

Faulty software has been estimated to cost as much as \$75 billion per year. Even a single fault can be expensive: NASA lost a \$125 million orbiter in part because of a programming error that mixed data in English and metric units. This error could have been prevented by a better programming language: researchers have developed languages with type systems that enable a programmer to express the scientific unit of a value, and such languages guarantee that values with different units will not be mixed.

Like support for scientific units, most new ideas in programming languages offer both expressive power and new guarantees. Another example is *software transactional memory*, which enables a programmer to say directly that a section of code must execute atomically, as opposed to writing code that implements atomic sections indirectly by acquiring and releasing locks. Software transactional memory guarantees that an atomic section executes atomically not only with respect to other transactions but also with respect to the rest of the code. And unlike a language with locks, a language with software transactional memory guarantees that transactions will never deadlock.

To evaluate a new language idea like software transactional memory, the best way is to observe how programmers use it in practice. To make this possible, we need an implementation which is good enough that programmers will use the idea. In most cases, this means a compiler that generates good machine code. Finding new ways to build good compilers is the goal of my research.

Cheap, reliable, adaptable compilers

To help develop and deploy new language features that people can use to write better programs, my research focuses on finding easier ways to develop reliable, adaptable compilers. I build compilers whose users demand advanced language features and performant implementations. When appropriate, I also use theoretical techniques to prove that my work provides high-quality solutions.

Minimizing the cost of developing portable compilers

Data shows that when a compiler becomes popular, its users eventually demand support for a variety of machines. But adding support for a new machine is costly, because writing an instruction selector requires intimate knowledge both of the machine and of the compiler's implementation. The current best practice is to live with this cost. Finding a way to do better has been a research problem of long standing. One way to do better is to encapsulate knowledge of the machine in a machine description and to encapsulate knowledge of the compiler in a tool that analyzes the description and generates an instruction selector. Tools developed during previous research have generated incomplete instruction selectors that must be completed by hand.

In my 2010 POPL paper, I presented novel theoretical and practical results for this problem. I proved that in principle, generating an instruction selector is an undecidable problem, which suggests why previous work did not generate complete instruction selectors. I also developed a new search algorithm and a novel pruning heuristic that work well in practice. My algorithm is based on Hoare logic: by reasoning about how a sequence of machine instructions will mutate machine state, my algorithm finds sequences of machine instructions that implement the compiler's intermediate codes. My algorithm generates complete instruction selectors for real machines, with no human intervention.

Compiler design that reflects programming-language theory

A compiler doesn't just compile programs; it is also an infrastructure for experimenting with new ideas. Unfortunately, experimentation is frequently inhibited, rather than supported, by the compiler's internals.

Part of my work is to develop better compiler internals with simpler interfaces, to make it easier to change compilers and try new ideas.

In the Quick C-- compiler, Christian Lindig, Norman Ramsey, and I investigated five mutable representations of control-flow graphs, and with each representation, both the flow-graph implementation and its clients were complicated by the need to maintain invariants on mutable pointers. We developed a novel, purely applicative representation of control-flow graphs that uses only immutable pointers, and we showed that both the implementation and the client code became simpler.

On top of our control-flow graph, we have built an optimization framework called HOOPL. HOOPL provides a simple interface which enables a compiler writer to define new optimizations using the elegant mathematical formalisms that appear in textbooks. HOOPL can also combine optimizations to implement more effective “superanalyses,” using an algorithm developed by Lerner, Grove, and Chambers. The optimization-combining algorithm makes temporary, speculative transformations of the flow graph; our applicative flow graph makes the implementation of this algorithm particularly simple. Working with Simon Peyton Jones, we are also using Haskell’s expressive type system to guarantee that no part of the compiler can produce a malformed control-flow graph. This guarantee applies even to new optimizations that will be written by others. We presented our control-flow graph at the 2005 ML Workshop, and we have a draft paper on HOOPL ready to submit to the International Conference on Functional Programming.

Verified compilation for real machines

A programmer may work hard to ensure that source code is correct, but the code that actually runs is correct only if the compiler translates source code to assembly code correctly. To show that that a compiler translates correctly, the state of the art is machine-checked proof. To construct such a proof, a researcher defines a formal semantics for the source language and a formal semantics for the assembly language, then proves, in terms of those semantics, that the compiler produces assembly code that is equivalent to the input program. But because hardware manufacturers provide machines without any formal semantics, it is not obvious how to write such a proof for a real machine. Previous work has avoided this difficulty by weakening the theorem: researchers define the semantics for an idealized machine, then prove that the compiler is correct with respect to that machine.

I plan to prove the correctness of a compiler that generates assembly code for real machines. This problem has two separate parts: developing proofs that apply to real machines and specifying semantics for real machines. To develop proofs that apply to real machines, I will generalize the techniques I developed in my dissertation work. To specify semantics of real machines, I will use machine descriptions; and I will develop techniques to check a machine description for correctness with respect to the hardware.

Proving correctness of the optimizations that give the best known performance

A compiler’s optimizer should get the best performance out of the machine and should be easy to prove correct—but these goals appear to be incompatible. Jack Davidson has shown that the best code is obtained by improving actual machine instructions, but optimizations written in Davidson’s style are not well-suited to formal proof. Sorin Lerner has developed techniques for building optimizers that can be proven correct, but they cannot use knowledge of the machine to generate the best code. I plan to work on the design of optimizers that are both machine-specific and easy to prove correct. A good first step will be to define a domain-specific language for writing optimizations, where the language is general enough to describe optimizations independent of any fixed representation of instructions.