Research Statement

Adam Chlipala

The basic question that I want to explore is, what is the ideal architecture for software systems, given all of the new capabilities that mechanized theorem-proving has brought us? Today, much programming goes on in “general-purpose languages” which make most programming tasks too difficult and also make it too difficult to build confidence that programs will work correctly. I would like to separate these desiderata and make them the basis of a slogan: *verification for systems, synthesis for applications*.

Computer proof assistants are rapidly gaining popularity among researchers in programming languages and other software-centric specialties. These tools make it possible to construct completely rigorous mathematical proofs of a wide variety of theorems, from the four-color theorem to the full functional correctness of particular programs. Such a proof can be checked by a small proof-checking program, which understands only a small number of problem-independent axioms and can thus be made very trustworthy.

Formal verification of this kind has a reputation of not being cost-effective. My work departs from standard practice, in ways that dramatically increase cost-effectiveness. For systems which admit understandable specifications, I have focused on techniques for raising the level of abstraction in proofs, replacing long sequences of low-level reasoning steps with more human-readable descriptions that can adapt to the proofs of many related theorems. Several of my projects have shown how this style of proving can reduce the costs of *evolving* verified programs to the point where formal proof may even be cheaper than rigorous testing. I have also taken a technology that has historically been confined to proof assistants, namely dependent type systems, and incorporated it into a very practical language for Web application development. This language implementation also uses proof automation in several different ways, so that programmers can check important properties of their applications, without needing to spend much time on tasks whose only purpose is to help the compiler check a program.

Past and Current Projects

Formal Correctness Verification of Programming Language Tools

The first part of my PhD thesis was a library for building memory safety verifiers for x86 machine code [1]. This was a library for the Coq proof assistant, providing support for formal proof that any program accepted by a verifier really is memory-safe. Each verifier was organized as a tower of components that successively raised the level of abstraction. For instance, different components add support for understanding the call stack, conditional instructions, and custom type systems.

The second part of my thesis dealt with compiler verification. Most work in that area has considered compilers for low-level languages like C. My interest has mostly been in verifying compilers for higher-level languages with higher-order features. The verification from my thesis [2] is for a compiler for a small statically-typed functional language. This was the first formal verification considering a compiler that is *type-preserving*; that is, the intermediate languages of the compiler retain type information, and output assembly programs pass that information to a garbage collector interface. The final, machine-checked theorem of this work is that, if a source program exits with a particular return code, then its compiled version also exits with the same code. The structure of the proof mirrors the structure of the compiler, which uses the classic functional programming language transformations of conversion to continuation-passing style and closure conversion. The proof must argue that type information is used correctly in driving garbage collection, where the collector is specified at a lower, language-independent level.

My most recent work in this area [6] involved a more ambitious case study, where I applied a novel syntax representation [3] to a more realistic language in a compiler that performs some optimization. The compiler's input language is an untyped “Mini-ML” language that retains the key semantically-interesting features of the ML language family, including first-class higher-order functions, exceptions, and heap-allocated, linked, mutable data structures. This was the first formal compiler verification for a language that combines higher-order features with side effects, yet the proof was an order of magnitude simpler than those previously published for formal verification of realistic, side effect-free functional languages. The full development includes under 2000 lines of proof script, almost all of which describes theorem-specific proof-finding programs, rather than details of proof cases. These programs proved to be very adaptive in the face of the evolution of the compiler and its specification. For instance, I was able to add some features to the source programming language without changing a single line of proof.

Ur/Web, a Domain-Specific Programming Language for Web Applications

Most Web applications today are implemented in general-purpose languages. These are usually *dynamic* languages, where the language tools provide very little support for static checking. I have designed the Ur/Web language to demonstrate how to bring pervasive static checking to this domain without giving up flexibility. Programs in this domain-specific language type-check only when they “make sense” as Web applications, with no faulty intra-application links, no syntax or typing errors in communication with a database back-end, and so on. Security bugs like code injection vulnerabilities are also ruled out naturally.

Prior work by others has considered several ways of achieving this kind of result. In this project, I have gone further by adding support for *statically-typed metaprogramming* [5], where an Ur/Web function can produce pieces of applications customized to database schemas and other dimensions of variation. Metaprogramming for the Web is already in wide use with dynamic languages. Ur/Web supports the popular kinds of metaprogramming found in the wild today, and any Ur/Web metaprogram that type-checks will never produce an application piece that violates the rules sketched in the prior paragraph.

My work starts with a core statically-typed language Ur, whose type system is expressive enough to encode the validity rules of HTML and SQL in the type signatures of libraries. The Ur type system is based on ideas usually only found in dependently-typed programming languages, which have tended to provide very limited type inference. A customized inference engine for Ur applies theorem-proving techniques heuristically such that it is possible to write sophisticated metaprograms without doing anything that feels like formal verification.

In Ur/Web, program elements like SQL database access are represented very explicitly, which enables some very useful sound program analyses. For instance, I have implemented the UrFlow analysis [4], which does static checking that Ur/Web applications follow information flow and access control security policies. UrFlow uses a new approach to policy specification, where policies are SQL queries that, in a sense, select allowable program behaviors. An information flow policy, which specifies which information clients are allowed to learn, is an SQL query that selects a subset of the database which is safe to reveal. To support authentication, policies may refer explicitly to *which secrets the user knows*, where the user is only presumed to know values contained in the request his Web browser sends. This primitive enables handling of a variety of access control schemes, without requiring any program annotations or run-time instrumentation.

Cost-Effective Verification of Low-Level Programs with Interactive Theorem-Proving

The Ynot project supports the implementation and formal verification of higher-order imperative programs in Coq. I led a reimplementation of Ynot with a focus on proof automation [7]. To verify simple data structures, the previous version had required tens of lines of proof per line of program code. The new version provides a library of parameterized proof procedures that can discharge most verification obligations completely automatically. The programmer proves a set of lemmas about his data structure and then feeds them into the generic procedure, which applies the hints as needed. With this style of proof, the development cycle changes dramatically, as formal verification of a new function for an established data structure can actually take less effort than building a credible test suite.

Ynot supports all of Coq's high-level language as a subset of the imperative programs. I am now working on a Coq framework called Bedrock, which brings the Ynot proof style to programs that are more suitable for implementing the lowest levels of software stacks. Several recent projects by others have verified assembly code for pieces of operating systems. Each of these projects falls into one of two main categories. The more traditional category relies on automated theorem provers, which requires trust in large amounts of complicated prover code and fails to handle some kinds of higher-order reasoning that are very helpful in systems infrastructure. The other category uses general-purpose proof assistants that have small, trustworthy proof checkers, at the expense of verification that may involve hundreds of lines of proof per line of program code. With Bedrock, I have replicated some of the results from this latter category, but with completely automated proofs, reducing the verification cost by orders of magnitude while retaining the benefits of general-purpose proof tools. For instance, I have built a simple verified cooperative threading library in 250 lines of code, including program code, specifications, and proofs.

Future Directions

Verification for Systems

For the code that forms the foundation of our software systems, we should demand languages and tools that provide excellent support for formal verification. A common complaint against formal verification is that it requires formal specifications, which can be prohibitively expensive to build. One surprising thing about systems software verification is that specifications are often very short and comprehensible. For instance, the correctness of a compiler can be phrased solely in terms of semantics of the source and target languages, and we can say an operating system kernel is correct when its proof may be used as a lemma to establish the correctness of arbitrary applications running on top of it.

Most systems infrastructure is used often enough that, to me, the cost of formal verification seems well justified. I also think that this cost can be brought much lower than the standard wisdom assumes, and I hope my work on compiler and data structure verification has provided a hint on how proof costs may be kept down. Furthermore, we need not stop at verifying the same old kinds of software that exist today. We can come up with new, simpler designs that integrate verification thoroughly. For example, using the technique of *proof-carrying code* [9], we can enforce that a particular OS kernel will only run applications about which certain key properties have been proved formally. In such a setting, we can do away with all hardware mechanisms for protecting software from software, and we can do away with the systems code that drives those mechanisms.

Concretely, I would like to develop proof-of-concept platforms for mobile phones and shared "cloud computing" servers, based on pervasive use of proof-carrying code. Operating system designs like those coming out of the exokernel project have provided resource multiplexing without abstraction. Still, these systems export specific run-time interfaces and use much of traditional hardware protection. Performance goals force the addition of complexity, such as the use of fixed packet filter languages to support high network throughput. I would like to explore an alternate approach, where "the kernel" has almost no run-time code. Rather, traditional protection functions are the responsibility of a general-purpose mathematical proof checker. Specifications formalize which hardware resources are available and how each program would like to use those resources. Each program must come with a formal proof that its claims are accurate. Application authors would be free to design their own abstractions, without any requirement of compatibility with a fixed run-time interface.

This architecture seems generally useful in computer systems, but I see an especially convincing match for the settings I mentioned above. On mobile phones, severe resource constraints mean that there is much to be gained by minimizing the use of hardware protection or software bloat that results from conforming to fixed system abstractions. There are also thriving ecosystems of untrusted applications for systems like the iPhone and Android. The iPhone App Store in particular is notorious for an arbitrary-seeming process of manual application approval by a team at Apple. There are good reasons for consumers to want oversight of applications, to prevent blatant security problems. However, with a pervasive proof-carrying code architecture, applications can be constrained cheaply, with no human inspection, based entirely on checking of formal proofs provided by app authors. The most untrusted application can be run with no run-time monitoring or other consumer-visible overhead. The flexibility of mathematical proof leaves all of this compatible with a broad range of specialized resource-usage policies and a broad range of programming techniques for adhering to those policies. All of these same arguments apply to shared Internet servers, where the performance goal is to maximize the number of mutually-untrusting applications that can be run per server, rather than maximizing the performance of one application per phone.

Synthesis for Applications

Most high-profile applications are written in general-purpose programming languages, where it is often difficult to disentangle the essence of a program from the data structures and other low-level techniques used to realize that essence. Program synthesis is a venerable research subject that involves automatic construction of programs from their logical specifications. The boundary between “specification” and “program in a high-level domain-specific language” can be blurry, and I am interested in both perspectives.

There has been a lot of work recently on what I call *combinatorial synthesis*, where programs are synthesized from specifications via different kinds of reduction to SAT. The most direct realizations of the idea only apply soundly to programs with finite input domains. I would like to explore architectures for *symbolic synthesis*, which would be based more on proof search in the style of logic programming. Programmers could build libraries of verified recipes for implementing partial specifications, and an extensible engine would figure out how to combine these recipes to construct particular programs. One example of recipes would be for data structure implementations, making the combining engine very similar to a query planner in a database management system. I envision these data structures being implemented in Bedrock, building on my past work in cost-effective verification of data structures. The new contribution would be automatic data structure selection for high-level code that could look more like mathematics than like most of today’s programs. Where the (smart) brute force enumeration behind combinatorial synthesis would not be expected to invent splay trees, symbolic synthesis would allow programmers to implement and verify splay trees as a library module, such that the overall engine could integrate that data structure with many others, producing an efficient low-level program with a formal proof of correspondence to the original.

I also want to continue my work on the Ur programming language family. Currently, Ur/Web is the only family member, but the domain-independent functionality of Ur is well-delineated. I would like to study extensible compiler architectures that would make it possible to build new domain-specific languages on top of Ur as painlessly as possible. This would include parsing extensions, code generation requirements, and specialized optimization rules. Ur has an expressive enough type system that I believe many domains could get by without any new implementation work on type-checking or type inference, but there are also interesting questions in extensible type inference engines. I would also like to do formal verification of an extensible compiler, via a proof parameterized on obligations that must be satisfied by the creators of each new domain-specific language.

A Few More Directions

In general, I’m interested in improving human productivity in mechanized theorem-proving and language implementation.

Most of the effort in implementing the Ur/Web compiler has gone into debugging a significant number of compiler passes that are run to simplify uses of abstraction, modularity, and higher-order features. Even after much debugging, the compiler still uses significant amounts of time and memory. I have a hunch that this usage can be reduced by relying on a framework for automatic combination of declarative specifications of program simplification rules, and I would like to explore that technique, both in the setting of Ur and for more canonical functional languages like ML and Haskell.

My work on formal verification of compilers has left me unsatisfied with the options for reasoning about the syntax of higher-order languages. Several tools today build on a very attractive formalism called Edinburgh LF [8], but none of these tools provide support for scripted proof automation. To me, this is a critical weakness, especially when it comes to combining proofs about syntax with proofs about other mathematical objects. I would like to aim for the best of both worlds by implementing an LF library in Coq, with associated proof tactics.

I'm also interested in exploring how my formal proof approach can be applied in further mathematical domains, including algorithms and complexity theory.

References

[1]
Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *Proc. ICFP*, 2006.

[2]
Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, 2007.

[3]
Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proc. ICFP*, 2008.

[4]
Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. OSDI*, 2010.

[5]
Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. PLDI*, 2010.

[6]
Adam Chlipala. A verified compiler for an impure functional language. In *Proc. POPL*, 2010.

[7]
Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proc. ICFP*, 2009.

[8]
Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. of the ACM*, 40(1):143—184, January 1993.

[9]
George C. Necula. Proof-carrying code. In *Proc. POPL*, 1997.
