A Simple Proof System for Lock-free Concurrency

Luís Caires[1], Carla Ferreira[1] and António Ravara

CITI and Departamento de Informática, FCT
Universidade Nova de Lisboa

1 Motivation

The use of locking is the classical approach to prevent interference in concurrent access to shared data structures. Using locks it is not difficult to ensure data consistency (a safety property) and deadlock-freedom (a liveness property). Many sophisticated implementation and reasoning techniques for lock-based concurrent programming have been developed over some decades. Lock-based synchronization has however the drawback of reducing the opportunity for concurrent execution, a limitation that may hinder the most effective use of intensive multiprocessor hardware, commonly available today. As an alternative, programming styles based on "non-blocking synchronization" have emerged [HS08]. In this context, alternative synchronization / cooperation techniques have been proposed, that ensure correctness (in particular data consistency) of shared resources even when they are used by multiple threads, and avoid pessimistic locking protocols. Locking protocols rely in general on the established abstraction of critical region. In the critical region world, shared resources are only accessible inside a delimited critical region in the program code. Critical regions are protected by locks, ensuring that only a single thread may use the shared resource at a given time. Threads then must queue to access the lock, creating contention and hindering concurrency. On the other hand, optimistic algorithms using non-blocking synchronization usually rely on transaction-like abstractions. In this case, several threads may engage into a transaction, and only at commit time conflicts (*e.g.*, arising from races) are detected, causing concurrent transactions to either abort or commit. This latter model favors concurrency, if the "abort" rate turns out to be negligible. This general approach has been successfully encapsulated in the STM [HS08] abstraction, that has deserved much attention recently.

2 Aim

Several works have already successfully addressed the issue of finding proof system expressive enough to reason about "non-blocking synchronization", usually building on programming idioms using the CompareAndSwap (CAS) primitive [HS08], which perhaps does not reveal the optimistic transaction analogy so clearly as we would like. In this work, we are instead exploring the analogy between the transaction abstraction and programming idioms building on the LoadLink/StoreConditional (LL/SC) pair of primitives [HS08], where we would like to interpret the LoadLink processor instruction as the transaction start, and StoreConditional processor instruction as the transaction commit, returning either "success" or "abort". This analogy is not so clear when the CAS primitive is used. The LL/SC primitives turn out to be encodable using more common atomic operations such as CAS [JP05], even if already available as primitives in some processors (*e.g.* MIPS, Alpha). Recently several works have addressed the design of proof systems for modular verification of concurrent programs, both lock-based (e.g. [O'H04]) and lock-free (e.g. [GCPV09]), usually building on separation logic, or combinations of rely guarantee with separation logic [Vaf08]. In this work, we have been designing a Hoare-Separation-style proof system to modularly prove correctness of lock-free algorithms based on the LL/SC model, where we are particularly interested in data consistency, and possibly deadlock-freedom.

3 Approach

We illustrate our approach with an example: proving correctness of a program to insert new nodes in a shared linked list. The program, listed below, repeatedly tries to insert a newly created node in a shared list. We can imagine the body of the while loop as a transaction that either commits when `SC` returns true, or aborts otherwise. In that case, the transaction needs to be retried. More concretely, the loop starts by executing operation `LL`, which receives the address of memory location x (global variable) and assigns its contents to y (local variable). Next, the new node w is inserted at the top of list y. Last, operation `SC` will try to store w in memory location x: if location x has not been written by any concurrent thread since `LL`, the new value is written atomically to location x and returns true, according to the standard semantics of `LL`/`SC`. We show that if the loop eventually terminates, then the insertion succeeded, and the data structure is still a list (the one resulting from the initial list by inserting a new node). We list the program code, together with the relevant assertions.

```
                                                        $insertNode\{$
$\{list(*x)\}\{emp\}$                                      $z = \mathtt{F}$;
$\{list(*x)\}\{z = \mathtt{F} \wedge emp\}$                while($not\ z$) {
$\{list(*x)\}\{z = \mathtt{F} \wedge emp\}$                    $y = \mathtt{LL}(x)$;
$\{list(*x)\}\{z = \mathtt{F} \wedge list(y)\}$                $w = \mathtt{new}(y)$;
$\{list(*x)\}\{z = \mathtt{F} \wedge (w \to y * list(y)\}$
$\{list(*x)\}\{z = \mathtt{F} \wedge list(w)\}$                $z = \mathtt{SC}(x, w)$
$\{list(*x)\}\{z \Rightarrow emp\}$                        }
$\{list(*x)\}\{z = \mathtt{T} \wedge emp\}$
$\{list(*x)\}\{emp\}$                                   $\}$
```

Our triples have the general form $\{G\}\{L\}\ S\ \{G'\}\{L'\}$ where G, G' describes invariants over global state, and L, L' the local state (local view) of the thread S under consideration. All formulas are written in separation logic [Rey02, O'H04], although interference is allowed between G and L, and between G' and L' as well (so this differs from the notions of local and shared state of RGSep [Vaf08]). The effect of `y=LL(x)` is to create a local snapshot of a shared data structure referenced from a given memory cell, by instantiating the global invariant. In the example, $list(*x)$ is copied to the local view as $list(y)$. A challenge posed to the proof system is then to ensure that if the anchor variable $*x$ is not changed then the global invariant is preserved, even in the presence of interference. Another challenge is to deal with multiple interleaved or nested transactions. The effect of `z=SC(x,w)` is to enforce a well defined state (emp in the example) only when the "transaction" succeeds. Notice that the while loop ensures that the control variable z is set to `T` at exit, so that the desired post-condition emp is established (meaning that the local view is lost to the global state).

In the talk, we will present a core imperative calculus, equipped with concurrency and LoadLink/StoreConditional (`LL`/`SC`) primitives. We define the structural operational semantics and a proof system of this small language, prove its soundness, and discuss some simple examples.

References

[GCPV09] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 16–28, New York, NY, USA, 2009. ACM.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[JP05] Prasad Jayanti and Srdjan Petrovic. Efficient wait-free implementation of multiword ll/sc variables. In *ICDCS*, pages 59–68. IEEE Computer Society, 2005.

[O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS02*, pages 55–74. IEEE Computer Society, 2002.

[Vaf08] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, 2008.