Statement of Purpose

Research

My interests in Computer Science are fairly wide-ranging: in graduate school, I did work in artificial intelligence, cognitive psychology, and programming languages. Since then, I have worked in information infrastructure and mobile computation, and consulted on database systems.

The language thesis

However, there has been one central problem to which I have returned consistently: supporting what I have come to term "the language thesis," that is, the notion that language matters. Software engineers construct patterns of computation for a living. Programming languages are our main tool for describing these computations. I am fascinated by the implications of these notations upon the task of constructing large, complex computational systems.

In particular, I am interested in developing advanced programming languages as practical tools for systems programming. An example is scsh, a systems-programming environment embedded inside Scheme, which I developed to experiment with the utility of functional languages for real systems work.

Unfortunately, these languages are limited in applicability by efficiency; they are not competitive with C or FORTRAN. Here, the research goal is to make programs written in Scheme and ML as efficient as their counterparts written in C and FORTRAN. My thesis work provides one of the basic techniques necessary for implementing highly-optimizing Scheme and ML compilers: data-flow analysis.

Much work remains to be done for these languages. Can they be compiled for parallel machines, in several possible models of parallelism, *e.g.*, VLIW, scan-vector, or MIMD? Can the data representations be optimized in the presence of polymorphism? Can the traditional body of data-flow optimizations be mapped over to these languages, using the techniques outlined in my dissertation? Can we mine the work done in the optimizing FORTRAN community, recasting it in the lambda-calculus framework of functional programming languages? Can techniques developed for PDG-based compilers be adapted to CPS-based compilers? Can CPS lambda-calculus representations be modified to expose fine-grained parallelism, and retain the advantages of a formal semantics based on the lambda calculus? There are many, many interesting issues here.

Languages and operating systems

It is now commonplace to assume that compilers and architectures should be designed in tandem. However, the interaction between the compiler and the operating system is still relatively unexplored, particularly in the cases involving advanced languages. In my group, we have developed a research platform, ML/OS, which is an implementation of Standard ML that runs on a bare PC, with no underlying operating system. We have begun to use ML/OS as a tool to investigate the possibilities of the language-centered operating system.

Operating systems in the service of advanced languages. Unix and C are mutually supporting: C is used to implement Unix, and Unix provides the run-time services that the C model requires. Implementing a sophisticated Scheme or ML system on top of Unix requires one to implement a small, virtual operating system on top of Unix, tuned to the requirements of these languages. Interrupt handlers, storage allocators, linking loaders, and I/O all must be written to support the model expected by advanced languages. It is interesting to explore the structure of an operating system that directly provides the services required by advanced languages, such as persistent heaps, or lightweight user trap handlers.

Advanced languages in the service of operating systems. We can also exploit the benefits advanced languages provide to operating systems. For example, partial evaluation and type systems based upon modal logic may be able to provide language-level support for the sort of order-of-magnitude optimizations Massalin realized in the Synthesis kernel.

As another example, perhaps the only benefit provided by operating system "microkernel" architecture is modularity. But microkernels pay a high run-time price for their modularity: system calls must cross multiple process boundaries. Advanced programming languages, however, provide modularity mechanisms that do not have run-time artifacts. A program written in a "safe" programming language, for example, can be run in supervisor mode, and system calls can be open-coded. The techniques employed in high-performance functional-language implementations for atomicity can be applied here, as well.

One of the more fascinating elements of functional programming-language theory is the notion of continuations. These are extremely general control operators that can be represented in lambda calculus; as such, they are subject to the invariants of the calculus, and its related analyses.

My thesis explored ways to apply this theory to classical compiler optimizations. Lately, I have found two other applications of continuations. First, in a language-centered operating system, we can provide fully virtualisable thread models that are based on continuations, and consequently subject to compiler-based optimization. Second, we may describe the composition of on-line transducers, such as network-protocol layers or DSP modules, using continuations. Doing so allows continuation-based analyses to optimize across the composition.

Theory for practice

Engineers draw upon a body of theory that enables them to construct and optimize artifacts that have predictable behavior: bridge builders use statics, and electrical engineers use Kirchoff's laws. Computer science supports software engineering to the extent that we provide programmers with a similar body of theory. Unfortunately, our record is fairly sparse, although algorithmic analysis and type theory are two examples of theories that support the practice of software engineering.

I am interested in finding ways to apply theoretical computer science to the engineering task of constructing complex software systems. For example, I am currently supervising a doctoral student in my group who is searching for an extensible annotation system that will allow a programmer to make statements about his program that can be automatically checked by a software tool, and that may also assist the compiler. My student has developed a novel deduction system for this task. Corresponding to a mechanization of the entailment relation in logic, a "program" in this notation is a proof; executing the program produces either a valid theorem or an exception. What is important is that proofs in this notation appear to be more compact and intuitive than proofs in classical deduction systems—the hope is, after all, that programmers will be able to use this notation. What is also striking is the impact his use of the elements of program-language design—abstraction, scope, naming—has had on the power and compactness of this proof notation.

While I do not believe it is practical today to pursue complete program verification, programmers know many facts about their programs that cannot be encoded in the program text. If programmers could state these facts, and their support, then automatic tools could check to ensure these statements when the program changed during development or maintenance. Furthermore, some of these statements, such as "Procedure P is associative," or "Data-structure D may be allocated on the stack," may enable compiler optimization. What we are attempting to do is push beyond the limits of type systems, to systems that are sound but not complete.

Teaching

As with many, my style of teaching is based on the examples set by those teachers who have inspired me. In terms of the mechanics of teaching, I believe that teaching is a performing career. Although my task is to educate, not entertain, I need to capture and hold my students' attention in spite of their fatigue, other courses, and the many distractions of undergraduate life. Finding ways to engage students mentally, to get them to be intellectually and emotionally invested in the class is what enables interaction—and this is what distinguishes the classroom from the library. In the past I have run a denotational-semantics variant of the "Jeopardy" quiz show in the last class before the midterm. I name homework problems after the student who asked a related question or proposed a related device during a prior lecture. The aim is always to get students to "attach" to the material.

When I teach, I seek to find the few deep principles that underly the mass of material the students must study; I emphasize these principles at every possible opportunity. Some examples of these principles or themes would be "syntax and semantics," "notation and computation," "levels of machines," "a program is a document," or "the role of abstraction and modularity." A handful of ideas can, over the course of a semester, illuminate a wide array of material. This is, of course, easier to do when teaching a course on which one has passionate opinions.

I also believe that there is a moral element to teaching. An instructor has an obligation to support and teach a code of ethics, and this means clearly establishing and enforcing a policy on coursework. Following through on this has always been the most emotionally taxing component of my teaching.