Research Statement

Sorin Lerner

My research goal is to develop program analysis and language techniques for making software systems easier to write, maintain and understand. My experience in program analysis has spanned a wide spectrum of projects, ranging from implementing a low-level dynamic optimizer for x86 [*FDDO 2000*], to developing a scalable and efficient analysis in C++ [*PLDI 2002*], to proving theoretical properties about analysis frameworks [*POPL 2002*, *PLDI 2003*, *POPL 2005*]. I enjoy implementing practical solutions, while at the same time developing the theoretical foundations to clearly state what properties these solutions have.

For the past several years, my research has focused on the particular problem of improving software reliability: I developed a domain-specific language for implementing dataflow analyses and transformations that can be checked for correctness automatically; I developed an easy-to-program, and therefore less error-prone, technique for dataflow analyses to interact in mutually beneficial ways; and I worked with colleagues to develop a tool that not only finds API usage bugs, but can also prove their absence in large C and C++ programs. I present these three research projects in more detail, and then I describe my plans for future projects.

Correctness of Program Analyses and Transformations

The reliability of any program depends on the reliability of all the tools that process it, including compilers, static and dynamic checkers, and source-to-source translators: a correct program compiled incorrectly is effectively incorrect; a guarantee of memory safety (no buffer overruns) or security safety (no high-security information flows to low-security variables), if provided by a buggy checker, is in fact no guarantee at all. Unfortunately, program analysis and transformation tools (PATs) are really hard to get right, even for experts in the field of program analysis. It takes years before a new compiler is stable enough for programmers to trust it, and even then the compiler will still have numerous bugs. Aside from having an impact on software reliability in general, the difficulty of writing correct PATs also hinders the development of new languages and new architectures, and it discourages end programmers from extending PATs with domain-specific checkers or optimizers.

My thesis research is aimed at making it easier to build reliable PATs by providing the right abstractions for implementing them, while at the same time providing strong theoretical guarantees about their correctness. The most error-prone parts of a PAT are the analyses and transformations themselves. Each new program processed by a PAT potentially triggers new patterns of interactions among its analyses and transformations, and it is difficult to cover all these interactions in test suites. Furthermore, it is becoming less and less feasible to increase the reliability of a PAT by simply disabling most of its analyses and transformations: with the widespread adoption of systems whose good performance depends heavily on compiler optimizations – for example just-in-time compilers and higher-level languages like C# and Java – turning off the optimizer is no longer an option; with more and more systems aiming to provide strong guarantees, it is no longer possible to disable the static checkers that provide these very guarantees.

I developed a language called Rhodium (a previous version of which was called Cobalt) for implementing program analyses and transformations over C-like programs [*PLDI 2003*, *POPL 2005*]. In Rhodium, PAT writers declare *dataflow facts* that represent useful information about a program, and then they write *declarative rules* for propagating these facts across statements and for using these facts to trigger program transformations. Aside from reducing the potential for errors by makings PATs easier to write and maintain, the restricted domain of Rhodium makes it amenable to rigorous static checking that would be impossible otherwise. I have implemented a tool that leverages the stylized form of Rhodium analyses and transformations to check them for correctness *automatically*. Furthermore, Rhodium analyses and transformations can be run directly in our research compiler; they can be interpreted as flow-insensitive and/or context-sensitive or -insensitive interprocedural analyses; and they can be automatically combined to yield more precise solutions. Finally, Rhodium can also be used to improve the reliability of already existing PATs

by filtering the analysis and transformation results they produce. The Rhodium system is practical and useful: I have used it to implement and check the correctness of a variety of realistic analyses and transformations, including partial redundancy elimination, partial dead code elimination, arithmetic-invariant detection, loop-induction-variable strength reduction, and Andersen's pointer analysis with heap summaries.

Colleagues and I are currently taking the next step in making correct PATs easier to write: instead of requiring programmers to *write* the rules for propagating facts across statements, we are designing a tool that *infers* these rules automatically from a declaration of the kinds of dataflow facts to compute. Our current algorithm for automatically generating Rhodium rules employs a search technique similar in nature to those used in automatic theorem provers. On the cases that we have tried so far, our algorithm is able to infer almost all of the Rhodium rules we wrote by hand, and it also discovered useful new rules that we had not thought of previously.

Exploiting Mutually Beneficial Interaction between Analyses

Taking advantage of interactions between separate analyses is often critical for getting good analysis results. For example, it is well-known that running inlining and class analysis together is important for generating high-quality code for object oriented languages. Unfortunately, it is difficult to exploit these mutually beneficial interactions in a way that is both easy-to-program and efficient. Colleagues and I devised a new approach for exploiting mutually beneficial interactions that allows analyses to be defined easily and modularly, while still enabling them to be combined automatically and profitably [*POPL 2002*]. I co-designed the approach and implemented it in our research compiler. I also expressed our technique formally, proved its soundness, and showed that it can only improve (and therefore never worsens) the results of individual analyses.

Static Detection of API-Usage-Rule Violations

Software systems often include "API usage rules" that must be obeyed to ensure their correctness. For example, a system may require that a lock be released only if it was previously acquired, or that a file be written to only if it was previously opened. ESP is a sound program analysis tool for verifying that large C and C++ programs adhere to such API usage rules. The key challenge in building a tool like ESP is to design an analysis algorithm that scales to large programs, while retaining enough precision so that the user is not flooded with false positives. ESP addresses this challenge with a novel interprocedural property simulation algorithm that keeps track of only those predicates (expressions tested by branches) that matter for the property being checked. By examining only relevant predicates, ESP avoids the exponential blow-up associated with tracking *all* predicates, while at the same time pruning infeasible paths that would otherwise produce false positives. I was a member of the ESP research group at Microsoft Research, where I co-designed and implemented ESP's property simulation algorithm [*PLDI 2002*]. ESP has since been applied to the entire Windows kernel at Microsoft, and it has been successful at finding API usage bugs (and also proving their absence) in Microsoft production code under development.

Future Work

The goal of my future research is to broaden the impact of program analysis and transformation tools (PATs) by expanding the benefits they provide to programmers. One aspect of this agenda is making PATs reliable (so that they can be trusted), scalable (so that they be applied to realistic code bases), and efficient (so that they can be invoked on every compilation). I not only want to develop useful analyses that have these properties, but I also want to understand the underlying principles behind writing such analyses, and use this understanding to *automate* more and more of the PAT-writing process. The grand challenge that I propose is to automatically generate an efficient, scalable and correct PAT in its entirety from a simple goal-directed specification. Despite a long line of work dating back to the 70s on generating various parts of a PAT from specifications (for example the parser, the type checker, and the code generator), there has been little work on automatically *inferring* analyzers and optimizers, and on doing so in a way that guarantees their correctness. Some recent advances in this poorly explored direction are nevertheless encouraging, including

my own work on inferring Rhodium rules, and recent work by Amarasinghe *et al.* on inferring compiler heuristics using genetic programming. I believe that there are many more opportunities for novel research in this area, and I plan to investigate these opportunities. For example, one could develop tools that make analyses scalable by examining their run-time traces and then using the gathered information to guide the automatic application of various representation optimizations (such as switching to BDDs or bit-vectors for encoding dataflow information); tools to automatically (or semi-automatically) explore the tradeoffs between the scalability and precision of an analysis; tools that infer entire dataflow analyses from a specification of the desirable transformations; or tools that infer potential optimizing transformations from a high-level goal-directed specification. This research direction not only poses interesting challenges in the areas of programming languages, compiler optimizations, and formal methods, but also provides opportunities for applying techniques from other research areas to program analysis problems. For example, I believe that search techniques from AI and from the theorem proving community will be helpful in generating analyzers and optimizers automatically.

Another research direction that will broaden the impact of PATs is to empower end-user programmers with the ability to create their own domain-specific analyses and transformations. Oftentimes, it is the end programmer who has the application-domain knowledge and the coding-pattern knowledge necessary to implement useful analyses. My thesis research has provided a practical foundation for extensible PATs: end programmers, with little or no knowledge of program analysis, can now easily extend existing PATs without fear of breaking them. I want to use this foundation for investigating how domain-specific static checking and domain-specific optimizations can be made practical and useful. I also want to develop domain-specific languages that, like Rhodium, are targeted at a particular domain, and as a result can provide strong guarantees via rigorous static checking. More broadly, I want to investigate techniques for allowing end-user programmers to develop their own lightweight domain-specific languages that can be easily incorporated into mainstream languages, while still providing strong guarantees. I look forward to collaborating with colleagues in other areas of computer science and with researchers in pure and applied sciences, to understand the requirements of domains other than the ones I am familiar with, and to investigate the possible use of analysis techniques for these domains.

Most broadly, my future research will address a range of issues in software engineering and software reliability. Using language and analysis techniques, I want to investigate how to find bugs statically in large code bases, how to concisely express complicated general-purpose code using declarative constructs, and how to compute static program information to help developers better understand software systems. As in my previous research, I plan to tackle these and other problems using a combination of language design, analysis techniques, theorem proving technology, and formal theory.