Deductive Verification of Real-World C++ Weak-Memory Programs

Pascal Wiesmann

November 3, 2018

1 Introduction

The performance that can be achieved by using low-level concurrent C++ is often desired in library-programs. For the sake of performance, the memory model that was introduced in C++11 provides very weak guarantees. This is the reason why it is called *weak memory*. Weak memory programs are very hard to write, as well as to test, for at least two reasons. First, the weak memory semantics allow many different executions of the same code. Second, some of these executions might be extremely rare and it would be nearly impossible to observe them using random test-cases. However, such low-level libraries are not only hard to write and to test, it is also absolutely crucial that they are reliable, because they are used in many complex software projects. Therefore, not even extremely rare failures are acceptable. Subtle bugs in the libraries may affect all of these projects in ways which are hard and frustrating to debug. In practice, these libraries are tested extensively, which is very time-consuming. Still, one can not be 100% sure that the program always is correct. Formal proofs, on the other hand, are also expensive, but they provide something very desirable which is usually impossible to achieve with testing: a guarantee that the program is correct for *all* scenarios.

2 Existing Work and its Limitations

2.1 Weak Memory Logics

Several logics for reasoning about weak memory programs have been proposed. To our knowledge, the main ones are RSL [7], FSL [2] and FSL++ [3] (both extensions of RSL) and GPS [6]. We focus on the RSL logics, but might benefit from some ideas from GPS. RSL and FSL are easier to automate than FSL++, but they are also much less expressive. In order to encode real world programs, a logic with a similar expressiveness to that of FSL++ is required. The problem with FSL++ is that the high flexibility of the encoding makes it hard to automate the proofs. Attempts to define a logic that is possibly slightly less

flexible than FSL++, but makes automation easier, have been made [4]. This work provides valuable insights, but it is not yet a finished proposal, leaving several research questions and design decisions which mus be addressed before being ready to use in a verifier.

2.2 Automation

A prototype tool for an automated verification of weak memory programs has been published [5]. It encodes large fractions of RSL and FSL into Viper, a language for which a reliable and established verifier exists. However, RSL and FSL only allow reasoning about individual atomic accesses in isolation. This is often not sufficient for the verification of real-world programs, as argued in [3]. The reason is that identical operations can have different significance in a particular library implementation, depending on what happened earlier in the program. In FSL++, ghost state allows to artificially remember what happened earlier in the program, even when it is not possible to determine it by looking at the program state.

3 Core Goals

Preparation:

- Manually encode proofs from the extended FSL++ described in [4] into Viper in order to understand their potential for automation.

Define an expressive and automatable logic for weak memory programs:

- Permission structures: Proofs for weak memory programs are mostly about permissions. A straightforward approach for reasoning about permissions is with fractional permissions. At each point in a program, the thread that is executing the current statement has a certain amount of permission ($\in [0, 1]$) to each memory location. A permission > 0 corresponds to the permission to read from the memory location. The full permission ($= 1$) corresponds to the permission to write to the memory location. Fractional permissions are suitable for automation, which is why they are used in the Viper verification framework. However, they make it hard to prove properties of concurrent programs, as explained in [1]. One issue is that an unlimited amount of threads should be allowed to read from a memory location simultaneously. Additionally, each reader thread should itself be allowed to spawn an unlimited amount of reader threads. If the amount of permission that is given away to a reader thread is fixed, the amount of reader threads is limited and reader threads cannot spawn additional reader threads. If, on the other hand, the reader threads do not all get the same permission, it becomes hard to find a mapping between the amount of permission that is held, and the count of readers that have

been spawned. In order to deal with this complexity, FSL++ allows the usage of custom partial commutative monoids for permissions. For a verifier, supporting arbitrary monoids does not seem feasible because different monoids require different custom decomposition/recomposition operators on the verifier state. Also, in order to use Viper as a backend, there needs to be a mapping between the custom monoids and the supported monoids in the Viper language. If it is not possible to do so, suitable features have to be designed and added to the Viper language. Which permission structures are most fitting for usage in a C++ verifier is an open problem which will be tackled during this thesis.

- Transition invariants: FSL++ potentially leaves many choices for proof steps that are applicable at a certain position in the program. Often, the proofs involve sophisticated manipulation of ghost state, which an automated verifier cannot be expected to come up with. Fortunately, these manipulations are usually not arbitrary. They typically reflect the modeling of a concept relevant to the data structure in question; the concept and appropriate rules can be defined once and then stay the same for the whole program. In [4] it is argued that these concepts can be captured by transition invariants. These specify which of the possible proof steps has to be applied depending on the current state. Transition invariants have been proposed in [4]. At first sight, they seem suitable for encoding into Viper.

Automate proofs in this logic using Viper:

- Find suitable Viper encodings for the added logic features.

Implement a Front-End tool for Verification of C++ programs:

- Support actual C++ syntax: A minimal subset of C++ has to be defined for which a parser will be implemented. Hopefully, large parts of an existing C++ parser can be reused.
- Define annotation syntax: It has to be possible to write the annotations directly into the C++ code. These will also need to be parsed.
- Translation to Viper: Once an intermediate representation of the program and its annotations is available, it will have to be encoded into Viper.
- Report results of verification: One of the verifiers (Silicon or Carbon) for the Viper language will be invoked, and as the verification results sent back to the user. The errors will be shown to the user, but in a way that he does not need to know about the Viper backend (e.g. line numbers should correspond to lines in the C++ code).
- Tool infrastructure (e.g. VSCode plugin): The tool has to be easy to use. This is probably best achieved if it is integrated into some text editor or IDE in form of a plugin.

Evaluate using real world examples.

- Show non-trivial code examples which the verifier handles successfully.
- Measure the run-times on examples of different sizes.

4 Extension Goals

- Extend the subset of C++ that is supported.
- More substantial evaluation:
  - Take new examples from libraries and verify them.
  - Describe the limitations of the implemented tool and explain why they are hard to overcome.
- Prove the soundness of the used logic: The soundness of the RSL logics has been proved using Coq. It would make sense to also prove the soundness of the logic we will define during this thesis. Rather than using Coq as well, we might prove our logic using the RSL logics.
- Define equality for predicates in Viper in order to extend support for fences to arbitrary resources (predicates): The technique used in the prototype verifier to encode fences is probably going to be reused in this thesis. It does not work for arbitrary resources and could be further improved.

References

[1] John Tang Boyland, Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Constraint semantics for abstract read permissions. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

[2] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 413–430, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[3] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 448–475, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[4] Gaurav Parthasarathy. Applying and extending the weak-memory logic FSL++ (research in computer science project). 2017.

[5] Alexander J. Summers and Peter Müller. Automating deductive verification for weak-memory programs. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 190–209, Cham, 2018. Springer International Publishing.

[6] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. *SIGPLAN Not.*, 49(10):691–707, October 2014.

[7] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. *SIGPLAN Not.*, 48(10):867–884, October 2013.