This position paper appeared in the the CAV 2008 Workshop on Exploiting Concurrency Efficiently and Correctly, July 7, 2008.

# Foundational High-level Static Analysis

Andrew W. Appel
Princeton University
appel@princeton.edu

***Abstract.*** *A formal method (e.g., of software verification) is* foundational *if it proves program properties from the axioms of logic and from a low-level machine specification (ISA or transistors). The proofs should be machine-checked, because hand-checked proofs don't track real software systems well. With recent advances on several fronts (in static analysis, semantics, compiler verification) it is now feasible to put scalable, fully automatic program analyses (such as shape analysis of concurrent C programs) on a foundational footing.*

This is an exciting time for the formal verification of software, in part because several threads of research, in progress for decades, have the potential to cohere. These threads include a gradual revolution in the specification methods for operational semantics of programming languages (1994–2008); progress in the specification of weak memory models (1992–2008); steady progress in abstract interpretation (1978-2008); the maturation of mechanized proof assistants (1978-2008) and dependently typed logics (1988-2008); successes in compiler verification (1989, 2006); and finally, enough decades of Moore's law so that the proof assistants and the abstract interpretations finally have usable performance.

Now we can achieve end-to-end guarantees: based on fully automated static analyses of source programs, we can efficiently obtain machine-checked proofs about the behavior of machine language. In this paper I will outline the architecture of one such end-to-end system for concurrent programs. The system itself is not *at present* connected end-to-end; here I outline the possible, not report an achieved result. However, each of the components has been built, by different researchers at different institutions in different countries, and the method of connecting them has become clear.[1]

***A top-to-bottom verified architecture.*** At the top we have a C program that uses malloc/free, pointers, threads and locks, all with some conventional discipline. At the bottom we have the machine language of an instruction set architecture (ISA). (In fact, we can go higher than C and lower than the ISA, as I'll discuss at the end.)

We apply a modern automatic static analysis algorithm to the C program. This can guarantee important safety properties with little or moderate effort from the programmer: error messages from static-analyzer can usually give appropriate feedback in the programming process. For example, "lock $l$ is always held whenever shared variable $x$ is accessed." More sophisticated analyses can track dynamic patterns of lock-to-data correspondence, and can work even in the presence of pointers and aliasing.

As a case study, we will choose a particular static analysis. A *shape analysis* is an analysis of how a program uses pointer data structures. For example one of the classic papers on this topic explains in its title, "Is it a tree, a DAG, or a cyclic graph?" [5] More recent shape analyses include those of Gotsman *et al.* [7] or that of Guo *et al.* [8]. Both of these algorithms appeared in PLDI'07 and represent the state of the art. Gotsman's algorithm proves that a concurrent program accesses data only when it holds the relevant lock; Guo's algorithm deduces the backpointer/crosspointer/downpointer invariants of complex linked data structures, for use in a parallelizing compiler. Even more recent is Yang *et al.* [12] which scales up a separation-logic-based shape analysis algorithm so that it can verify real programs up to 10,000 lines of code.[2] They write, "It identifies memory safety errors and memory leaks in several Windows and Linux drivers and, after these bugs are fixed, it automatically proves integrity of pointer manipulation for these drivers."

***"Proves integrity?"*** Many static analyzers are unsound but useful—they catch some erroneous programs, but permit some buggy programs to slip through—we can characterize them as "bug-finding tools." In contrast, we are interesed in provably sound analyses, as are Gotsman, Guo, and Yang, whose analyses all come with soundness proofs: if the analyzer finds invariants about a program, then those invariants must hold in the operational semantics of the source language. However, these soundness proofs are problematic for several reasons:

[1] This is not a survey! At each level I will cite one or two illustrative examples, but of course there are many more that I don't have space to cite here.

[2] Years ago, such algorithms were impractical to imagine because they would require hundreds of megabytes of memory. In contrast, Yang's algorithm is practical today because it runs in only a gigabyte!

**Too hard.** The proofs (typically) address many things at once: an algorithm, an abstract interpretation lattice, an operational semantics. It takes great effort to build such a proof, and heroic effort to read it.

**Semiformal.** The static-analysis tool is a real program (not just an algorithm), operating on a real programming language (not just an abstraction). The proof is (typically) about an algorithm applied to an abstraction.

**Hand-checked.** Real software needs machine-checked proofs. There are too many details in the specification for any nonmechanized proof checker to keep track of. Real software is not frozen, it is maintained and evolved: and "by hand" proofs don't track changes in a reliable way.

**Source v. machine.** Most abstract interpretations—especially those that give usable feedback to the programmer—are at the source-language level, but programs execute in machine language. Can we trust the compiler?

***Separation Logic.*** Shape analyses have much to do with alias analysis. Therefore it is very natural to use Separation Logic to express what the analyses do, since SL is also about antialiasing. Indeed, many recent algorithms, including Gotsman, Guo, and Yang, are based on principles of Separation Logic.

Separation Logic describes program state with assertions using operators such as,

$P * Q$ Assertion $P$ holds on one part of memory, and $Q$ holds on a disjoint part of memory;

$x \mapsto y$ This assertion holds on a part of memory with just address $x$ holding contents $y$.

$\mathbf{emp}$ This assertion holds only on an empty part of memory, i.e. with no addresses at all.

$\mathbf{True}$ holds on any memory or portion thereof.

For example, the assertion $(x \mapsto 0) * (y \mapsto 0) * \mathbf{True}$ means that $m(x) = 0$, $m(y) = 0$, and $x \neq y$. A tree data structure can be described by,

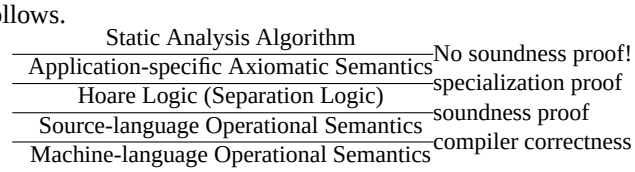
$$\begin{aligned} \mathrm{tree}(x) \;=\; & (x = 0) \;\vee \\ & \exists k, y, z.\ (x \mapsto k) * \\ & \qquad (x + 1 \mapsto y) * (x + 2 \mapsto z) * \\ & \qquad \mathrm{tree}(y) * \mathrm{tree}(z) \end{aligned}$$

The separation $*$ ensures that it is not a DAG or cyclic, because the addresses within $\mathrm{tree}(y)$ must be disjoint from those within $\mathrm{tree}(z)$ and from $x$.

***Architecture of a top-to-bottom proof.*** What Gotsman's, Guo's, and Yang's algorithms have in common is that they use sophisticated abstract-interpretation domains to find a set of program invariants $I$ that are expressible as separation logic formulae.[3] Once these invariants are found, it is a fairly simple matter to apply the axioms of Separation Logic to verify that the program satisfies the invariants. Therefore, to prove the soundness of one of these analysis algorithms, *it is not necessary to reason about the sophisticated abstract interpretation algorithm that finds the invariants.* Instead, we have to verify a much simpler algorithm, the one that checks the invariants by applying the axioms of separation logic.

Then, when an algorithm such as Yang's "proves integrity" of a source-language program, we can guarantee using machine-checked proofs that the compiled (machine-language) program will have this integrity. We address the problem of "Too hard" soundness proof in three ways. We peel away the abstract interpretation lattice, because it is used only to find the invariants, and we can use an axiomatic semantics (Hoare Logic, Separation Logic) to check the application of the invariants to the program. We peel away the operational semantics, because we will use the Separation Logic (an axiomatic semantics) as an abstraction layer.

The software architecture of our top-to-bottom proof is as follows.




*No soundness proof is necessary for the abstract interpretation,* because the invariants found by the shape analysis are checked by a straightforward application of an analysis-specific axiomatic semantics. It is that axiomatic semantics that must be foundationally proved sound. This may be identically a standard Separation Logic, or it may be a set of specialized inference rules proved as derived lemmas from a general Separation Logic (this is the "specialization proof.") Such specialization proofs are not very difficult, even in a machine-checked setting.

***Compiler correctness.*** In a remarkable tour de force, Leroy has demonstrated a *proved correct optimizing compiler* from C to machine language [10]. As part of this demonstration, Leroy specified an operational semantics for C minor, a high-level intermediate language for C; he specified an operational semantics for the PowerPC machine language; and he built a machine-checked proof in Coq that the compiler preserves behavior from one operational semantics to another. Part of Leroy's achievement is that he makes it look like it's not so difficult after all: now that he's found the right architecture for building verified compilers, everyone will know how to do it.

This result establishes the bottom two layers of the software architecture—as well as the *compiler-correctness proof* that connects them—*for sequential programs*.

***Axiomatic to operational.*** Appel and Blazy [1] have specified a Separation Logic for C minor, and proved it sound

[3] For interprocedural analyses, these invariants include pre/postconditions of functions.

with respect to the operational semantics. This establishes the *soundness proof* of the software architecture.

Because we are interested in the extension to concurrent C programs, we used a small-step operational semantics for C minor, in contrast to Leroy's mixed big-step/small-step semantics. However, Leroy is evolving his compiler-correctness proof toward a small-step semantics, one layer at a time, bottom-up. At present we can connect the Appel-Blazy soundness proof to the Leroy correctness proof via a small-step/big-step simulation result, proved in Coq.

Therefore, all the different layers are established in principle to connect a *sequential* shape analysis such as Guo's or Yang's to the operations of the machine language program. A layered construction will make it possible to achieve this connection with end-to-end machine-checked proofs. The axiomatic semantics is an important abstraction layer, as is the operational semantics is also an important layer. In fact, within the verified compiler there are another half-a-dozen layers of intermediate representation. Each phase of the compiler is proved correct w.r.t. the operational semantics of the intermediate representation above and below it.

***Concurrency.*** But we want a *concurrent* system. For example, we want to use a shape analysis for concurrent programs, such as Gotsman's [7].

O'Hearn [11] presented Concurrent Separation Logic as an extension of Separation Logic for reasoning about shared-memory concurrent programs with Dijkstra semaphores. It provides substantial flexibility in the dynamic assignment of resources to locks, and at the same time permits modular reasoning. However, it does not allow first-class locks and threads, that is, creating a lock at any address or forking a thread to complete asynchronously.

We presented an extension of O'Hearn's CSL to first-class locks and threads [9], and independently Gotsman et al. [6] presented a very similar CSL with first-class locks and threads; this agreement is evidence that we're both right. Gotsman et al. present a by-hand soundness proof with respect to an abstract model. However, we wanted to connect the layers of the software architecture; that is, we want a connection to the compilable operational semantics of C minor.

Here there was a nontrivial problem to solve. The author of an optimizing compiler, such as the CompCert compiler for C minor, wants to think of it as a compiler for a sequential programming language, with a library of concurrency features (threads and locks). Sequential reasoning is convenient not only when building the compiler, but when proving it correct. Boehm [3] explains very clearly why this naive model won't work, and explains that a good specification is needed for sequential languages with threads; but he doesn't provide the specification. We have done so, in a way that is as friendly as possible to the compiler writer (and prover). Our operational semantics for Concurrent C minor [9] preserves as much sequentiality as possible, by talking about *permissions within a single thread* instead of about concurrency.

Our operational semantics is based on ideas from Concurrent Separation Logic: the resource invariant of each lock is an explicit part of the operational model. There is a (classical logic) test for the satisfaction of the resource invariant whenever releasing a lock; this means that the operational semantics is nonconstructive. However, this model is very well suited for programs that are accompanied by proofs in CSL—either correctness proofs done interactively in a proof assistant, or safety proofs done automatically by a shape analysis such as Gotsman's.

Thus, using a combination of Gotsman's shape-analysis algorithm with our soundness proof for CSL, we can achieve the software proof architecture (sketched above) for concurrent programs as well as for sequential ones. Programs proved safe or correct in the source language will have the right behavior in machine language.

***Low-level libraries.*** Not explicitly shown in the architecture diagram, but certainly necessary, will be correctness proofs for low-level assembly-language libraries for locks, threads, memory allocation, and interrupt handling. Recent work on machine-checked formal verification using on a Hoare-logic-like framework specialized to this low level [4] is very encouraging, as a way to implement and verify this component.

***Onward and upward (and downward).*** One can extend the proof layers upward and downward. At the top, one can put a more expressive concurrent programming language than C-threads—for example, software transactional memory or a monadic dependently typed functional language. At the bottom, one can prove that the machine-language program executes equivalently in weakly consistent memory models (which will be true, because the CSL model does not permit race conditions); and one can prove (machine-checked) the correspondence of the ISA to the gates [2].

## References


[1] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *20th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2007)*, 2007.

[2] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together: Formal verification of the vamp. *STTT Journal*, 8(4-5):411–430, August 2006.

[3] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, New York, 2005.

[4] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with preemptive threads. page (to appear), 2008.

[5] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[6] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.

[7] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI '07: 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[8] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI '07: 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, 2007.

[9] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP'08: 17th European Symposium on Programming*, page (to appear). Springer, April 2008.

[10] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54, 2006.

[11] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[12] Hongseok Yand, Oukseh Lee, Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. On scalable shape analysis. Technical Report RR-07-10, Queen Mary, Univ. of London, November 2007.