Reasoning about Memory for Program Correctness
Research Statement

William Mansky

1 Motivation: Improving Software Correctness

Modern code is plagued by bugs, both in the code itself and the compilers that translate it into instructions for multiprocessors. Even GCC, the compiler that millions of people worldwide rely on to execute their code, has been shown to have a large number of bugs [12], and even more when compiling concurrent programs [9]. Formal verification research promises to solve this problem by providing strong assurances of program correctness, and starting with CompCert [5], the verified C compiler, recent projects have shown that formal methods are ready to be applied to large-scale systems software. However, there are a few major theoretical gaps that prevent formal methods from being widely used to verify real-world code, and especially real-world concurrent code. In my thesis [6], I proposed a technique for closing these gaps, by decomposing the problem of program verification into reasoning about code and reasoning about memory. This approach promises to improve the reusability of proofs about programs, allow target-independent and memory-model-independent reasoning, and make the complexities of concurrent memory models comprehensible to non-experts. By pushing this approach forward, improving our formal models for memory and concurrency, and developing new techniques for reasoning about programs and program transformations involving them, I aim to equip verification experts and programmers alike with the tools they need to understand the trickiest aspects of modern code.

Writing correct code is both difficult and critical. Almost everything now relies on software, but popular programming languages are full of corner cases and unexpected behaviors, and even extensive testing may not reveal potential defects. To make matters worse, correctly written programs may still run incorrectly due to errors in the compilers that translate the programs into the machine code that is actually executed. In the quest for higher performance, architects and language designers come up with ever more aggressive optimizations and, more recently, have shifted to emphasizing concurrent execution and relaxed memory models, all of which make it even harder to understand and reason about the actual behavior of executing programs. Multicore processors and relaxed memory models are now ubiquitous, and we cannot hope to produce correct software without a better understanding of their principles. Formal methods and programming language theory offer the tools we need to provide strong guarantees of correctness for real-world programs, whether security-critical (car or airplane control systems, medical devices) or simply useful and widely used (operating systems, C compilers).

Background: Language Semantics and Formal Methods
Formal methods and programming language theory give us a few vital tools for guaranteeing correctness of software. First and most important is the ability to define correctness: we must have a clear and quantitative description of the desired behavior of a system, whether in terms of functional correctness (this program performs the following computation) or avoidance of some class of errors (this program will not access protected memory). Second is the semantics of languages: we must know precisely what code actually means, whether via a description of its execution (operational semantics) or in terms of the mathematical functions it models (denotational semantics). Finally, we have theorem provers, the tools and IDEs that allow us to construct machine-checked formal guarantees for programs and program transformations. The combination of these elements has produced remarkable advances in high-assurance software, including the CompCert verified C compiler [5], the seL4 operating system kernel [4], and Verdi [11], a system that automatically transforms distributed protocols to protect against network faults. Thanks to improvements in software models and reasoning techniques,
software verification is finally starting to catch up with software, and I intend to continue to close the last few gaps between proved-correct models and real-world code.

2 Past and Current Research: Reasoning about Memory and Concurrency

My research to date has focused on two of the most difficult aspects of modern software: memory and concurrency. In graduate school, I began by studying the problem of verifying compiler optimizations, and then looked into what it would take to write similar optimizations and proofs for concurrent programs and relaxed memory models. As a postdoc, I worked on abstractions for reasoning about memory, and looked for reasoning principles that hold across a range of memory models.

Verifying Compiler Optimizations My first project as a graduate student [8] was to give formal semantics to TRANS [2], a language for specifying compiler optimizations, using the Isabelle theorem prover. TRANS provided a concise and innovative way for compiler designers to write optimizations, and in combination with semantics in a theorem prover allowed for the production of compiler optimizations with strong guarantees of correctness. I used the language to specify and verify a translation into Static Single Assignment form, a common transformation in optimizing compilers. Later, I returned to TRANS with the goal of extending it to apply to concurrent languages. I began with the same intuitive semantics for memory that many programmers have, modeling memory as a global map from locations to values and using sequential consistency as my notion of concurrency. However, I soon learned that the biggest problem with writing correct concurrent code comes from relaxed memory models, which exhibit more complicated memory behavior.

Relaxed Memory and Compositional Verification After spending some time studying compilers for relaxed memory models, including CompCertTSO [10], the relaxed-memory extension of the CompCert compiler, I created VeriF-OPT [6], a verification framework for compiler optimizations on concurrent programs. VeriF-OPT built on the potential of TRANS as a tool for developing high-assurance compiler optimizations. A user can specify a candidate optimization in PTRANS, a parallel extension of TRANS, and run the optimization on sample programs, refining the conditions under which the optimization should be applied. Once the optimization seems to be correct, the designer can pass it off to a formal verifier, who can use Isabelle along with a library of compiler-related lemmas to formally prove the correctness of the optimization. One of the key features of the framework, intended to reduce the effort involved in verifying each new optimization, is its approach to specifying intermediate languages: the semantics of memory are separated from the semantics of the language itself. This approach made it easy to specify two base languages and instantiate them with a range of concurrent memory models. The ultimate goal was to minimize the effort involved in verifying a new compiler optimization by factoring out 1) elements common to a single language under different memory models and 2) elements common to multiple languages under the same memory model. This brought up an important challenge: there are very few facts that are known to hold across multiple memory models. Even so, this approach—decomposing the definition of a language into language and memory components, and using this decomposition as a path to compositional verification—proved useful.

Reasoning about Memory After graduating, I took a postdoc position to help redesign Vellvm, a formal semantics for LLVM, with an eye towards concurrency. One of my first projects was to test a C-like memory model that could be used to reason about pointer arithmetic [3], and verify several program transformations under the model. Inspired by this work, I examined the sequential memory model used by CompCert and LLVM, and developed an abstraction over sequential memory models that allowed for more platform-dependent behavior than the previous version while still providing strong enough guarantees to verify optimizations [7]. Most recently, I’ve returned to concurrent memory models, and am currently developing a relaxation of the widely-used DRF guarantee, that race-free programs have sequentially consistent
behavior, to well-synchronized sections of racy programs. I am also in the process of using my formalization of relaxed memory models to verify a simple instrumentation pass for dynamic race detection.

3 Future Research: Taming Relaxed-Memory Verification

Moving forward, I plan to build on my work on sequential and concurrent memory semantics on two fronts: expanding the range of language features that can be captured and the number of reasoning principles available, and using the techniques of abstraction and compositional reasoning to build tools for verifying complex concurrent programs.

Sequential Memory Principles In the short term, I would like to develop a more extensive generic specification of sequential memory models. By starting with my axiomatic specification of sequential models [7] and adding features such as pointer casts and structured data, I aim to make it possible to characterize and reason about memory behavior and memory safety across different languages, compilers, and platforms. Concurrent memory models are often written as if the semantics of sequential memory is obvious, and the only interaction between memory is in the form of reads and writes to distinct locations. In practice, memory-related operations in programs can include allocation and deallocation, casts between pointers and data, manipulation of structured data, etc. My previous work on sequential models allowed for memory management operations, but not for structured data or complex pointer-based reasoning. Pointer-data casts, in particular, are assigned a wide range of behaviors across different languages, compilers, and platforms, and I would like to come up with a specification that admits as many of these behaviors as possible while still providing enough guarantees to reason usefully about programs with casts (without dropping down to the underlying implementation). It seems reasonable to expect that, for instance, using addition to index into an array within the array’s bounds will have the same semantics across all models, even if an access outside the bounds may have dramatically different behavior. This may even lead to a language-independent characterization of the set of memory-safe programs, programs that do not perform unauthorized pointer operations or otherwise rely on platform-specific behavior.

Verified Relaxed-Memory Compilers Once the basic properties of memory models are established, I would like to put my memory abstractions to use by creating an improved version of CompCert for relaxed memory models. The proofs in CompCert were done against a fairly restrictive (and purely sequential) model with a variety of simplifying assumptions. The CompCertTSO project [10] extended CompCert with thread creation operations, used Total Store Order as its concurrent memory model, and removed some of the simplifying assumptions. Importantly, it also decomposed the semantics of memory from the semantics of individual threads. Given a good generic interface for relaxed memory models, it should be possible to use CompCertTSO as a basis for a verified compiler with pluggable memory models, which automatically performs only the optimizations that have been proved correct under the requested model or class of models. CompCert and CompCertTSO include very few optimizations that change the memory behavior of programs, even under the simplest memory models, so I would need to integrate some such optimizations in order for the differences between models to become apparent. CompCertTSO also relies on an operational characterization of TSO, while most relaxed models are given axiomatically, in terms of predicates on complete executions, so a fully generic relaxed-memory CompCert would require either overcoming this restriction or finding operational semantics for broader classes of models (such as the family of models described in the work of Alglave et al. [1], which includes TSO, Power, and Alpha). A proved-correct compiler that could be set up to translate from, for instance, C under the full C11 memory model to ARM machine code under ARM’s memory model would be a boon to everyone who writes, reasons about, or struggles to debug real-world concurrent code.

Compositional Reasoning for Relaxed Memory In the long term, I would like to significantly expand our understanding of the properties that hold across relaxed memory models, and use this to extend the
approach of VeriF-OPT [6] to a truly compositional framework for verifying program transformations under relaxed memory models. Thus far, reasoning about concurrent programs has been hindered by the sheer range of relaxed memory models – every processor architecture and every major imperative language has its own, not-obviously-comparable concurrency semantics. My work with extending the DRF guarantee suggests that there are more reasoning principles yet to be found that are supported by all common models. Furthermore, there are also large classes of models with similar properties: for instance, the class of concurrent memory models that allow reads to be reordered past writes. I would like to create a complete hierarchy of properties of memory models, from properties that hold for all models (the DRF and critical-section guarantees) down to properties on small classes of related models (such as TSO and PSO). With a rich collection of cross-memory-model properties, we can finally begin to do truly compositional reasoning on concurrent programs and program transformations, separating program behavior from memory behavior and cleanly characterizing the conditions (and thus determining the models) under which transformations are correct. Given such a hierarchy, programmers and compiler designers working in C, for instance, would no longer have to guess and check whether their designs work under the C11 relaxed memory model; they would have direct access to a set of principles for justifying their programs.

The science of software verification is ready to tackle large-scale systems, but limitations in formal models of memory and concurrency behavior limit the reach of current approaches. I plan to tackle these last major obstacles by developing theories and techniques that make memory and concurrency tractable. Improved formal models will allow formal methods researchers to more easily verify a wider range of programs and program transformations, but even more importantly, they will lead to abstractions that can be used by ordinary programmers to understand the behavior of concurrent programs. Ultimately, a better understanding of memory semantics, both sequential and concurrent, will lead to more comprehensible guarantees to programmers and better tools for compiler designers and program verifiers, resulting in more correct and reliable code in modern software of all kinds.

References


