My research revolves around programming languages. I aspire to use techniques from the field of programming languages to improve the quality of software. My current research concentrates on static (compile-time) analysis of programs. In my dissertation, I developed novel static analyses and applied them to several problems including to detect races in C programs, to compile programs with atomic sections into programs using locks, and to provide compile- and run-time support for transparent futures and asynchronous method invocations in Java.

I am fascinated by many areas in programming languages, like type theory, static analysis, formal proofs and semantics, and also the elegance of the logic and mathematics behind them. I believe one can directly apply the recent theoretical results and advancements in these fields to solve real-world problems. Conversely, one can draw motivation from practical issues rising in software, to formulate interesting theoretical problems. My aim is to employ techniques from theoretical programming languages research to attack current, actual problems with software; back my solutions with strong theoretical reasoning and proofs; and try to generalize the resulting techniques into novel theoretical results. The remainder of this document presents in more detail the research that I have done so far, and considers future directions I plan to pursue.

Data race detection for C

The increased availability of multi-core processors tends to make concurrency the norm even for the average programmer. Currently, locks constitute a very widely used synchronization mechanism in concurrent programs. Errors in using locks can lead to data races, with possibly catastrophic consequences.

Working with my advisors Michael Hicks and Jeffrey Foster, I developed LOCKSMITH [4], a tool that uses both well-known and novel techniques in static analysis to find races in C programs. LOCKSMITH aims to be a practical tool requiring no or minimal annotations from the programmer, and is able to handle all the commonly used patterns in concurrent programming, producing verbose warnings. To test the practicality of LOCKSMITH, I used a wide range of benchmarks, totalling more than 200,000 lines of code, and found many races, several of which were actual bugs. I formalized the novel context-sensitive *correlation analysis* used to infer "guarded-by" relations between locks and memory locations in LOCKSMITH, and proved it sound.

Transparent proxies in Java

A proxy object is a surrogate or placeholder that controls access to another object. Proxies can be used to support distributed programming, lazy or parallel evaluation, access control, and other simple forms of behavioral reflection.

Working with Michael Hicks and Jaime Spacco, I developed PROXYC [6], a tool that adds transparent proxies to Java. PROXYCallows the programmer to treat a proxy as the actual object, tracks the flow of proxy objects in the program, and automatically inserts a "claim" (unwrapping of the proxy) at every concrete use of the object. We used PROXYC to, among other applications, add asynchronous method invocations to Java, where a method call is evaluated in a separate thread, while the caller continues using a transparent proxy of the result (called a future), until the result becomes available. At the core of PROXYC is a type-qualifier inference framework, which we use to track the flow of proxies in the program, and insert the necessary claims (or synchronization, in the case of futures) wherever necessary. We formalized the system for a subset of Java and proved its soundness.

Inferring locks for atomic sections

The notion of *atomic sections* is a mechanism for limiting concurrency in a program. Namely, an atomic section is a section of code that is guaranteed to appears to be executed atomically, in isolation from other

threads. In comparison to code using locks, atomic sections are easier to use, and provide a higher-level approach to synchronization. Atomic sections are traditionally implemented using transactional memory, although in some cases that is not feasible, especially in the presence of I/O and other side-effects that cannot be rolled back.

Together with Michael Hicks and Jeffrey Foster, I have developed a tool called LOCKPICK [2] that compiles C programs with `atomic{}` blocks of code into programs that use locks for synchronization. LOCKPICK uses the same analysis engine as LOCKSMITH to infer the shared locations in the program and the minimum number of locks needed to protect them, without sacrificing any parallelism. I proved that the general problem of lock allocation for atomic sections is NP-complete, and presented a fast heuristic algorithm as an approximating solution. LOCKPICK does not require annotations (other than atomic sections) to infer locking, but does not use dynamic locks in data structures.

Static analysis of data structures

In LOCKSMITH, it is necessary to reason about the aliasing of lock variables that occur in the program. Motivated by that and the occurence of locks in recursive data structures that is somewhat common in C programs, I worked with Michael Hicks and Jeffrey Foster on a novel context-sensitive label-flow analysis with increased precision for recursive data structures [3]. The analysis is formalized as a constraint-based type system with existential and universal polymorphism, used to encode context sensitivity for function calls and data structure elements. I worked on proving the soundness of this generalized alias analysis, and reduced it into a context-free language (CFL) reachability problem, known to be solved in sub-cubic time.

Contextual effects

A very important part of LOCKSMITH and LOCKPICK is a sharing analysis that infers memory locations accessed by more than one threads in the program. At the core of that analysis is the computation of *continuation effects*, the effects of the program after a given point. Generalizing that idea, I worked with Iulian Neamtiu, Michael Hicks and Jeffrey Foster on a type-and-effect system for *contextual effects*, the effects of the program before and after any given expression. I formalized the system and proved its soundness, and also encoded the proofs using the Coq proof assistant.

Program verification

Proof-carrying code is the technique of annotating untrusted code with a formal, machine-checkable proof that it is safe to execute. Proof-carrying code is mostly used to verify simple properties like type- or memory-safety, but can also be used to verify more complex properties.

During an internship at Microsoft Research in 2006, I worked with Chris Hawblitzel and the Singularity Group on a proof verifier for proofs that annotate bytecode [1]. We used the calculus of inductive constructions extended with linear types as the proof language. I developed a type-checker for the proof language, which, by the Curry-Howard isomorphism, is also a proof-verifier for proofs in that language. I then used that checker on small programs to verify hand-written proofs that every array access is within bounds.

Mechanized proofs

Recently, there have been several attempts for mechanizing the metatheory of programming languages. Systems and proofs found in programming languages literature are usually large and complex. Therefore, the mechanization and machine-verification of interesting properties in programming languages strengthens the validity of a proof, by ensuring the absense of small mistakes that often arise in complex proofs.

Moving in that direction, I encoded the type system of contextual effects and its proof of soundness in the Coq proof assistant [5]. The statement of soundness for contextual effects has the unusual property that it depends on the position of an evaluation inside the evaluation derivation for the whole program. Mechanizing the formalism and proof of soundness for contextual effects emphasized that property, clarified the reason behind it, and highlighted what I believe are the unusual and interesting aspects of the proof. Finally, engineering the formalization and proof of soundness for the non-trivial type-and-effect system with contextual effects, lead to my deeper understanding of mechanized proofs and working with Coq.

Conclusions

In my dissertation, I developed several novel techniques in static analysis and type systems and applied them in several projects aimed to improve the quality of software. I developed several tools that assist the programmer in writing correct multi-threaded software, or aid them in fixing problems with existing software. In the process, I encountered and tackled interesting problems in the general fields of static analysis, type systems and verification, producing general solutions, backed up with rigorous proofs.

I consider my background in type systems, static analysis, formal proofs and programing languages in general, a strong asset in achieving the goal for better software. I aim to develop efficient and practical solutions to existing software problems, while also reasoning about their correctness with formal proofs of soundness.

I am interested in continuing to pursue my dissertation goal, towards safer, reliable, more efficient, easy to develop, quality software. At the same time, I am looking forward to discovering challenging problems that will motivate further study and research in related areas, expanding my research horizons in the field of programming languages and computer science in general.

References

[1] Juan Chen, Chris Hawblitzel, Frances Perry, Mike Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikakis. Type-preserving compilation for realistic object-oriented compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008. To appear.

[2] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[3] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Existential label flow inference via CFL reachability. In *Proceedings of the Static Analysis Symposium (SAS)*, Seoul, Korea, August 2006.

[4] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[5] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks, and Iulian Neamtiu. Formalizing soundness of contextual effects, February 2008. Submitted for publication.

[6] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.