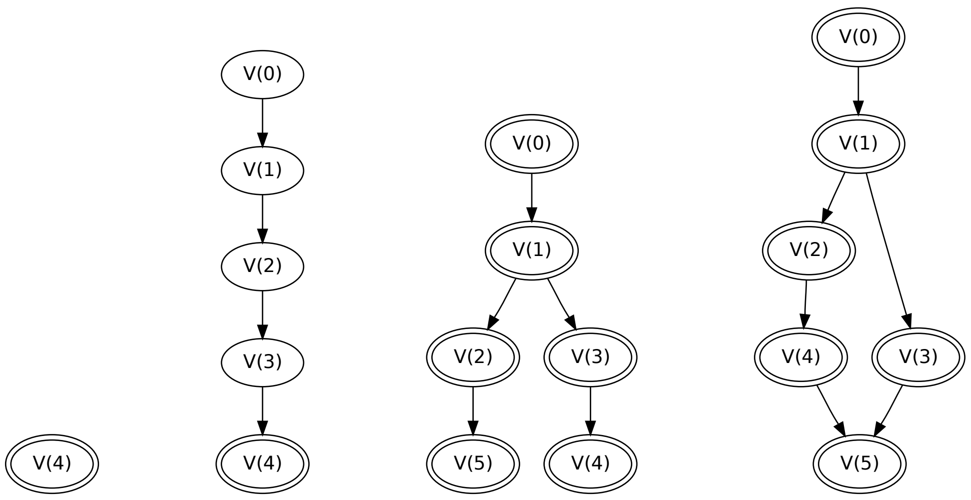
An Immutable Data Structure in which only mutating access functions record the root when they start and validate it before replacement is partially persistent. Past versions can be accessed by tardy readers but successful mutations always act on the most recent version. Mutations acting on past versions are not successful. A partially persistent Immutable Data Structure, like all of the Immutable Data Structures we consider in this thesis, are structurally linearizable.

Section 4.2.3 describes the classification of persistent data structures.

Concurrent applications implemented using mutual exclusion always access the most recent version of a data structure, because it is the only version available. Programmers are not used to questioning whether the most up to date version is required. In many application domains the very latest version is not always required. For example, an on-line reservation system does not need to present the customer with the most recent version of inventory while they are browsing, but an up to date inventory is required when the customer is making a purchase. A partially persistent data structure is suitable for such a purpose because it ensures that mutations are serialised while permitting concurrent read accesses.

4.2.3 The classification of persistent data structures

Most of the data structures encountered by programmers are both mutable and ephemeral. A data structure is called persistent if it permits access to all versions and it is called ephemeral otherwise. The data structure is partially persistent if



(a) Ephemeral (b) Partial persistence (c) Full persistence (d) Confluent persistence

Figure 4.1: **Version graphs** for various types of persistent data structure. Versions which can be modified are represented by an ellipse with a double border.
(a) An ephemeral data structure restricts both read and write access to the most recent version.
(b) A partially persistent data structure restricts write access to the most recent version but permits read access to past versions.
(c) A fully persistent data structure permits both read and write access to all versions.
(d) A confluently persistent data structure permits both read and write access to all versions and provides a meld function that can combine past versions.

all versions can be accessed but only the most recent can be modified. The structure is fully persistent if every version can be both accessed and modified. The data structure is confluently persistent if it is fully persistent and has a meld operation which combines more than one version. The evolution of a persistent data structure can be represented by a directed graph in which each vertex represents a version and each edge a transformation between versions [Kap04].

A version graph is a representation of the evolution of a data structure.

Figure 4.1 illustrates the version graphs of common types of persistent data structure.

A persistence type is a restraint on the access to past versions. There are many possible restraints and therefore many persistence types. Conchon describes a semi-persistent data structure which permits access only to those past versions that are ancestors of the most recent version [CF08].

An Immutable Data Structure is composed entirely of immutable values. Informally, we describe a persistent data structure as an Immutable Data Structure that possesses a look-up function capable of accessing past versions. However, not all persistent data structures are Immutable Data Structures. Persistent Data Structures constructed using the Fat Node or the Node Copying techniques contain singly assigned values that are not immutable. Persistent Data Structures constructed using the full copying or path copying technique are Immutable Data Structures.

4.2.4 Previous work

Driscoll describes persistent data structures in a seminal paper which details all of the techniques introduced in this section [DSST86]. Kaplan provides an accessible introduction to persistent data structures [Kap04].

Interest in persistent data structures originated from the development of text editors. Reps proposed that persistence can be used to create a text editor with an undo operation [RTD83].

Sarnak describes how persistent data structures can be applied to solve problems in computational geometry [ST86]. Computational geometry is a branch of computer science concerned with algorithms that can be stated in terms of geometry. Computation geometry is relevant to the subject of computer graphics and much of the early development of persistent data structures occurred in the mid 80's when computer graphics became commercially important.

Persistence type	**Permitted read access**	**Permitted write access**
Ephemeral	Most recent only	Most recent only
Partially persistent	Tardy reads	Most recent only
Fully persistent	Tardy reads	Tardy writes
Confluently persistent	Non-conflicting reads	Non-conflicting writes

Table 4.1: **Persistence types for Transactional Data Structures** are characterised by the access to past versions that they permit.

The challenges of computational geometry required complex data structures so the first persistent data structures to be developed were certainly not the simplest. The subject of persistent data structure has always focused on highly optimised solutions to challenging problems whilst the implementation of simple persistent data structures has been somewhat neglected.

The research literature does not make a distinction between persistent data structures and Immutable Data Structures. The use of a persistent data structure in a concurrent execution environment and the access to a past version by a tardy reader have not been considered before.

To permit access to past versions a persistent data structure must implement some look-up mechanism. Generally, the research literature describing persistent data structures does not describe the version look-up function in detail. The focus of research is on the accessibility of past versions rather than the mechanisms to support that access. Previous work either neglects the details of the look-up function or assumes that versions are accessed by indexing an array mapping a version number to the root address of the corresponding version.

4.2.5 The classification of Transactional Data Structures

This thesis introduces Transactional Data Structures which are data structures that permit access to past versions, although not all accesses are successful.

For each type of persistent data structure there is a corresponding Transactional Data Structure. A Transactional Data Structure with linearizable access functions is ephemeral because accesses to past versions are not successful. A Transactional Data Structure that permits tardy readers but prevents writers from successfully accessing all but the most recent version is partially persistent. A fully persistent Transactional Data Structure permits both tardy readers and mutations. A Transactional Data Structure which permits simultaneous accesses while ensuring serialisability is confluently persistent. It has a validate function that causes conflicting functions to be unsuccessful and a meld function that unites past versions.

Table 4.1 summarises the persistence types for Transactional Data Structures.

A Transactional Data Structure is necessarily an Immutable Data Structure because it permits simultaneous access to values.

Bibliography

[CF08] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[DSST86] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Kap04] Haim Kaplan. Persistent data structures. In *Handbook Of Data Structures And Applications*. Chapman & Hall/CRC, 2004.

[RTD83] Thomas W. Reps, Tim Teitelbaum, and Alan J. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, 1983.

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