Verified Software Toolchain
(Invited Talk)

Andrew W. Appel

Princeton University

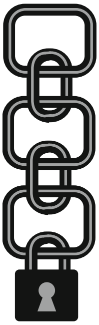
Abstract. The software toolchain includes static analyzers to check assertions about programs; optimizing compilers to translate programs to machine language; operating systems and libraries to supply context for programs. Our *Verified Software Toolchain* verifies with machine-checked proofs that the assertions claimed at the top of the toolchain really hold in the machine-language program, running in the operating-system context, on a weakly-consistent-shared-memory machine.

Our verification approach is modular, in that proofs about operating systems or concurrency libraries are oblivious of the programming language or machine language, proofs about compilers are oblivious of the program logic used to verify static analyzers, and so on. The approach is scalable, in that each component is verified in the semantic idiom most natural for that component.

Finally, the verification is *foundational:* the trusted base for proofs of observable properties of the machine-language program includes only the operational semantics of the machine language, not the source language, the compiler, the program logic, or any other part of the toolchain—even when these proofs are carried out by source-level static analyzers.

In this paper I explain some semantic techniques for building a verified toolchain.

Consider a software toolchain comprising,



A *Static analyzer* or *program verifier* that uses program invariants to check assertions about the behavior of the source program.

A *compiler* that translates the source-language program to a machine-language (or other object-language) program.

A *runtime system* (or operating system, or concurrency library) that serves as the runtime context for external function calls of the machine-language program.

We want to construct a machine-checked proof, from the foundations of logic, that *Any claims by the static analyzer about observations of the source-language program will also characterize the observations of the compiled program.*

We may want to attach several different static analyzers to the same compiler, or choose among several compilers beneath the same analyzer, or substitute one operating system for another. The construction and verification of just one component, such as a compiler or a static analyzer, may be as large as one project team or research group can reasonably accomplish. For both of these reasons, the interfaces between

G. Barthe (Ed.): ESOP 2011, LNCS 6602, pp. 1–17, 2011.

components—their specifications—deserve as much attention and effort as the components themselves.

We specify the observable behavior of a concurrent program (or of a single thread of that program) as its *input-output* behavior, so that the statement *Program p matches specification S* can be expressed independently of the semantics of the programming language in which p is written, or of the machine language. Of course, those semantics will show up in the proof of the claim! Sections 10 and 11 explain this in more detail.

But observable "input-output" behaviors of individual shared-memory threads are not just the input and output of atomic tokens: a lock-release makes visible (all at once) a whole batch of newly observable memory, a lock-aquire absorbs a similar batch, and a operating-system call (read into memory buffer, write from memory buffer, alloc memory buffer) is also an operation on a set of memory locations. Section 1 explains.

The main technical idea of this approach is to define a thread-local semantics for a well-synchronized concurrent program and use this semantics to trick the correctness proof of the *sequential* compiler to prove its correctness on a concurrent program. That is, take a sequential program; characterize its observable behavior in a way that ignores intensional properties such as individual loads and stores, and focus instead on (externally visible) system calls or lock-aquire/releases that transfer a whole batch of memory locations; split into thread-local semantics of individual threads; carry them through the modified sequential proof of compiler correctness; gather the resulting statements about observable interactions of individual threads into a statement about the observable behavior of the whole binary.

Because our very expressive program logic is most naturally proved sound with respect to an operational semantics with fancy features (permissions, predicates-in-the-heap) that don't exist in a standard operational semantics (or real computer), we need to erase them at some appropriate point. But when to erase? We do some erasure (what we call the transition from *decorated* op. sem. to *angelic* op. sem.) *before* the compiler-correctness proof; other erasure (from *angelic* to *erased*) should be done much later.

We organize our proof, in Coq, as follows. (Items marked • are either completed or nearly so; items marked ○ are in the early stages; my principal coauthors on this research (in rough chronological order) are Sandrine Blazy, Aquinas Hobor, Robert Dockins, Lennart Beringer, and Gordon Stewart. Items marked – are plausible but not even begun.)

- • We specify an expressive *program logic* for source-language programs;
- ○ we instrument the static analyzer to emit witnesses in the form of invariants;
- ○ we reimplement just the *core* of the static analyzer (invariant checker, not invariant inference engine) and prove it correct w.r.t the program logic;
- • we specify a *decorated operational semantics* for source-language programs;
- • we prove the soundness of the program logic w.r.t. the decorated semantics;
- • we specify an *angelic operational semantics*;
- • we prove a correspondence between executions of the decorated and the angelic semantics;
- ⋆ we prove the correctness of the optimizing compiler w.r.t. the angelic operational semantics of the source and machine languages;

  - we prove the equivalence of executions in the angelic operational semantics in a *weakly consistent* memory model and in a *sequentially consistent* memory model;
- we specify an *erased* (i.e., quite conventional) *operational semantics* of the machine language;
- we prove a correspondence between executions of the angelic and erased semantics.

The composition of all these steps gives the desired theorem. Q.E.D.

The $\star$ verified optimizing compiler is not by our research group, it is the CompCert compiler by Leroy *et al.* [24] with whom we have been collaborating since 2006 on adjusting the CompCert interfaces and specifications to make this connection possible. Since Leroy *et al.*'s correctness proof of CompCert in Coq constitutes one of the components of our modular verification, it seemed reasonable to do the rest of our verification in Coq. In addition, some of the logical techniques we use require a logic with dependent types, and cannot be expressed (for example) in HOL (higher-order logic, a.k.a. Church's simple theory of types). We wanted to use a logic whose kernel theory (in this case, CiC) is trustworthy, machine-checkable, and well understood. Finally, we wanted a system with well-maintained software tools, mature libraries of tactics and theories, and a large user community. For all these reasons we use Coq for our Verified Software Toolchain (VST), and we have been happy with this choice.

1 Observables

Specifications of program behavior must be robust with respect to translation from source language to machine language. Although the states of source- and target-language programs may be expressed differently, we characterize the *observable behaviors* (or *observations*) of source- and target-language programs in the same language [15].

Our source language is a dialect of C with shared-memory concurrency using semaphores (locks). The C programs interact with the outside world by doing system calls—with communication via parameters as well as in memory shared with the operating system—and via lock synchronization (through shared memory). One thread of a shared-memory concurrent program interacts with other threads in much the same way that a client program interacts with an operating system. That is, the thread reads and writes memory to which the other threads (or operating system) have access; the thread synchronizes with the other threads (or OS) via special instructions (trap, compare-and-swap) or external function calls (read-from-file, semaphore-unlock). We want the notion of *observable behavior* to be sufficiently robust to characterize either kind of interaction.

For each system call, and at each lock acquire/release, some portion of the memory is observable. But on the other hand, some parts of memory should not be observable.[1] In private regions of memory, compilers should have the freedom to optimize loads and stores of sequential programs: to hoist loads/stores past each other, and past control operations, to eliminate redundant loads and stores, etc.—subject only to dataflow constraints.

[1] Of course, the operating system can observe any part of memory that it wants to. But the O.S. should not "care" about the thread's local data, while it does "care" about the contents of memory pointed to by the argument of a write system call.

Of course, with a naive or ill-synchronized approach to shared memory, it is possible for one thread to see another's loads and stores, and therefore the compiler might alter the output of a program by "optimizing" it. At least for the time being, we restrict our attention to well-synchronized programs—in which any thread's write (read) access to data is done while holding a semaphore granting exclusive (shared) permission to that data. In such a regime, we can achieve that *loads and stores are not observable events.* This is an important design principle.

"Volatile" variables with observable loads/stores can be treated in our framework, with each load or store transformed—by the front end—into a special synchronization operation. Leroy has recently instrumented the front end of CompCert to perform this transformation.

Extensional properties. This approach to program specification deliberately avoids intensional properties, i.e. those that characterize *how* the computation proceeds. This design decision frees the compiler to make very general program transformations. However, that means that we cannot specify properties such as execution time.

Permissions and synchronization. When a thread does a system call or a lock synchronization, there is implicitly a set of addresses that may be "observed." Our specification of observations makes this explicit as *permissions*. A thread that (at a given time) has write permission to a memory address can be sure that no other thread (or operating system) can observe it. Lock synchronizations and system calls can change a thread's permission in controlled and predictable ways. From the compiler's point of view, synchronizations and system calls are instances of *external function calls*, which can change memory permissions in almost arbitrary ways.

The permissions have no concrete runtime manifestation—since, of course, we are compiling to raw machine code on real machines. Instead, they are a "fictional" artifact of static program invariants: a static permission-based proof about a program says something about its non-stuck execution in a permission-carrying operational semantics. Then, the permissions present in our decorated and angelic operational semantics disappear in the erased operational semantics.

In fact, the difference between the decorated and angelic operational semantics is in how the permissions change at external function calls. The decorated operational semantics is annotated with sufficient program invariants to calculate the change in permissions; the angelic semantics is equipped with an "angel" that serves as an oracle for the change in permissions; see Section 7.

2 The Programming Language and Compiler

Our VST project was inspired in part by Leroy's CompCert verified optimizing C compiler [24]. CompCert is a remarkable achievement, on the whole and in many particulars. For example, the CompCert memory model by Leroy and Blazy [25] supports storing of bytes, integers, floats, and relocatable pointers in a byte-addressable memory; is sufficiently abstract to support relocation and the kinds of block-layout adjustments that occur during compilation; and is sufficiently general that the same memory model can be used for the source-level semantics, the target-machine semantics, and at every level in between. The tasteful design of the memory model is one reason for CompCert's success.

CompCert "merely" proves that, whenever it compiles a C program to assembly language, any safe execution in C corresponds to a safe execution in assembly with the same observable behavior. But where do safe C programs come from, and how can one characterize the observable behavior of those C programs? If those questions cannot be answered, then it might seem that CompCert is like a highly overengineered hammer without any nails.

Our Verified Software Toolchain uses *C minor* [24] as a source language. C minor is the high-level intermediate representation used in CompCert for the C programming language. CompCert compiles the C programming language (or rather, a very substantial subset called *C light* [9]) through 7 intermediate languages into assembly language for the PowerPC processor; the third language in this chain of 9 is C minor. For each language in the chain, Leroy has specified the syntax and operational semantics in the Coq theorem prover. Each of the 8 compiler phases between these intermediate languages is proved (in Coq) to preserve observational equivalence between its input (a program in one intermediate language) and its output (a program in the next language).

We chose to use C minor (instead of C light) as the target for VST, for two reasons: C minor is more friendly to Hoare-style reasoning as there are no side effects inside expressions;[2] and C minor can be used as a target language from source languages such as ML or Java.

3 Oracle Semantics

Remarkable as CompCert is, its original specification was too weak to express the interaction of a thread with its context—whether that context is the operating system or other concurrent threads. We want to remedy that weakness without making fundamental changes to the CompCert correctness proof.

We factor the operational semantics to separate *core programming-language* execution from execution steps in the *operating-system/runtime-system/concurrency context,* which we call the *oracle* for short. We do this because the same oracle will be applied to both the source- and machine-language programs; and because (conversely) different kinds of oracles will be applied to the same source- or machine-language program.

The CompCert C compiler [24] is proved correct with respect to a source-level operational semantics and a machine-level operational semantics. In early versions of CompCert, the source-level semantics was big-step and the machine-level was small-step. Each semantics generated a *trace*, a finite or infinite sequence of observable events, that is, system calls with atomic arguments and return values.

This specification posed obstacles for integration into our verified toolchain: the big-step semantics was inconvenient for reasoning about concurrent threads; the system calls did not permit shared memory between client and operating system (e.g. did not permit the conventional model of system calls); the lack of shared memory meant that

[2] More precisely, because C minor is not a subset of C, we had some flexibility to negotiate with Leroy a few modifications to the specification of C minor: e.g., unlike C minor 2006, C minor 2010 has no side-effects inside expressions.

a shared-memory concurrency model was difficult; the coarseness of memory-access permissions meant that concurrent separation logic was difficult; and so on.

Therefore we worked with Xavier Leroy to adjust the specification of CompCert's source-level operational semantics. Now it is a small-step semantics, with a lattice of permissions on each memory address. Instead of a trace model–in which the behavior of a program is characterized by a sequence of atomic values read or written, we characterize the behavior by the thread's history of (shared-memory) interactions with the oracle. Each state of the small-step semantics contains three components: oracle state Ω, memory m, and core state q. A *core step* is a small-step of computation that is *not* an external function call; an *oracle step* is taken whenever the computation is at an external function call. Each core step affects only m and q, leaving Ω unchanged; each oracle-step affects only Ω and m.

What is the oracle? If we are modeling the interaction of a sequential C program with its operating system, then the oracle is the operating system, and the external function calls are system calls. If we are modeling the interaction of a sequential C thread with all the other threads in a shared-memory concurrent computation, then the external function calls are lock-acquire and -release, and the oracle is *all the other threads*. An early version of this oracle model of concurrency is described by Hobor *et al.* [20,21].

The advantage of the oracle model is that, from the point of view of the C compiler and its correctness proof, issues of concurrency hardly intrude at all. Still, Hobor's oracle model was only partly modular: it successfully hid the internal details of the oracle from the compiler-correctness proof, but it did not hide all the details of the programming language from the oracle. This was a problem, because it was difficult to characterize the result of compiling a concurrent C program down to a concurrent machine-language program. Dockins [15] has a substantially more modular oracle model, which is the basis for the verified toolchain described here.

4 The Program Logic

Proofs of static analyses and program verifiers are (often) most convenient with respect to an *axiomatic semantics* (such as a Hoare Logic) of the programming language; we call this the *program logic*. Proofs of compilers and optimizations are (often) most convenient using an *operational semantics*.[3] Therefore for the source language we will need both an axiomatic and an operational semantics, as well as a machine-checked soundness proof that relates the two.

We intend to use the program logic as an intermediate step between a static analysis algorithm and an operational semantics. We would like to do this in a modular way—that is, prove several different static analyses correct w.r.t. the same program logic, then prove the program logic sound w.r.t. the operational semantics. Therefore we want the program logic to be as general and expressive as possible.

We start with a Hoare Logic for C minor: the judgement $\Gamma \vdash \{P\}c\{Q\}$ means that command c has precondition P and postcondition Q. Somewhat more precisely, if c starts in a state satisfying P, then either it safely infinite-loops or it reaches a state

[3] At the very least, the particular proof that we want to connect our system to—CompCert—is w.r.t. an operational semantics.

satisfying Q. However, C minor has two control-flow commands that can avoid a fall-through exit: $\mathsf{exit}\ n$ breaks out of n nested blocks (within a function), and $\mathsf{return}\ e$ evaluates e and returns its value from the current function-call. Thus, Q is really a triple of three postconditions; an ordinary assertion for fall-through, an assertion parameterized by n (equivalently, a list of assertions) for exit and an assertion parameterized by return-value for return.

To take a simple example, the judgement

$$\Gamma \vdash \{\exists x.\ (v \Downarrow x) * (x \overset{4,\pi}{\mapsto} 6) * (x{+}4 \overset{4,\top}{\mapsto} x)\}\ [v{+}4] :=_4 0\ \{\exists x.\ (v \Downarrow x) * (x \overset{4,\pi}{\mapsto} 6) * (x{+}4 \overset{4,\top}{\mapsto} 0)\}$$

can be read as follows: Before execution of the *store* statement, local variable v points to some address x; the current thread has partial permission π to read (but not necessarily write) a 4-byte integer (or pointer) at x; the current 4-byte contents of memory at x is 6; the thread has full permission $\top$ to read or write a 4-byte integer (or pointer) at $x + 4$; the current contentents at $x + 4$ is x. Furthermore, by the rules of $*$ in separation logic, x is not aliased to $x + 4$, though in this case we didn't need separation logic to tell us this obvious fact!

After the assignment, the situation is much the same except that the contents of $x + 4$ is now a null pointer. The exit and return postconditions (not shown) are both **false** since the store command neither exits nor returns.

The global context Γ maps addresses to their function specifications. The assertion $f : \{P\}\{Q\}$ means f has precondition P and postcondition Q.

Since C (and C minor) permits passing functions as pointers, in the program logic one can write $\exists v.\ (f \Downarrow v) * (v : \{P\}\{Q\})$, meaning that the name f evaluates to some pointer-value v, and v has precondition P, etc. Since the function can take a sequence of parameters and return an optional result, in fact P and Q are functions from value-list to assertion.

The operators $\exists$, $*$, $\mapsto$, and : are all constructed as definitions in Coq from the underlying logic. We use a "shallow embedding," but because of the almost-self-referential nature of some assertions ("$f(x)$ is a function whose precondition claims that x is a function whose precondition..."), the construction can be rather intricate; see Section 6.

In a Hoare logic one often needs to relate certain dynamic values appearing in a precondition to the same values in a postcondition. For example, consider the function $\mathsf{int\ f(int\ x)\{return\ x + 1;\}}$. One wants to specify that this function returns a value strictly greater than its argument. We write a specification such as,

$$\exists v.\ f \Downarrow v\ \wedge\ v : (\forall x : \mathsf{int}.\{\lambda a.a = x\}\{\lambda r.r > x\})$$

or, using a HOAS style for the : operator, $\exists v.\ f \Downarrow v\ \wedge\ v : [\mathsf{int}]\{P\}\{Q\}$
where $P = \lambda x \lambda a.(a = x)$ and $Q = \lambda x \lambda r.(r > x)$. This notation permits the shared information x to be of any Coq type τ, where in this example $\tau = \mathsf{int}$.

The use of HOAS (higher-order abstract syntax) hints that we shallowly embed the assertions in the surrounding logical framework, in this case Coq, to take advantage of all the binding structure there.

To reason about polymorphic functions or data abstraction, our Hoare logic permits universal and existential quantification: $\forall x : \tau.P \qquad \exists x : \tau.P$

where τ can be any Coq type, and P can contain free occurrences of x. Of course this is just Coq "notation" for the HOAS version: $\forall[\tau]P'$ $\exists[\tau]P'$
where P' is a function from τ to assertion.

It is particularly important that τ can be any Coq type *including* quantified assertions. That is, the quantification is *impredicative.* Impredicativity is necessary for reasoning about data abstraction, where the abstract types in the interface of one module can be themselves implemented by abstract types from the interface of another module.

To specify inductive datatypes, our program logic has an operator μ for recursively defined assertions: $\mu x.P(x)$ satisfies the fixpoint equation $\mu x.P(x) = P(\mu x.P(x))$, (or in HOAS $\mu P = P(\mu P)$), provided that P is a contractive function [6].

Ordinary Hoare logics have difficulty with pointers and arrays, especially with updates of fields and array slots. Therefore we use a *separation logic*, which is a Hoare logic based on the theory of bunched implications. In an ordinary Hoare logic, the assertion $P \wedge Q$ means that P holds on the current state and so does Q. In separation logic, $P * Q$ means that the current state can be expressed as the disjoint union of two parts, and P holds on the first part while Q holds on the second part. The assert emp holds on exactly the empty state, and **true** holds on any state. Thus the assertion $(P * Q) \wedge (R * \text{true})$ holds on any state s that can be broken into two pieces $s_1 \oplus s_2$, where P holds on one and Q on the other; *and* such that R holds on some substate of the state.

Our separation logic has a notion of partial ownership, or "permissions." The assertion $x \overset{\pi}{\mapsto} y$ means that the current state has exactly one address x in it; the contents of this address is the value y; and the current state has nonempty permission π to read (or perhaps also to write) the value. Permissions form a lattice; some permissions give read access, stronger permissions give write access, and the "top" permission gives the capability to deallocate the address. In contrast to previous permission models [11,29] ours is finite, constructive, general, and modeled in Coq [16].

Partial ownerships and overlapping visibility mean that "disjoint union" is an oversimplification to describe the $\oplus$ operator; we use a *separation algebra,* following Calcagno *et al.;* but we have a more general treatment of units and identities [16].

To reason about concurrent programs, we use a *concurrent separation logic* (CSL). O'Hearn [28] demonstrated that separation logic naturally extends to concurrency, as it limits each thread to the objects for which it has (virtual) permission to access. Portions of the heap can have their virtual ownership transferred from one thread to another by synchronization on semaphores. We have generalized O'Hearn's original CSL to account for dynamic creation of locks and threads [21]; Gotsman *et al.* made a similar generalization [18]. The assertion $\ell \overset{\pi}{\leadsto} R$ means that address ℓ is a lock (i.e., semaphore) with visibility (permission) π and resource invariant R. Permission $\pi < \top$ means that, most likely, some other thread(s) can see this lock as well; if only one thread could see ℓ then it would be difficult to use for synchronization! Any nonempty π gives permission to (attempt to) acquire the lock. Resource invariant R means that whenever a thread releases ℓ, it gives up a portion of memory satisfying the assertion R; whenever a thread acquires ℓ, it acquires a new portion of memory satisfying R. While ℓ is held, the thread can violate R until it is ready to release the lock.

The basic CSL triples for acquire/release are

$$\{\ell \overset{\pi}{\leadsto} R\}\ \mathsf{acquire}\,\ell\ \{R * \ell \overset{\pi}{\leadsto} R\} \qquad \{R * \ell \overset{\pi}{\leadsto} R\}\ \mathsf{release}\,\ell\ \{\ell \overset{\pi}{\leadsto} R\}.$$

In summary, programs have pointers, we want to reason about aliasing, hence Separation Logic rather than just Hoare Logic. Programs have higher-order functions, polymorphism, and data abstraction; hence we want impredicative quantification in the logic. Programs have concurrency, hence we want resource invariants. Finally, different front-end static analyses will make different demands on the program logic, so we want the program logic to be as expressive and general as we can possibly make it. We end up with *impredicative higher-order concurrent separation logic.*

5 Instrumenting Static Analyses

Hoare-style judgements can be proved in an interactive theorem prover by applying the inference rules of the program logic. But there are many kinds of program properties that, with much less user effort, a static analyzer can "prove" automatically. We are interested in removing the quotation marks from the previous sentence.

One way to connect a static analyzer to a program logic is to have the analyzer produce derivation trees in the logic. But this may require a large change to an existing analyzer, or significantly affect the software design of an analyzer; and the derivation trees will be huge. Another way would be to prove the analyzer correct, but analyzers are typically large programs implemented in languages with difficult proof theories; sometimes they have major components such as SAT solvers or ILP solvers.

We propose to instrument the analyzer in a different way. A typical analyzer infers program *invariants* from *assertions* provided by the user. In a sense, the invariants are the induction hypotheses needed to verify the assertions. We propose that the analyzer should output the program liberally sprinkled with invariants. A much simpler program than the analyzer, the *core* of the analyzer, can check that the invariants before and after each program statement (or perhaps, program block) match up.

We will implement the core of each analyzer in the Gallina functional programing language embedded in the Coq theorem prover. We will use Coq to prove the soundness of the core w.r.t. our program logic. Then we use Coq's program-extraction to obtain a Caml program that implements the core. The toolchain then starts with the full (untrusted, unverified) static analyzer, whose results are rechecked with the (verified) core analyzer.

My undergraduate student William Mansky conducted one successful experiment in this vein [26]: he reimplemented in Gallina the concurrent-separation-logic-based shape analysis by Gotsman *et al.* [19], and proved it correct[4] with respect to our program logic. He found that this shape analysis was simple enough that he could implement the entire analysis including invariant inference in Gallina, without needing an unverified front end.

[4] Well, almost proved. The problem is that Gotsman *et al.* assume a Concurrent Separation Logic with imprecise resource invariants, whereas our CSL (as is more standard [28]) requires precise resource invariants. The joy of discovering mismatches such as this is one of the rewards in doing end-to-end, top-to-bottom, machine-checked proofs such as the VST.

6 Semantic Modeling of Assertions

We want to interpret an assertion P as a predicate on the state of the computation: more or less, $s \models P$ is interpreted as $P(s)$. Some of our assertion forms characterize information about permissions, types, and predicates that are nowhere to be found in a "Curry-style" bare-bones operational semantics of a conventional target machine. Therefore we use more than one kind of operational semantics: a *decorated* semantics with extra information, an *angelic* semantics with partial information, and an *erased* semantics with even less information.

We have erasure theorems—that any nonstuck execution in the decorated semantics corresponds to a nonstuck execution in angelic semantics with similar observables, and ditto for the angelic to the erased semantics. Assertions require the decorated semantics.

A judgement $s \models 10 \overset{\pi_1}{\mapsto} 8 * 11 \overset{\pi_2}{\mapsto} 0$ means that s (in the decorated and angelic semantics) must contain not only 8 at address 10 and 0 at address 11, but must also keep track of the fact that there is exactly π_1 permission at 10, and π_2 at 11.

The assertions $\ell \overset{\pi}{\leadsto} R$ (meaning that ℓ is a lock with resource invariant R and visibility π) and $v : [\tau]\{P\}\{Q\}$ (v is a function with precondition P and postcondition Q) are interesting in that one assertion is characterizing the binding of an address (ℓ or v) to another assertion (R, P, Q).

Somehow the decorated state s must contain predicates; which in turn predicate over states. Achieving this—in connection with impredicative quantification and recursive assertions—is quite difficult; the semantic methods of the late 20th century were not up to the task. If we interpret assertions naively in set theory, we violate Cantor's paradox.

We solve this problem using the methods of Ahmed [4] as reformulated into the "Very Modal Model" [6] and "Indirection Theory" [22]. The recent formulation of Birkedal *et al.* [8] could also be used. The basic trick is that each state s indicates a degree of approximation (a natural number k_s). When $s \models \ell \overset{\pi}{\leadsto} R$, then the assertion R can be applied to states s' only strictly more approximately than k_s. When the computation takes a step $s \longrightarrow s'$, it must "age" the state by making sure that s' is strictly more approximate than s, that is, $k_{s'} < k_s$.

In a decorated state an address can map to any of:

$\mathsf{val}_\pi\ v$	Memory data byte v with permission π
$\mathsf{lock}_\pi\ R$	Lock with resource-invariant R
$\mathsf{fun}_\pi\ [\tau]PQ$	Function entry point with precondition P, post Q

In the angelic semantics the predicates are removed—leaving only $\mathsf{val}_\pi\ v$, lock_π, fun_π—and in the erased semantics the permissions are removed.

7 Partial Erasure before Bisimulation

Our toolchain has, near the front end, a higher-order program logic; and near the back end, an optimizing compiler composed of successive phases. Each phase translates one language to another (or to itself with rewrites), and each phase comes equipped with a bisimulation proof that the observable behavior is unchanged.

The fact that program-states s contain "hair" such as approximation indices and predicates, poses some problems for the modularity of the top-to-bottom verification of the toolchain. To verify the soundness of the program logic w.r.t. the operational semantics, we need states s containing predicates. As explained in Section 6, Indirection Theory requires that predicates embedded in states must be at specific levels of approximation, and these need to be "aged" (made more approximate) at each step of the computation. On the other hand, to verify the correctness of the optimizing compiler the predicates are unneeded, and the aging hampers the bisimulation proofs. For example, a compiler phase might change the number of steps executed (and thus the amount of aging).

Thus we erase the predicates in moving from the decorated to the angelic semantics. But there are a few places where the predicates have an operational effect, and we replace these with an *angelic oracle* that supplies the missing information. All of our ageable predicates are within the Ω (oracle) component of the state; and thus this erasure has no effect on the m and q components.

Let address $\ell \overset{\pi}{\leadsto} R$ be a lock (semaphore) in Concurrent Separation Logic, with resource invariant R. R must be a precise separation-logic predicate, meaning that R is satisfied by a unique subheap (if any subset) of a given memory. When a thread releases this lock, the (unique) subheap satisfying R, of the current heap, is moved from thread-ownership to lock-pool ownership.

The operational effect on fully-erased memory is only that the semaphore goes from state 0 to state 1. The operational effect on decorated or angelic memory is that the permissions π of the thread (and of the lock pool) change. The way the permission-change $\Delta\pi$ is calculated, in the decorated semantics, is that R is fetched from the state and (classically, not constructively) evaluated.[5] That is, in a decorated state, at each lock address, is stored the resource invariant R.

Unlike decorated states, angelic states contain no predicates. Instead of calculating $\Delta\pi$ from R, the *angel* is an oracle that contains a list of all $\Delta\pi$ effects that would have (nonconstructively) satisfied from deleted predicates. At each semaphore-release operation, the next $\Delta\pi$ is consumed from the angel.

Dockins [15] shows the proof that *For each safe execution in the decorated semantics, there exists an angel that gives a safe execution in the angelic semantics with the same observable behavior.*

8 Bisimulation

After the static analyzer or a program verifier has proved something about the observable behavior of a source program in the decorated semantics, we partially erase to the angelic semantics. The compiler-correctness proof, showing that source-language and machine-language programs have identical observables, is done in the angelic semantics of the several intermediate languages.

[5] Our own proofs (and those of CompCert) do not use classical axioms such as choice and excluded middle; we use only some axioms of extensionality. But classical axioms are consistent with CiC+extensionality, so the user of our program logic may choose to use them. Therefore we do not assume that the satisfaction of R must be constructive.

The original CompCert proofs were done by bisimulation, with respect to a simple notion of finite or infinite traces of atomic events, with no shared memory. Because the CompCert languages are deterministic, the bisimulations reduce to simpler simulation proofs. These proofs rely on a set of general lemmas about simulations and bisimulations in small-step semantics equipped with observable traces.

We are able reuse the existing CompCert proofs with very little modification, by ensuring that our oracle/observable interface is as compatible as possible with the original trace-based specification. From oracles and observables, Dockins [15] proves a set of simulation lemmas very similar to the original versions in CompCert. From these, Leroy can easily adapt the CompCert proofs.[6] The fact that the operational semantics is angelic is almost invisible in the CompCert bisimulations, because the angel is consulted only at certain of the external function calls, and never during ordinary computation steps corresponding to instructions emitted by the compiler.

If there had been predicates in the heap (i.e. the decorated semantics), the bisimulations would have been harder to prove (and require more changes to the CompCert proofs), because the aging of predicates required by Indirection Theory would age the state by different amounts depending how many instructions a compiler optimization deletes.

Even so, there have been a few changes to the CompCert specification to accommodate our verified toolchain. In addition to the new notion of observables, CompCert now has an address-by-address permission map. Leroy has already modified CompCert (and its proof) to accommodate these permissions (CompCert 1.7, released 03/2010).

The need for permissions in the (angelic) operational semantics is in response to a problem explained by Boehm: “Threads cannot be implemented as a library.” [10] He points out that hoisting loads and stores past aquire/release synchronizations can be unsafe optimizations, in shared-memory concurrent programs. He writes, “Here we point out the important issues, and argue that they lie almost exclusively with the compiler and the language specification itself, not with the thread library or its specification.” Indeed, the permissions in our decorated or angelic semantics form such a specification, and CompCert is proved correct with respect to this specification.

9 Weak Memory Models

The angelic semantics (and its management of permissions) must be preserved down to the back end of the compiler, so that each optimization stage can correctly hoist loads/stores past other instructions. Only then can full erasure of permissions be done, into the erased semantics.

At this time, the program is still race-free, and the permissions in the angelic semantics allow a proof of this fact. This suggests that executions in a weakly consistent memory model should have the same observable behavior as sequentially consistent executions. It should be possible to prove this, for a given memory model, and to add this proof to the bottom of the verified software toolchain. Recent results in the formal specification of weak memory models [31,12] should be helpful in this regard.

[6] As of mid-2010, CompCert is not yet fully ported to the new model of observations.

10 Foundational Verification

Foundational verification is a machine-checked formal verification where we pay particular attention to the size and comprehensibility of the *trusted base,* which includes the specification of the theorem to be proved, the axiomatization of the logic, and the implementation of the proof checker.

It is instructive to use Proof-Carrying Code as a testbed for gedanken experiments about what constitutes the trusted base of a formal method for software. Then in the next section we can apply this methodology to the Verified Software Toolchain, and compare both the size of the trusted base and the strength of what can be proved.

Consider the claim that *Program P in the ML (or Java) language does not load or store addresses outside its heap.* What is the trusted base for a proof of this claim of memory safety? Implicit in this claim is that P is compiled by an ML (or Java) compiler to machine language. A full statement of this theorem in mathematical logic would include: (1) the specification of the operational semantics and instruction encodings of the machine language, (2) the specification of memory safety as a property of executions in the machine language, and (3) the entire ML (or Java) compiler. A verification is a proof of this theorem for P, and naively it might seem that part of this proof is a proof of correctness of the ML (or Java) compiler.

ML and Java are type-safe languages: a source-language program that type-checks cannot load or store outside the heap when executing in the source-language operational semantics. Consequently, the machine-language program should also be memory-safe, if the compiler is correct. Proof-carrying code (PCC) [27] was introduced as a way to remove the compiler from the trusted base, without having to prove the compiler correct—machine-checked compiler-correctness proofs seemed impractical in 1997. Instead of relying on a correct compiler, PCC instruments the (untrusted, possibly incorrect) compiler to translate the source-language types of program variables (in the source-language type system) to machine-language types of program registers (in some machine-language type system). Then one implements a program to type-check machine-language programs. Let P' be the compilation of P to machine-language; now the theorem is that P' *is memory-safe*. The trusted base includes the machine-language type-checker, but not the compiler: that is, one trusts that, or proves that, if P' type-checks then it is memory safe.

Proof-carrying code originally had three important limitations:

1. There is no guarantee that P' has the same observable (input/output) behavior as P, because the compiler is free to produce *any* program as output so long as it type-checks in the machine-language type system;
2. For each kind of source-level safety property that one wishes to proof-carry, one must instrument the compiler to translate source-level annotations to checkable machine-level annotations;
3. The claim that *Any program that type-checks in the machine-language type system has the desired safety property when it executes* is still proved only informally; that is, there was a LaTeXproof about an abstraction of a subset of the type-checker.

Foundational proof-carrying code [5] addresses this third point. We construct a formal specification of the machine-language syntax and semantics in a machine-checkable logic, and a specification of memory-safety in that same logic. Then the theorem is, *Program P' is memory-safe*, and neither the compiler nor the machine-language type annotations need be trusted; the type annotations are part of the *proof* of the theorem, and the proof need not be trusted because it is checked by machine.

The Princeton foundational proof-carrying code (FPCC) project, 1999-2005, demonstrated this approach for core Standard ML. We demonstrated that the trusted base could be reduced to less than 3000 lines of source code: about 800 for a proof checker, written in C and capable of checking proofs for any object logic representable in LF; a few lines for the representation of higher-order logic (HOL) in LF; and about 1500 lines to represent instruction encodings and instruction semantics of the Sparc processor [32]. We instrumented Standard ML of New Jersey to produce type annotations at each basic block [13] and we built a semantic soundness proof for the machine-level type system [1]. Other research groups also demonstrated FPCC for other compilers [17,14].

The late-20th-century limitation that inspired PCC and FPCC—that it is impractical to prove the correctness of an optimizing compiler—no longer applies. Several machine-checked compiler-correctness proofs have been demonstrated [23,24,30]. Proof-carrying code is no longer the state of the art.

11 What Is the Trusted Base?

Our main theorem is, *Claims by the static analyzer about the source-language program will characterize the observations of the compiled program.*

How is this statement to be represented in a machine-checkable logic? We may choose to use complex and sophisticated mathematical techniques to prove such a statement. The compiler may use sophisticated optimization algorithms, with correspondingly complex correctness proofs. We can tolerate such complexity in *proofs,* because proofs are machine-checked. But complexity in the statement of the theorem is undesirable, because some human—the eventual consumer of our proof—must be able to check whether we have proved the right thing.

A "claim" by a static analyzer relates a program p to the specification S of its observable behavior. That is, let A be the static analyzer, so that $A(p, S)$ means that the analyzer claims that every observation o of source-language program p is in the set S. Let C be the compiler, so that $C(p, p')$ means that source program p translates to machine-language program p'. Let M be the operational semantics of machine language, so that $M(p', o)$ means that program p' can run with observation o.

Then the main theorem is, $\forall p, p', S, o.\ A(p, S)\ \wedge\ C(p, p')\ \rightarrow\ M(p', o)\ \rightarrow\ o \in S$. That is, "if the analyzer claims that program p matches specification S, and the compiler compiles p to p', and p' executes on M with observation o, then o is permitted by S."

The *statement* of this theorem is independent of the semantics of the source language! If we expand all the definitions in this theorem, the "big" things are A, C, S, and M. The consumer of this theorem doesn't need to understand A or C, he just needs to make sure that A and C are installed, bit-for-bit, in his computer. The definition S is just the specification of the desired behavior of the program, and this will be as large or as concise as the user needs. The semantics M is only moderately large, depending on the machine architecture; in the FPCC project the Sparc architecture was specified in about 1500 lines of higher-order logic. [32]

What is *not* in the statement of this theorem is the operational or axiomatic semantics of the source language, or the semantic techniques used in defining the program logic! All of these are contained within the *proof* of the theorem, but they do not form part of the trusted based needed in interpreting what the theorem claims.

It seems paradoxical that a claim about the behavior of a program can be independent of the semantics of the programming language. But it works because of the end-to-end connection between the program logic and the compiler. The theorem can be read as, "When you apply the compiler to the syntax of this program, the result is a machine-language program with a certain behavior." That sentence can be uttered without reference to the particular source-language semantics.

What else do you need to trust? The analyzer A and compiler C are written in Gallina, the pure functional programming language embedded in the Coq theorem prover. Coq contains software s_1 to check proofs about Gallina programs, and software s_2 to translate Gallina to ML. The OCaml compiler s_3 translates ML to machine-language; the machine language runs with OCaml runtime-system s_4. All of these are in the trusted base, in the sense that bugs in the s_i can render the proved theorem useless.

FPCC had a much smaller trusted base, avoiding s_1–s_4.[7] But at least these components s_1–s_4 are fixed for all proofs and projects, and well tested by the community; and perhaps some of the techniques used in the FPCC project, such as a tiny independent proof checker, could be applied here as well to remove the s_i from the trusted base.

12 Conclusion

Highly expressive program logics require different semantic methods than compiler-correctness proofs. Proofs about a sequential thread require different kinds of reasoning than proofs about operating systems and concurrency. A top-to-bottom verified toolchain requires all these kinds of reasoning, and thus we choose the right formalism for each component, and prove the relationships between the formalisms. In this way we achieve a system that is sufficiently modular that its major components can be built and proved by entirely separate research groups.

In the process, more work goes into thinking about specifications and interfaces than into the individual components. This is not a bad thing, in the end.

A formal proof may be "wrong" for either of two reasons: There may be a mistake in one or more steps of the proof, so that the "proof" fails to prove the stated theorem; or the statement of the theorem may be not what was intended. Using a proof assistant to mechanically check the proof prevents the first kind of problem, but not the second. This is why I find it so important to do a big top-to-bottom system and connect all the components together in the same metalogic. For example, our paper in LICS'02 [2] was a correct proof of a nice-looking theorem (semantic model of a type system with mutable references, polymorphism, and recursive types) but only in the application to the big Foundational Proof-Carrying Code project did we discover that it was the wrong theorem: the type system had predicative quantification, but we needed impredicative. We had to go back to the drawing board and figure out how to do impredicativity as well

[7] For the explanation of why no compilers are in FPCC's trusted base, see [7, §8.2]. Unfortunately that argument does not apply to the VST.

[3,4]. Footnote 4 describes one such incident in the VST project, and there have been many more along the way. Big "systems" projects have an important place in research on the formal semantics of software.

Acknowledgments. Gilles Barthe, Lennart Beringer, Alexey Gotsman, Julia Lawall, Aleks Nanevski, Gordon Stewart, and Shaz Qadeer all gave me very helpful suggestions on how to organize and present this material. This research was supported in part by the Air Force Office of Scientific Research (grant FA9550-09-1-0138) and the National Science Foundation (grant CNS-0910448).

References

1. Ahmed, A., Appel, A.W., Richards, C.D., Swadi, K.N., Tan, G., Wang, D.C.: Semantic foundations for typed assembly languages. ACM Trans. Program. Lang. Syst. 32(3), 1–67 (2010)
2. Ahmed, A., Appel, A.W., Virga, R.: A stratified semantics of general references embeddable in higher-order logic. In: 17th Annual IEEE Symp. on Logic in Computer Science, pp. 75–86 (June 2002)
3. Ahmed, A., Appel, A.W., Virga, R.: An indexed model of impredicative polymorphism and mutable references (January 2003), `http://www.cs.princeton.edu/~appel/papers/impred.pdf`
4. Ahmed, A.J.: Semantics of Types for Mutable State. PhD thesis, Princeton University, Princeton, NJ, Tech Report TR-713-04 (November 2004)
5. Appel, A.W.: Foundational proof-carrying code. In: Symp. on Logic in Computer Science (LICS 2001), pp. 247–258. IEEE, Los Alamitos (2001)
6. Appel, A.W., Melliès, P.-A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007), pp. 109–122 (January 2007)
7. Appel, A.W., Michael, N.G., Stump, A., Virga, R.: A trustworthy proof checker. J. Automated Reasoning 31, 231–260 (2003)
8. Birkedal, L., Reus, B., Schwinghammer, J., Stovring, K., Thamsborg, J., Yang, H.: Step-indexed Kripke models over recursive worlds (2010) (submitted for publication)
9. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Symp. on Formal Methods, pp. 460–475 (2006)
10. Boehm, H.-J.: Threads cannot be implemented as a library. In: PLDI 2005: 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, pp. 261–268 (2005)
11. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL 2005, pp. 259–270 (2005)
12. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: POPL 2009, pp. 392–403 (2009)
13. Chen, J., Wu, D., Appel, A.W., Fang, H.: A provably sound TAL for back-end optimization. In: PLDI 2003: Proc. 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 208–219 (June 2003)
14. Crary, K., Sarkar, S.: Foundational certified code in the twelf metalogical framework. ACM Trans. Comput. Logic 9(3), 1–26 (2008)
15. Dockins, R., Appel, A.W.: Observational oracular semantics for compiler correctness and language metatheory (2011) (in preparation)

16. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 161–177. Springer, Heidelberg (2009)
17. Feng, X., Ni, Z., Shao, Z., Guo, Y.: An open framework for foundational proof-carrying code. In: Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI 2007), January 2007, pp. 67–78. ACM Press, New York (2007)
18. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
19. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI 2007: 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (2007)
20. Hobor, A.: Oracle Semantics. PhD thesis, Princeton University (2008)
21. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
22. Hobor, A., Dockings, R., Appel, A.W.: A theory of indirection via approximation. In: POPL 2010: Proc. 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 171–184 (January 2010)
23. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. on Programming Languages and Systems 28, 619–695 (2006)
24. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54 (2006)
25. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning 41(1), 1–31 (2008)
26. Mansky, W.: Automating separation logic for Concurrent C minor. Undergraduate thesis (May 2008)
27. Necula, G.: Proof-carrying code. In: 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 106–119. ACM Press, New York (1997)
28. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1), 271–307 (2007)
29. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
30. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
31. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
32. Wu, D., Appel, A.W., Stump, A.: Foundational proof checkers with small witnesses. In: 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (August 2003)