Contents

6.2 Non-blocking Algorithms

The construction of non-blocking algorithms is a challenging programming task. Non-blocking algorithms are scalable because they permit simultaneous access to a data structure and they offer strong progress guarantees without requiring centralised contention management. This section describes a simple technique for constructing non-blocking algorithms. Non-blocking functions acting on Immutable Data Structures are the foundation on which scalable concurrent programs can be built.

Non-blocking algorithms are scalable and relieve the programmer from having to reason about locks. A concurrent system in which semantically linearizable functions act concurrently on an Immutable Data Structure can guarantee lock-free progress. In order to simplify the construction of non-blocking algorithms concurrent systems should also ensure that serialisable functions guarantee lock-free progress.

The main contribution of this section is a general technique for implementing non-blocking algorithms. This section focuses on allowing simultaneous access to Immutable Data Structures.

6.2.1 Ensuring serialisability without blocking

The enforcement of serialisability is more involved than the enforcement of linearizability because conflict detection is performed by a function instead of an atomic hardware instruction.

Semantic linearizability is enforced by recording the value of the root of the Immutable Data Structure at the start of the execution of an access function and checking that its value has not changed before atomically replacing the root at the end of the execution. This replacement relies on an atomic compare-and-swap instruction which serialises access to the root and implements a memory barrier that ensures that the speculative path is atomically transformed into a new shared version of the data structure.

Semantic linearizability is lock-free because it guarantees that when the program runs for sufficiently long at least one function makes progress. An access function can be prevented from completing only if the value of the root changes whilst it is executing. However, if the value of the root changed then another function successfully completed so at least one function made progress.

Semantic linearizability does not achieve the stronger condition of wait-freedom. An algorithm is wait-free if every operation eventually completes. An access function can be prevented from completing if the value of the root changed whilst it was executing. The function can be re-tried indefinitely but there is no guarantee that it will eventually succeed.

The serialisability of simultaneous accesses to an Immutable Data Structure is enforced by a validate function that implements the Time Stamp Ordering concurrency control protocol and a meld function that can combine two version of the data structure making it confluently persistent. The validate and meld functions do not take place instantaneously so the access function checks that no conflicting accesses occurred while they were executing. It also checks that no accesses complete successfully while the validation and meld functions are executing. The problem is to combine these actions in a way that guarantees progress.

6.2.2 Lock-free serialisability

An access function of an Immutable Data Structure includes both validate and meld functions which implement a simple distributed transaction manager. This transaction manager combines two forms of speculation. The first speculation is that the access function does not conflict with any functions accessing the data structure simultaneously. The second speculation is that the root of the Immutable Data Structure does not change while the validate and meld functions take place.

Figure 6.1 illustrates the execution of an access function in the presence of concurrent mutations.

An Immutable Data Structure access function records the value of the root of the Immutable Data Structure at the moment in time that it starts. This reference represents the starting version of the data structure. The function is applied to this starting version to produce a path in isolation. The function records the value of the root of the Immutable Data Structure at the moment in time that it completes the path, we call this version the first snapshot. This reference represents a version of the data structure which might have been arbitrarily mutated by concurrently executing functions. The validation function ensures that the execution of the function and the first mutation do not contain conflicting operations. The meld function combines the path with the first snapshot version

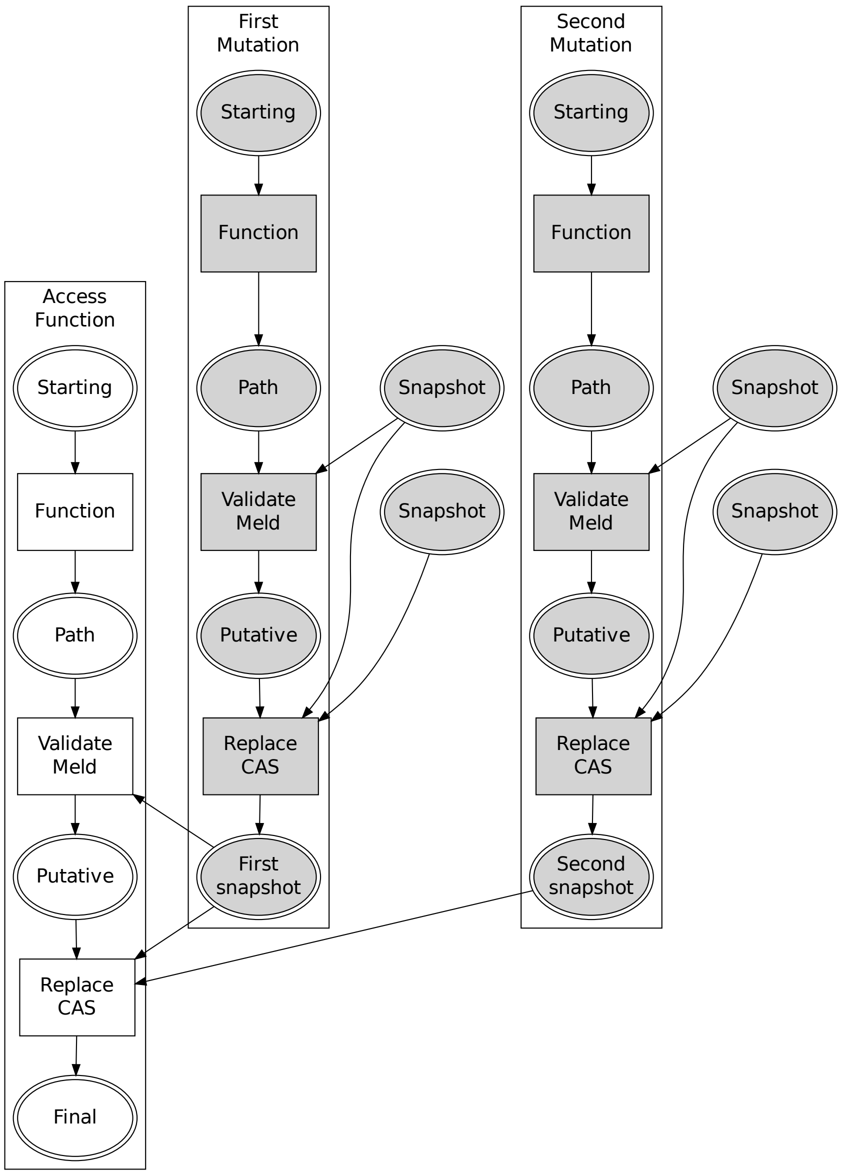

Figure 6.1: **The execution of an access function in the presence of concurrent mutations.** Each operation takes a version of the data structure, represented by an ellipse, as its argument and produces a new version. The operations executed by another processor are shaded.
The first mutation may successfully complete, while the access function is executing, creating the first snapshot version. If so, the path created by the access function is validated against this snapshot version and melded with it to create a putative path. A second mutation may complete, while this validation is taking place, creating the second snapshot version.

to produce a putative version of the data structure. Both the validate and meld functions execute in isolation, their only inputs are the path and the first snapshot version. An atomic compare-and-swap instruction replaces the root with the putative version if and only if the root has the same value as the first snapshot.

An Immutable Data Structure access function completes successfully if and only if both speculations are successful. The first speculation is that the access function does not conflict with the first mutation. The second speculation is that the second mutation does not complete successfully in the period between the first and second snapshot. In the first case a conflict is detected after the first mutation successfully modified the root which implies that the first mutation made progress. In the second case the value of the root only changes after the second mutation successfully modified it which implies that the second mutation made progress. Either the function or one of the mutations makes progress in each case. The execution is lock-free because it guarantees that when a program runs for sufficiently long at least one processor makes progress.

6.2.3 Previous work

Herlihy describes the state of research into non-blocking algorithms in a book entitled ‘The art of multiprocessor programming’ [HS08].

Non-blocking algorithms which access mutable values are complex, difficult to reason about and are usually regarded as the domain of expert programmers. Given an ADT there is no general technique for constructing a non-blocking algorithm that conforms to it.

A particularly difficult problem, the ABA problem, contributes significant complexity to the implementation of non-blocking algorithms. Fraser describes the ABA problem which is a pathology of the atomic compare-and-swap instruction which occurs when addresses are re-used [FH07]. The implementation of a non-blocking algorithm is simplified by ensuring that all the data it acts upon is immutable and that addresses are not re-used. The immutability of the vertices of an Immutable Data Structure ensures that the root cannot be assigned the same value more than once so the ABA problem cannot occur.

An atomic compare-and-swap instruction cannot modify two non-contiguous locations so non-blocking data structures with cycles, such as doubly linked lists, are difficult to construct. The ADTs presented by non-blocking algorithms are often similar to those presented by purely functional data structures which are

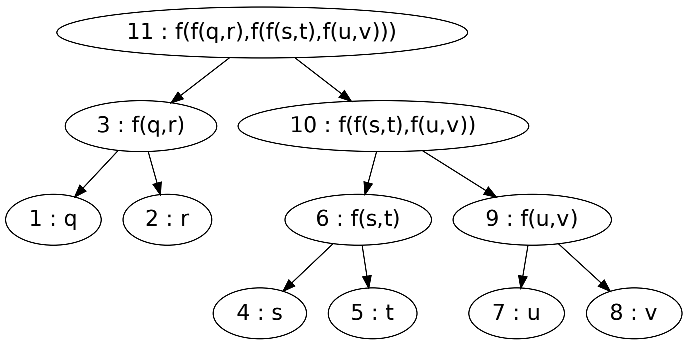

Figure 6.2: **The abstract syntax tree of an expression** in which each value is associated with a tag number. The expression is $f(f(q,r), f(f(s,t), f(u,v)))$.

also single pointer structures.

Goetz examines the performance of the non-blocking algorithms included in `java.util.concurrent` library [GBB$^+$06].

Allemany found that many non-blocking algorithms perform badly on Chip Multi-Processors [AF92]. This poor performance can be attributed to the use of the atomic compare-and-swap instruction which may take thousands of clock cycles to complete. Hennessy provides an introduction to the complex performance issues surrounding the use of the atomic compare-and-swap instruction [HP06]. Non-blocking routines which use the atomic compare-and-swap instruction sparingly can perform very well.

6.2.4 Non-blocking evaluation

Non-blocking algorithms based on immutable data are more flexible than those based on mutable data. The following example illustrates how a non-blocking algorithm can be used to load-balance the evaluation of an arbitrary expression on multiple processors. This problem is introduced in section **??**.

The expression $f(f(q,r), f(f(s,t), f(u,v)))$ can be described by an abstract syntax tree. The problem is to load balance the evaluation of the expression between processors. It is difficult to scheduler the concurrent execution of this expression because the execution time of each function is not known. A solution is to dynamically schedule the execution of functions as their arguments become available.

Figure 6.2 illustrates the abstract syntax tree of the expression.

The evaluation may be dynamically load-balanced by recording intermediate values in an Immutable Data Structure. Initially, the data structure contains only the arguments of the expression. The final version contains all of the arguments and intermediate values as well as the result. The data structure maintains an immutable record of the evaluation of the expression.

Figure 6.3 illustrates the initial and final versions of an Immutable Data Structure which represents the evaluation of the expression.

Figure 6.4 illustrates the non-blocking evaluation of the expression by multiple processors.

In the illustration, each function's arguments are available in the version of the Immutable Data Structure that it starts with. If the function is unable to find its arguments in the Immutable Data Structure then it is re-started with a new version. Eventually, all of the functions in the expression complete and the value of the expression can be obtained from the Immutable Data Structure.

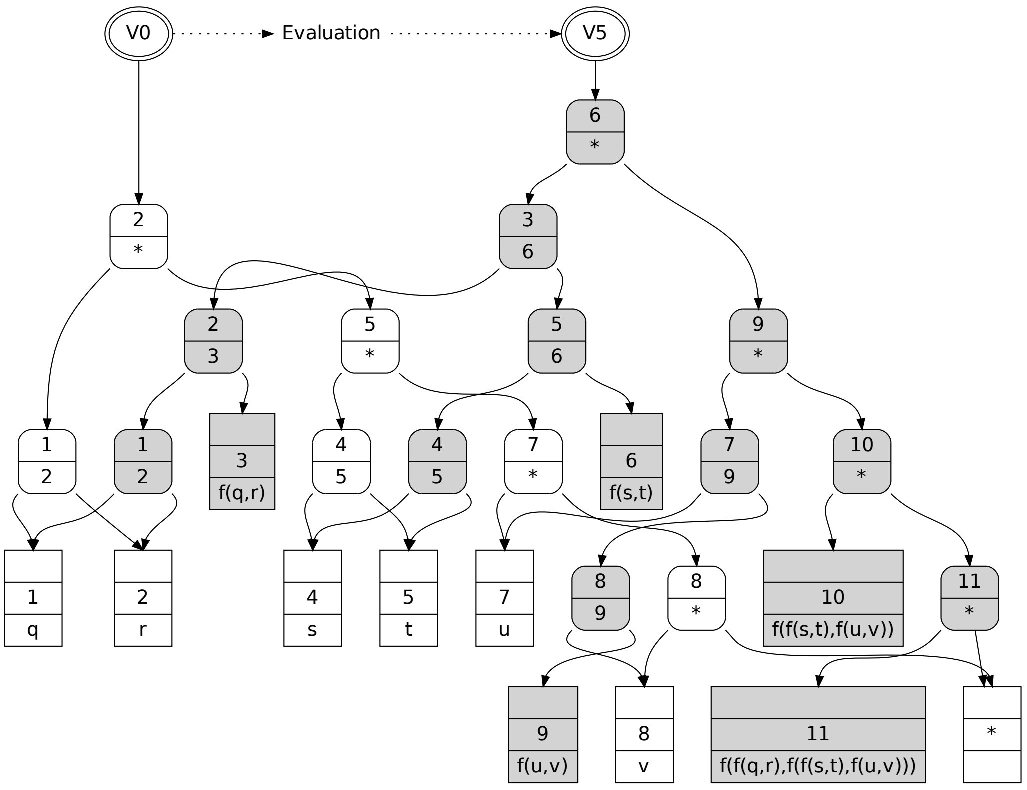

Figure 6.3: **An Immutable Data Structure representing the evaluation of an expression.** The Immutable Data Structure representing the evaluation of the expression $f(f(q,r),f(f(s,t),f(u,v)))$ is an interval tree, with a sentinel, which maps the tag number of a value in the abstract syntax tree to a leaf. The Immutable Data Structure contains all of the arguments and intermediate values as well as the result. Only the initial and final versions of the Immutable Data Structure are shown. The final version is shaded.

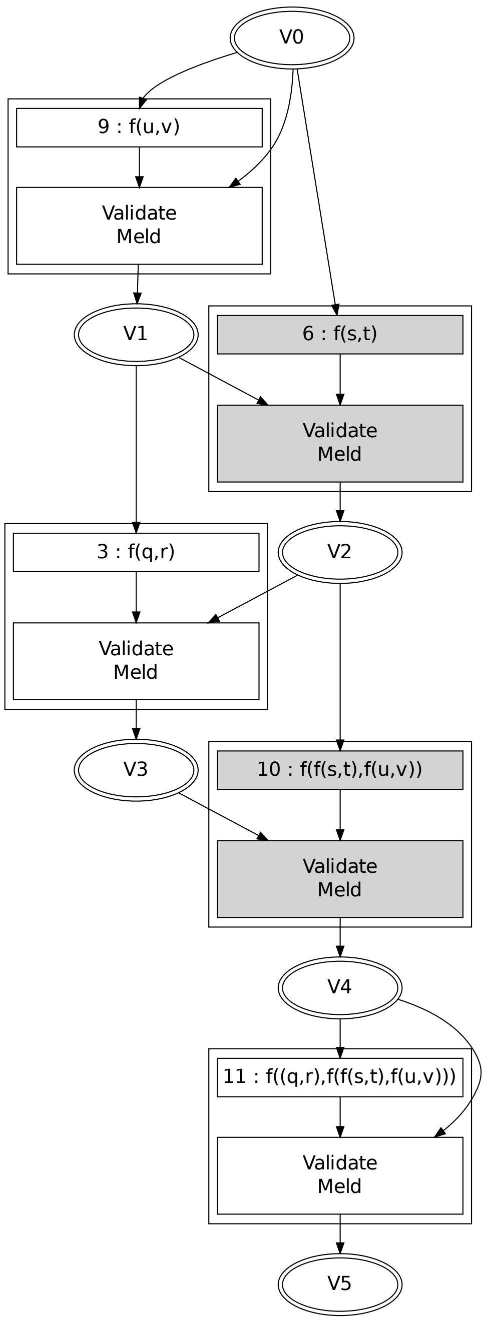

Figure 6.4: **The non-blocking evaluation** of the expression $f(f(q,r), f(f(s,t), f(u,v)))$ by multiple processors load-balances the work between them. Each operation takes a version of the data structure, represented by an ellipse, as its argument and produces a new version. The operations executed by another processor are shaded. Only validation against the first snapshot version is show.

Bibliography

[AF92] Juan Allemany and Ed Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134. ACM Press, August 1992.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[GBB$^+$06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice.* Addison-Wesley Longman, Amsterdam, 2006.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.