Building Correct Programs
Research Statement

William Mansky

1 Motivation: Do You Trust Your Code?

The vast majority of programs are written in imperative languages, requiring careful manipulation of data in memory and synchronization between threads. This has inevitably led to the proliferation of bugs—missed corner cases like buffer overflows and segmentation faults, logic errors arising from the gap between the programmer's goals and the code used to realize them, and compiler errors that introduce bugs into correctly written programs. Techniques for conclusively demonstrating the absence of bugs have been known since the 1970s, and in the past decade they have been successfully applied to code of all sorts—compilers [8], distributed systems [20], highly concurrent algorithms [19]. The fundamental principle of these techniques is to express both the effects of code and the high-level goals of programs mathematically, and then to prove that all possible executions of a program match the desired behavior. I am interested in finding ways to model and reason about those aspects of programming languages that still resist formal analysis, in developing code that comes with the strongest possible guarantee of correctness, and in using ideas from formal verification to make it easier to write correct programs in the first place. My long-term goal is to make verification powerful and accessible enough that it is feasible to prove the correctness of most real-world programs.

Background: Modeling, Specification, and Verification To prove the correctness of a program, we must answer three questions. First, what does the program do? Second, what should it do? Third, how do we know that the two coincide? The first question is answered by *semantics*, a mathematical model of the behavior of programs in some particular language. A semantics for a language like C includes a logical model of the machines on which C executes, including memory layout and thread pool, as well as a precise description of the effects of each C command. The second question is answered by a *specification*, a precise logical description of the program's desired effects. This description is generally phrased in a program logic: for instance, separation logic can be used to describe the way that structures in memory are transformed by a function. The final question is answered by a *proof*: if semantics and specification are defined in a common logical language, then the rules of that logic can be used to show that every behavior of a program matches its specification. Because programs are often large, complex, and regularly structured, so likewise are their proofs of correctness, and we need tools and techniques to manage our proofs, check our work, and even produce parts of proofs automatically. Using proof assistants such as Coq [16] and Isabelle [17], we can build scripts of *tactics* that describe the steps of a proof; the proof assistant checks that each step is logically valid and guarantees that we end up with a mathematically sound proof of whatever property we originally stated. As long as the semantics is correct and the specification is the one we meant to prove, this process produces the strongest possible guarantee that the program will always behave as intended.

2 Past and Current Research

In my research to date I have developed tools for writing correct programs, infrastructure to facilitate proving correctness of certain classes of programs, and theory needed to reason about programs that have traditionally been beyond the reach of verification.

Verifying Compiler Optimizations As a graduate student, I focused on developing techniques for specifying compiler optimizations and proving that they did not introduce bugs in compiled programs. My first project [13] was to give formal semantics to TRANS [5], a declarative language for describing optimizations as transformations on program graphs. In TRANS, the static analyses that are generally used to determine whether an optimization can be applied to a program are replaced by temporal logic formulae on paths through the program, directly stating properties such as "x is not used before it is redefined." By giving formal semantics to the language, we were able to formally prove correctness of optimizations written in this style, so that TRANS could be used to rapidly prototype optimizations and then formally demonstrate that they were safe to apply to any program.

The goal of my later PhD work was to extend the approach of TRANS to concurrent programs. The end result was VeriF-OPT [9], 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 compiler designer could begin by writing an optimization in PTRANS, a parallel extension of TRANS; then run the optimization on sample programs, refining the conditions under which it could safely be applied; and finally hand it off to a verification expert, who could use a provided library of compiler-related lemmas in Isabelle to formally prove the correctness of the optimization. In the latest version of the framework, we replaced PTRANS with Morpheus [14], a more flexible domain-specific language that allows users to break compiler optimizations down into combinations of fundamental operations on control flow graphs.

Reasoning About Memory My work on PTRANS and Morpheus drew my interest to two aspects of programs that significantly complicate program verification: memory and concurrency. As a postdoc at UPenn on the Vellvm project, I helped to redesign existing formal semantics for the intermediate representation used in the LLVM compiler framework [7], focusing on extending it with a concurrency model and developing improved reasoning principles for the interaction between programs and memory. In the process, I helped design and test an extension of the memory model of the CompCert verified C compiler [8] that accounted for casts between pointer and integer values [6], and used to model to verify several program transformations. In an effort to systemize CompCert's memory model (which was also used for Vellvm) and distill it down to the guarantees it provided to programmers, I developed an abstraction over sequential memory models that allowed for more platform-dependent behavior than the previous version while still providing the reasoning principles needed to verify optimizations [12].

Race Detection While working on Vellvm, I also collaborated with from UPenn's Architecture and Compilers Group on building better tools for finding data races in imperative programs. Currently, the most common way to find concurrency bugs in C and Java programs is using dynamic race detectors such as ThreadSanitizer [18], which instrument a program so that any race that occurs when the program is run will be reported to the user. I built the first formal correctness proof of race detection instrumentation [15], proving that the instrumentation was sound and complete and did not change the behavior of the instrumented program. Building on the proof for a toy language, I am in the process of scaling it up to a realistic instrumentation pass for C. I also worked on defining formal semantics for race detection on GPUs, which became the formal backing for a new GPU race detection tool [4].

Verifying Concurrent Programs My main current research project is to develop the necessary techniques for proving correctness of realistic C programs, especially those that use the range of concurrency features introduced in C11 [3]. My work builds on the Verified Software Toolchain (VST) [2], a system for proving correctness of C programs in the Coq theorem prover. The first such system I verified was a lock-free message-passing system used in an automated vehicle [11]. More recently, I developed a new approach to verifying fine-grained data structure implementations that use weak-memory atomics (specifically the release-acquire pattern), and used it to prove correctness of highly concurrent implementations of key-value stores and hashtables [10]. I am also part of the team within the DeepSpec project [1] working on verifying a C implementation of a web server.

3 Future Research: Verified Programs in the Real World

Recent projects (both mine and others) have served as proofs of concept, showing that real-world code can be verified. My next overarching research goals are to apply these techniques where they will be most useful, producing verified artifacts that can be used as reliable components in real-world systems; and to connect the insights into program behavior that we gain from formal verification to the practice of programming, making it easier for everyone to write programs that work properly and carry strong guarantees of correctness.

Verified Highly Concurrent Programs The current expert consensus about relaxed-memory atomics is "don't use them unless you really, really know what you're doing": their effects are too complex and subtle for anyone except experts to reliably write programs that behave correctly in all possible executions, and even experts may struggle without a way of mechanically checking their reasoning. Formal verification offers a chance to make this kind of high-performance concurrent code available to a wider user base, by giving performance-focused programmers the tools they need to prove correctness of complicated concurrent algorithms and data structures. Using the techniques I have already applied to a weak-memory key-value store and hashtable, I plan to develop an entire library of proved-correct fine-grained data structures in C, including stacks, queues, linked lists, and so on. To show that the specifications proved for the data structures are not merely correct but also useful, I will also verify a sample application that uses the data structures, such as a high-performance key-value store. The verified data structures will be usable by any programmer who wants to gain the benefits of concurrency without the dangers of concurrency errors, and the proof development itself will serve as a baseline for future proofs of more complex and efficient concurrent data structures.

Programming as Verification The fundamental technique of separation-logic-based program verification is symbolic execution. To verify a function's specification (a pair of pre- and postcondition), we start with the resources and assumptions described by the precondition, and then step through the function, modifying the resources and assumptions according to the effects of each statement. In an interactive tool such as VST, this process is exposed to the user at each step: at each point in the function, the verifier can see the resources held by the current thread and observe how they are modified, and is required to provide evidence to the proof assistant that invariants are preserved and postconditions met at each step. What if the same information was available to the person writing the program in the first place? What if instead of flying blind, a programmer could see at each line a summary of the resources available and the conditions that needed to be met in order to make the function work as intended? The information gathered during (partial) verification would make it easier to write correct programs, and the process of programming would produce proofs of correctness almost as a side effect. One of my long-term research goals is to create an IDE that makes this possible, integrating programming and proving in a way that makes both tasks easier. This will require new techniques for automatically constructing proofs from code, representing the program state abstractions used in verification so that they are useful for the programmer, and reasoning about the properties of code as it is being written. By allowing the two processes to feed into each other in real time, we can make correct programming easier without significantly changing the process of programming itself, and make it so that principled programming automatically produces proved-correct programs.

Code does not need to be buggy and unreliable. In the past 10 years, the verification of imperative code has gone from a theoretical possibility to an realizable goal; the next step is to make it a common task, requiring the same order of effort as writing a program. By extending the reach of formal models to the kinds of programs used in practice and by integrating verification techniques into programming tools, we can finally produce code that merits the highest degree of trust, and make writing such code achievable for researchers and programmers alike.

References

[1] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 375(2104), 2017.

[2] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[3] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 2011.

[4] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 126–140, New York, NY, USA, 2017. ACM.

[5] Sara Kalvala, Richard Warburton, and David Lacey. Program transformations using temporal logic side conditions. *ACM Trans. Program. Lang. Syst.*, 31(4):1–48, 2009.

[6] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, to appear.

[7] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[9] William Mansky. *Specifying and Verifying Program Transformations with PTRANS*. PhD thesis, University of Illinois at Urbana-Champaign, May 2014.

[10] William Mansky and Andrew W. Appel. Proving atomicity of release-acquire concurrent programs. Unpublished.

[11] William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2017. To appear.

[12] William Mansky, Dmitri Garbuzov, and Steve Zdancewic. An axiomatic specification for sequential memory models. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification*, volume 9207 of *Lecture Notes in Computer Science*, pages 413–428. Springer International Publishing, 2015.

[13] William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 371–386, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. Specifying and executing optimizations for generalized control flow graphs. *Science of Computer Programming*, 130:2 – 23, 2016.

[15] William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. Verifying dynamic race detection. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 151–163, New York, NY, USA, 2017. ACM.

[16] The Coq development team. *The Coq proof assistant reference manual*, 2017.

[17] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[18] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[19] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Proceedings of the 24th European Symposium on Programming (ESOP 2015)*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 2015.

[20] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 357–368, New York, NY, USA, 2015. ACM.