Mark Leone

Research Overview

My research centers on the design and implementation of programming languages, with an emphasis on performance and support for systems programming. I am interested in programming languages with solid theoretical foundations, but my methodology is experimental. In my opinion, language designs and compilation strategies profit considerably from application to real-world systems. There is a valuable synergy between language design and systems building: good language design promotes elegant system structures, and real-world systems impose hard constraints on the implementation of language designs.

The so-called "software crisis" that was proclaimed a decade ago has been exacerbated by new demands on software systems, such as distributed operation, increased security, improved reliability, and easier reconfiguration. These concerns are directly addressed by advanced languages such as Standard ML, which facilitates good system design with parameterized modules and sophisticated data structuring and control constructs, and which provides safety guarantees through strong typing and robust run-time error detection. However, such languages have not been widely adopted. I believe advanced languages will languish as academic curiosities unless substantial progress is made in improving the efficiency of their implementation and convincingly demonstrating their utility to builders of real-world systems. I intend to undertake both of these endeavors.

Currently I am investigating the use of run-time code generation for the optimization of programs written in Standard ML. By postponing action until run time, an optimizer can make use of values and invariants that cannot be exploited at compile time. For example, a general-purpose matrix multiplication routine can be optimized at run time to exploit the sparsity of its input matrices, a property that may not be apparent at compile time. Although this can lead to significant improvements in code quality, there is a tradeoff that must be considered: the time saved by the optimized code is offset by the time spent generating it at run time.

I have developed a general technique, based on previous work in semantics-based program transformation, that reduces the cost of run-time code generation by an order of magnitude. My insight was that, although this kind of optimization makes use of values that are only available at run time, the code being optimized is known at compile time, and therefore *partial evaluation* can be used to shift most of the work of run-time code generation to compile time in a principled way.

A partial evaluator is essentially an aggressive optimizer that takes the source code of a two-argument function, f, and the value of its first argument, x, and returns optimized code for the result of *partially* applying f to x:

$$opt(f, x) \ = \ f_x$$

The result is a *specialized* one-argument function, f_x, that when supplied with an argument, y, computes the same function as $f(x, y)$, but does so without repeating computations that depend only on x.

The cost of run-time optimization can be reduced at compile time by optimizing the optimizer to the function that will be optimized:

$$opt(opt, f) \ = \ opt_f$$

The result is the code for a specialized optimizer and code generator, opt_f, that computes f_x without the overhead of processing *any* intermediate representation of f.

I have implemented a compiler for a subset of Standard ML that incorporates these concepts. It automatically compiles ordinary programs into code that generates native MIPS code at run time with extremely low overhead (fewer than ten cycles per generated instruction). Portions of a compiled program are "hard wired" to optimize and generate code for specific portions of the source code. Most aspects of compilation, including register allocation and code generation, are performed at compile time, but some optimizations, such as inlining, constant propagation, and certain kinds of instruction selection, are postponed until run time.

The results are promising. For example, the BSD packet filter interpreter, which the BSD kernel uses for fast selection of network packets on behalf of user processes, runs nearly 40% faster on typical inputs when translated from C to ML and compiled by my system. Also, a *general-purpose* matrix multiplication routine written in ML and processed by my compiler achieves performance that is comparable to a highly specialized sparse-matrix routine written in C.

My future research plans include further investigation of the use of run-time code generation in numerically intensive applications, operating systems, and distributed systems. The rapid emergence of Web-based agents and executable "applets" demands new strategies for ensuring safety and good performance; I believe that run-time code generation will play a crucial role. I also anticipate applying the techniques I have developed to other classes of programming languages, including conventional programming languages like C. My preliminary work in this area suggests that advanced program-analysis techniques will be critical to making automatic run-time code generation practical for imperative programming languages.

My research interests also include abstract interpretation (and other semantics-based program analyses and optimizations) and the type-theoretic foundations of modules, objects, and concurrency. For example, I am interested in developing techniques for optimizing across module boundaries in the presence of separate compilation. Usually the "signature" or interface specification of a module determines how its data structures and functions may be accessed. But this is overly conservative and precludes desirable optimizations such as the use of efficient calling conventions and specialized data representations. I would like to investigate formal module calculi that admit more efficient compilation techniques. Also, I believe that postponing optimizations until link time (i.e. when modules are composed) will permit more aggressive program transformations; the framework that I have developed in my thesis work appears to be a promising foundation for this line of research.

I hope to collaborate with both programming language theorists and software system practitioners in the future; both communities can benefit from greater interaction. In the words of Alan Perlis, "if a language doesn't affect the way you think about programming, it's not worth knowing."