

Summary

My research interests are in programming languages and compilers, focusing on the use of type theory and logic as a foundation upon which to build practical and useful programming tools.

- Programming languages provide the mediums in which the vast majority of the art and artifacts of the computer profession are expressed. As such, I believe that this area of research is fundamental to the advancement of computer science. Without good programming languages in which to express our ideas, we are greatly limited in our ability to leverage automation to manage complexity.
- A compiler serves as the interface between the abstractions of a programming language and the concrete details of a machine. Without good compilers, the utility of a programming language may be limited by performance costs. Good compilers are essential to making good programming languages practical.
- A type system provides a means for programmers to write down facts and invariants about their programs within their programs, in such a way that these facts and invariants can be *automatically* checked. Types therefore provide a valuable tool for managing complexity by documenting and enforcing the key abstractions of a program. The ability to automatically check properties of programs is important for productivity and for security. A compiler can also take advantage of the invariants encoded in types to produce better code. In this way, types serve both as a tool to *check* facts about a program and to *communicate* facts about a program to a compiler.

My research is therefore focused on exploiting the deep connections between logic and type theory to make programming languages practical, useful, and secure.

Research Interests

An important goal of programming language research is to allow programmers to say what they mean – that is, to write down important properties and invariants of a program within the program itself. Many programming languages have only limited facilities for statically enforcing correctness (or even safety!) properties of programs. Programs written in these languages must therefore rely on properties that are enforced only by programmer convention. Languages with more powerful type systems allow some of these properties to be expressed in the program itself in such a way that the compiler can statically check that the properties are satisfied. Relatively well-known ideas from type theory (such as parametric polymorphism) are already making their way into mainstream languages: but these are merely the tip of the iceberg. The key ideas for the next generations of language technologies are already being developed: modular type systems, dependent types, refinement types, modal type systems and many others. I have a strong interest in using our growing understanding of the foundational underpinnings of these ideas to find ways to make them both useful and practical.

In order to be practical, a programming language must admit efficient implementation. Many of the properties of programs that are important to a programmer for safety and correctness are also of interest to a compiler for performance. A compiler spends a great deal of time trying to discover properties of programs

that it can exploit to produce better code. In many cases, these properties are apparent in the safety and correctness properties designed into the original program by the programmer. I am very interested in finding ways to use the annotations that programmers write for safety and correctness purposes (such as types) to benefit the compiler as well as the programmer. This is particularly important in the increasingly important setting of separate compilation, where small pieces of programs must be optimized independently using only programmer specified interfaces.

High-level programming languages attempt to provide good facilities for expressing invariants of programs in the program code. An objection that is sometimes raised to these languages is that they do so in part by abstracting away from the machine in ways that prevent programmers from writing programs that interact closely with the low-level machine model. On the other hand, the programming languages that do provide good access to the underlying machine do not provide much in the way of correctness or safety guarantees. This then is another important goal in my research: the need for high-level programming languages that allow for the possibility of fine-grained access to the underlying machine model without sacrificing safety and correctness.

Safe programming languages provide the kinds of security guarantees that are vital in a world where code is becoming increasingly mobile and computers and devices are becoming increasingly exposed to open networks. Bugs in programs written in unsafe languages account for great numbers of security holes despite prodigious efforts to find and eliminate them. Language based security techniques hold great promise for automatically eliminating large classes of exploitable bugs. I am interested in two different directions of research in this area: firstly in extending the class of safety and correctness properties for which automatic checkers and certifiers can be written, and secondly in extending the class of programs for which automatic checking can be done to include programs which now must be written in unsafe languages because of the need for efficiency and low-level access to the machine.

Research Experience

My initial exposure to programming language research came as an undergraduate working with Dr. Kim Bruce on an object-oriented programming language called LOOM [BFP97, Pet96]. Many type systems for object-oriented systems are fairly restrictive, requiring numerous dynamic typecasts to type common programs. In LOOM, we explored a new type theoretic construct based on an axiomatization of the inheritance relation. By separating the notions of inheritance and subtyping, we were able to provide a much more flexible type system [BFP97]. This work led into my undergraduate thesis [Pet96], in which I designed a module system for this extended language supporting a very rich notion of abstraction, including the ability to use modules to abstract some methods of an object while exposing others (useful for, among other things, capturing the C++ notion of “friends”).

My work on LOOM sparked my interest in finding ways to make programming languages more expressive. At the same time, I also became convinced that the notions of modularity and abstraction are essential components of programming languages that are under-represented in mainstream languages. In most programming done today, the ability to compile programs abstractly against the interfaces of libraries or other program components is essential. However, the language facilities and compiler technologies for managing this style of programming have lagged behind.

As a first-year graduate student at Carnegie Mellon University, I began working on a new compiler under development called the TILT/ML compiler [PCHS00, VDP+03]. TILT is an optimizing compiler for full Standard ML which has the unusual feature of preserving types throughout the compilation process. This is especially challenging since SML is a language with a very rich type system, and one of the only existing languages with a full-scale module system supporting separate compilation.

TILT uses type information both to check the correctness of its generated code, and also for the purpose of optimization. Type information is exploited to allow for non-uniform data representation and to do (almost) tag-free garbage collection. TILT demonstrates the idea that type information that is useful to the programmer as a safety and correctness check can also be useful to the compiler.

For my doctoral thesis, I chose to extend the TILT compiler to push type information even further into the compiler, producing object code annotated with type information in the style of the TAL typed assembly code project. These annotations provide a sort of certificate that can be used to check that the generated

code satisfies the safety properties guaranteed by the type system. In this way, the safety properties of the source language can be preserved as safety properties of the resulting binary.

In the course of my thesis, I have also been exploring a type system that allows programs to manipulate data at a very low-level without sacrificing type safety. This type system is based on a restriction of linear logic to allow programs written in a lambda calculus style to nonetheless have precise control over the layout of data in memory and the order and manner in which data is initialized. We presented some preliminary results of this work in a paper published this spring [PHCP03]. This work is very exciting to me because it provides some insight into the problem of reconciling two dominant programming language idioms: that of the low-level language which provides access to details of the underlying machine model at the expense of safety, and that of the high-level language which gives safety and abstraction at the cost of denying the programmer access to the bare machine.

Research Directions

I am very interested in continuing to build upon my recent work on data representation. The ability to carefully control data representations in a typed language could be useful in a number of settings: in general purpose languages, in the realm of domain-specific languages for talking about data, and for use in typed intermediate languages for compilers. One of the promising aspects of our approach is that it accounts for low-level issues in a very high-level manner similar to a standard lambda-calculus style programming language. This suggests the possibility that it might integrate well with existing user-level programming languages.

I am also interested in continuing the work on type-preserving compilation that I have done prior to and as part of my thesis. While the core ideas of a type-preserving compiler and typed compilation have been demonstrated in TILT, I believe that there are many interesting practical research problems that remain to be addressed in this area. Among many other things, I am interested in extending the sorts of internal languages used in TILT to serve as a target for a wider range of language constructs, such as objects.

One of reasons that I am excited about programming language research is that I believe that many of the key elements needed to advance to the next level are already in place. A great deal of hard foundational work has been done that is now paying off in the form of a wealth of new ideas and understanding. The challenge for the next decade is to take these ideas and use them to make better tools to help programmers write better programs.

This is the sort of thing that makes programming language research interesting to me.

References

- [BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good “match” for object-oriented languages. In *Extended abstract appeared in ECOOP '97 Proceedings, LNCS 1241*, pages 104–127, 1997.
- [MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.
- [PCHS00] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, School of Computer Science, Carnegie Mellon University, December 2000.
- [Pet96] Leaf Petersen. A module system for loom. Undergraduate Thesis, Williams College, 1996.
- [PHCP03] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *Conference Record of POPL 03: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 172–184, New Orleans, LA, January 2003.

- [VDP⁺03] Joseph C. Vanderwaart, Derek R. Dreyer, Leaf Petersen, Karl Crary, and Robert Harper. Typed compilation of recursive datatypes. In *Proceedings of the TLDI 2003: ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 98–108, New Orleans, LA, January 2003.