Towards Structured Mechanized Verification of Fine-Grained Concurrent Programs

Extended Abstract

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Anindya Banerjee
IMDEA Software Institute
anindya.banerjee@imdea.org

It has been long recognized that efficient parallelization is of crucial importance for high-performant software. Unfortunately, reasoning about correctness of concurrent programs, in which several computations can be executed in parallel and, thus, overlap in time, is challenging due to the large number of possible interactions between concurrent processes/threads on shared data structures.

One way to deal with the complexity of verifying concurrent code is to employ the mechanisms of so-called *coarse-grained* concurrency, *i.e.*, *locks*. By making use of locks in his code, the programmer ensures *mutually-exclusive* access to critical resources, so that at most one thread will be accessing the shared data structure at the same moment of time, therefore, reducing the proof of correctness of the concurrent code to the proof of correctness of the *sequential* code. While sound, this approach to concurrency prevents one from taking full advantage of parallel computations. An alternative is to implement shared data structures in a *lock-free* (*i.e.*, *fine-grained*) manner, so the threads manipulating such structures would be reaching a consensus through the active use of non-blocking read-modify-write operations (*e.g.*, CAS) instead of locks.

Despite the clear practical advantages of the fine-grained approach to the implementation of concurrent data structures, it requires a significant expertise to devise such structures and establish their correctness with respect to standard criteria, such as *linearizability* [2, 12]. Even worse, in the presence of the advanced programming features, such as higher-order functions, pointer aliasing, and internal parallelism, linearizability by itself might not be strong enough as a criteria to ensure the full functional correctness of a concurrent program, and generalizations of linearizability to match these programming features are a topic of an ongoing active research [3, 11].

An alternative approach to specify the behaviour of imperative, and, in particular, concurrent programs as well as to verify them is to make use of *program logics*. In such logical frameworks, program specifications are represented by means of *Hoare triples* $\{P\}\ c\ \{Q\}$, where c is a program being specified, P is a *precondition*, *i.e.*, a predicate that describes a state, in which the program should be run, and Q is a *postcondition*, which describes a state upon the program's termination. The logic-based approach has a number of appealing characteristics. First, checking the program correctness in a Hoare-style program logic can be done *structurally*, *i.e.*, by means of systematically applying syntax-directed inference rules, until its specification is derived. Second, verification via a program logic is *compositional*: once a library procedure is verified against a suitable specification in terms of pre- and postconditions, its code is not required to be re-examined ever again: all reasoning about the client code that makes use of that procedure can be carried out solely out of the procedure's specification, which is coherent with good practices of code encapsulation. Finally, modern program logics are sufficiently *expressive* in the sense that they can give specifications to the programs that operate with first-class executable code, locally-spawned threads and other features inherent for modern programming languages.

Since the process of structural program verification in a Hoare-style logic is largely mechanical, there has been a number of research projects recently, targeting to mechanize and automate it for *sequential* and *coarse-grained concurrent* programs by means embedding it into a general-purpose proof assistant, *e.g.*, Coq [4, 17], or implementing a standalone verification tool [14].

The situation is different with the logic-based verification of *fine-grained* concurrency, which requires reasoning about a number of concepts, that don't have direct analogues in the sequential or coarse-grained concurrent programming:

- **Custom resource protocols.** Each data structure (*i.e.*, *resource*), which can be used by several threads concurrently, needs to be supplied with a specific definition of an "evolution protocol", in order to enforce the preservation the structure's internal consistency. In contrast with the coarse-grained case, where the protocol is fixed and corresponds to locking/unlocking, fine-grained data structures require defining custom protocols with multiple states and transitions.
- **Interference and stability.** For the sake of local reasoning about the shared data structure's state from a single thread's perspective, one should formalize what are the admissible changes that can be made by other threads, *interfering* with the current one. That is, every thread-local assertion about the structure's state should be *stable*, *i.e.*, invariant under possible concurrent modifications.
- **Work stealing** is a common concurrent pattern, appearing in fine-grained programs due to relaxing the *mutual exclusion* policy, so several threads can simultaneously operate with a single shared resource. The "stealing" happens when a thread is scheduled for a particular task involving the resource, but the task is then is accomplished by *another* thread; however, the "contribution" is nevertheless ascribed to the initially assigned thread.

In addition, Hoare-style reasoning about coarse- or fine-grained concurrency requires a form of **auxiliary state** in order to partially expose the internal threads' behavior and relate local program assertions to global invariants, accounting for specific threads' contributions into a resource.

These four aspects, paramount for Hoare-style verification of fine-grained concurrent programs, have been recognized and formalized in one form or another in a series of recently published works by various authors [6–9, 13, 15, 18–21], providing logics of increasing expressivity and compositionality. However, in the formal proofs for concurrent libraries, performed in all these logical systems, most of the complexity comes *not* from the libraries' size in terms of lines of code, but from the intricacy of the corresponding data structure invariant and the presence of thread interference and work stealing. In particular, the later fact, in contrast with proofs about sequential and coarse-grained concurrent programs, requires one to establish stability of *every* intermediate assertion in the program verified. Needless to say that in such setting paper-and-pencil verification of fine-grained concurrent programs becomes a highly challenging and error-prone task, as it's too easy for a human prover to overlook a piece of resource-specific invariant or an assertion, unstable under interference, hence, rendering the whole reasoning unsound.

In this work, we present a framework for *mechanized* specification and verification of fine-grained concurrent programs based on the recently proposed *Fine-grained Concurrent Separation Logic* (FCSL) by Nanevski *et al.* [16].[1] FCSL is implemented as a library and an embedded domain-specific language (DSL) in the dependently-typed language of the Coq proof assistant [5]. Due to its design and the choice of a logical foundation, FCSL, as a verification tool for fine-grained concurrency, is:

- **Uniform:** FCSL's specification model is based on two basic constructions from computer science: *state-transition systems* (STSs) and *partial commutative monoids* (PCMs). The former ones are used for describing concurrent protocols and thread interference, whereas the later ones provide a generic treatment of shared resources and thread contributions, making it possible to encode, in particular, the work stealing pattern. In the talk, we will demonstrate how these two components are sufficient to specify a large spectrum of concurrent algorithms as well as to make the proofs of verification obligations to be uniform.
- **Expressive**: FCSL's *specification* fragment is based on the propositional fragment of Calculus of Inductive Constructions (CIC), which makes it capable of expressing a large variety of mathematical theories (*e.g.*, graphs, arrays, event traces, *etc*) as well as to encode a number of algebraic structures, all occurring in the program specifications in one way or another.
- **Realistic:** In addition to the ability inherited from CIC to employ higher-order predicates in the specifications, the *programming* fragment of FCSL features a complete toolset of modern programming abstractions, including user-specified algebraic datatypes, higher-order functions and pattern matching, since *any* Coq program is also an FCSL program. The monadic nature of FCSL's embedding into Coq makes it possible to encode a number of computational effects, such as thread spawning and general recursion. All this makes writing programs in FCSL to be very similar to programming in a higher-order language, such as ML or Haskell.
- **Foundational:** The soundness of FCSL as a logic has been proven in Coq with respect to a version of denotational semantics for concurrent programs in the spirit of Brookes [1]. Moreover, thanks to the design choice of representing FCSL specifications as mere Coq types, the soundness result scales to the *whole* language of Coq, not just a toy "core" calculus, therefore, ensuring the absence of bugs in the whole verification tool and, as a consequence, in any program, which is verified in it.

Structure of the talk

In the proposed talk, we will introduce the FCSL verification framework by example, specifying and verifying full functional correctness of a characteristic fine-grained program—concurrent graph-manipulating algorithm. Starting from the basic intuition behind the formalization of the algorithm and its specification, we will demonstrate the common stages and patterns of structuring program verification in FCSL. While doing so, we will show how the verification process, while not fully automated at this stage, nevertheless does not put a lot of proof burden to the human prover, thanks to the fact that, as an embedded DSL, FCSL inherits a number of features from the common infrastructure of Coq, enhanced by Gonthier *et al.*'s Ssreflect extension [10]: dependent records, boolean reflection and canonical structures. We will next elaborate on some important design choices made in the implementation of FCSL with respect to encoding of concurrent protocols and program specifications. We will then report on our experience of verifying in FCSL a number of benchmark concurrent programs and data structures: locks, simple memory allocator, concurrent stack and its client programs, atomic snapshot, non-blocking universal constructions, and graph-manipulating algorithms.

[1] Hereinafter, we will be using the acronym FCSL to refer both to the Nanevski *et al.*'s logical framework and to our implementation of it.

References

[1] S. Brookes. A semantics for concurrent separation logic. *Th. Comp. Sci.*, 375(1-3), 2007.

[2] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *PLDI*, 2010.

[3] A. Cerone, A. Gotsman, and H. Yang. Parameterised linearisability. In *ICALP*, 2014.

[4] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[5] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version V8.4*, 2014. `http://coq.inria.fr/`.

[6] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, 2014.

[7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, 2010.

[8] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[9] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

[10] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Technical Report 6455, Microsoft Research – Inria Joint Centre, 2009.

[11] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.

[12] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3), 1990.

[13] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

[14] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM*, 2011.

[15] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[16] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*, 2014.

[17] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, 2010.

[18] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, 2014.

[19] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.

[20] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

[21] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.