The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: it may be a nuclear warhead exploding over one of our own cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.

— C A R Hoare, "The Emperor's Old Clothes". 1980 ACM Turing Award Lecture.

Types have proven an effective mechanism for increasing the reliability of programming systems. Results from several different areas – mathematical logic, type theory, automatic verification, language implementation, and software engineering – have been combined and used to integrate types with many programming systems. Historically, the use of types has been limited to high-level languages. In this setting, a type system gives a notion of safety by articulating certain errors that can not occur untrapped in well-formed programs. Compilers enforce the static rules of a type system by rejecting ill-formed programs. Run-time systems trap the remaining type errors. While this use of types has been helpful, recent work suggests that types need not be limited to high-level languages. Specifically, using types in intermediate and assembly languages allows type-theoretic properties to be proved at many different levels of abstraction and facilitates exciting projects including certifying compilers, proof-carrying code, and frameworks for running untrusted code.

There are several ways to increase the safety of programming systems by using types at intermediate levels. First, types are useful as an aid for compiler writers. When type annotations are discarded after source-level analysis on the traditional approach, information is lost. If instead, high-level language types are encoded in intermediate language types, then general properties about a compiler's behavior can be proved from its encoding of types. Also, individual programs can be shown to conform to specific properties using automatic verification tools based on type analyses. Hence, a compiler writer can have more confidence that her translation is correct. Second, types may be useful in analyses for program optimizations. Traditional optimization analysis is based on semantic and syntactic properties of programs. By adding type-theoretic properties as a new space to explore, it should be possible to enhance existing strategies and perform new optimizations. Third, we notice that if compiler writers can prove properties of intermediate representations from types, then regular users should be able to do so as well! A hot topic in current security research is the development of frameworks for executing untrusted code (*i.e.*, binary programs received from untrusted sources) safely. If the untrusted program is compiled via a typed intermediate language, then the type annotations in the generated code can be used as a basis for formulating and proving safety properties about the program. Using this framework, an individual can ensure she only runs programs that adhere to a set of core safety properties.

Many different groups are already researching the application of typed intermediate languages to programming systems. The ConCert Project at Carnegie Mellon, the TAL project at Cornell, the Flint project at Yale, Appel's work at Princeton on certifying compilers, and

Necula's work at Berkeley on proof carrying code all have typed (or similar) intermediate languages as a component of their research. The interesting question at this juncture is how to exploit these frameworks in order to produce systems of practical benefit to programmers. A student working on any of these projects will need to extend the existing intermediate languages and type systems in order to support the strong safety results desired. Thus, I propose investigating typed intermediate languages in general, with an aim towards developing logics and type systems that are needed for the applications described above. This work likely involves forays into theoretical areas of Computer Science as suitable type systems and logics are designed, evaluated, and chosen. However, I plan to keep a constant eye towards applying the results in systems offering practical safety benefits to programmers.

My interest in this field is stimulated by several research experiences in programming language topics. Between June 2000 and September 2001, I designed and implemented $\mathcal{LOOJ}$, an extension to Java. This language incorporates many features from $\mathcal{LOOM}$ [BFP97] into the Java language. Specifically, it supports parametric polymorphism (generic types), a *MyType* construct (useful for self-referential data structures and so-called "binary methods" [BCC$^+$95]), and a novel implementation on the stock Java Virtual Machine, with full support for the new types in the run-time system. A previous student implemented a partial compiler for the language. My contributions included two independent rewrites of the extended compiler. Along the way, I redesigned several key parts of the type system and the source-to-source translation of extended $\mathcal{LOOJ}$ forms into standard Java ones. I also added a second *MyType* to the language, more closely capturing the type of '`this`'. Finally, I formalized the key parts of the type system in a calculus (as an extension of Featherweight Java [IPW01]). This work resulted in an honors thesis [Fos01b], a workshop paper [Fos01a], and a talk at a regional seminar (NEPLS). Sun Microsystems recently acquired a copy of the $\mathcal{LOOJ}$ compiler to investigate applications of its implementation techniques to future versions of Java.

While type systems capture key properties of high-level programming languages where complex data structures are important, in hardware design, temporal properties are often more useful. Model checking is a popular technique in hardware verification because it facilitates automatic verification of temporal properties. During the 2001-2002 academic year, I implemented a model checker for SAFL [MS00], a parallel, functional hardware description language currently in development at the University of Cambridge. The SAFL compiler takes a high-level description of a hardware circuit (closely resembling a Haskell or ML program) and produces Verilog code implementing the design. In my project, I modified the SAFL front-end to extract models for verification. The models are gained by first translating SAFL programs to a simple assembly-like language and then using the straight-forward operational semantics of that language to produce a transition relation between states of registers. Temporal logic specifications are checked against these models using a standard model checking algorithm. I also implemented a translation to a more powerful model checking back-end, SMV. My system integrates a model checker with the SAFL compiler and allows temporal properties of SAFL programs to be verified automatically. This work is described in an undergraduate dissertation, [Fos02].

Last summer I worked on the Glasgow Haskell Compiler (GHC) as an intern at Microsoft Research. GHC currently uses a C compiler as its back-end. The advantages of targeting a language such as C rather than the assembly language of a specific architecture are clear: C compilers are widely implemented and produce very good machine code. However, using C as an intermediate language makes writing run-time system code very difficult. My work on this project involved retargeting GHC to emit C-- code. C-- [RPJ00] is a language designed to succeed both where C does well and where it falls short. Specifically, it is a robust assembly language that also supports a basic run-time system API. I also worked towards translating a large body of run-time system code to work with the new C-- back-end.

The common theme that runs through these research experiences is an interest in languages. I enjoy tinkering with language systems, and always hope to see useful tools result from my efforts. This academic year I am reading for an MPhil in the History and Philosophy of Science. My interests in this field lie mainly in the philosophy of mathematics, language, and logic and especially in the epistemology of foundations. While this work is not directly translatable to applications in Computer Science, it is fascinating to explore questions that are important to Computer Science from a new perspective.

As well as language research, I am also interested in pursuing a career that involves a significant teaching component. Though I have not yet had the chance to teach a course entirely on my own, I have worked as a teaching assistant whenever possible. As an undergraduate, I was a teaching assistant for introductory programming, principles of programming languages, and a mathematics course on algebraic and quantum cryptography. This year I am a supervisor for undergraduate courses on optimizing compilers, semantics, and specification and verification. This unique opportunity allows me to work in small tutorial-style groups with two to three students and to help them master difficult lecture material and prepare for exams. I enjoy helping others learn and I look forward to building on my limited teaching experience in graduate school.

Next year I hope to begin a research degree that investigates ways of making language systems safer. An important property of this goal, if it is to have any real impact, is that the safety results be accessible to programmers. Theoretical results are important because they provide the rigorous frameworks needed for true reliability. However, formal methods and theory are often too complex and unwieldy to be of much use to working programmers. In order to reach Hoare's goal of real reliability in programming systems, we need automatic or semi-automatic tools that make the task of proving programs correct accessible to everyday programmers. Typed intermediate languages offer great hope for developing these kinds of tools in the very near future. Like type systems in existing languages, they can be introduced to programming systems without requiring significant user expertise. One of the best ways that we can be vigilant about reducing the risk of both calamities and less catastrophic errors is by investigating the use of types in intermediate languages.

References

[BCC$^+$95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[Fos01a] John N. Foster. Rupiah: Extending Java with match-bounded parametric polymorphism, a ThisType construct, exact types, and persistent constructors. In *Proceedings of 7th Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2001.

[Fos01b] John. N. Foster. *Rupiah: Towards an Expressive Static Type System for Java.* Williams College Senior Honors Thesis, 2001.

[Fos02] John. N. Foster. *Model Checking for a Functional Hardware Description Language.* University of Cambridge Computer Laboratory, Part II Dissertation, 2002.

[IPW01] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[MS00] A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Lecture Notes in Computer Science: Proc. 27th ICALP, vol 1853.* Springer-Verlag, 2000.

[RPJ00] Norman Ramsey and Simon Peyton-Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM Symposium on Programming Language Design and Implementation*, 2000.