Research Statement

Stephanie Weirich

Motivation and Vision

My research interests lie in extensions of strongly-typed programming languages to support greater prescriptive and descriptive power. In doing so, I hope to improve the process of building reliable, secure, and efficient software.

Why concentrate on typed programming languages? There are a number of formalisms for program verification, but type systems have been one of the most practical. The types of a programming language are an abstraction of the execution of a program written in that language, and so may describe its properties. Therefore, the type system may verify that the program is used consistently and correctly. This facility leads to more secure software because security flaws in software have often resulted from malicious users deliberately supplying arguments that do not satisfy the preconditions of an operation. For example, an intruder may supply a string argument that leads to buffer overflow in order to compromise a software system.

The level of assurance about the reliability and security of a program is determined by the precision that we specify its operation. If that specification is too precise, it may restrict what programs may be verified and render the system impractical. Therefore type systems incorporate mechanisms for type abstraction in order to allow user control of the precision of the specification. Likewise, we may need to break the provided abstractions in order to verify certain operations through the use of run-time type checks. For example, if the type of an array statically records its length, but that length has been held abstract, we must check that an array access is within bounds at run time. Therefore, it is important that an expressive type system includes a facility to analyze types at run time.

There is another reason to incorporate mechanisms for run-time type analysis: Modern systems are increasingly becoming more adaptive and evolutionary. These frameworks depend crucially on *reflection* facilities, the ability of programs to dynamically observe objects and adapt to them. Many necessary operations can be specified only in terms of abstractions of these objects (*i.e.* their types). For example, functions such as garbage-collector tracers, data marshalers, and structural-equality functions operate over the structure of data instead of its value. Therefore, it is important provide a firm basis for the programming languages that will be used to implement these new systems.

Theory of run-time type analysis

My thesis work is based on theoretical results about the nature of run-time type analysis. In that work I investigated the linguistic and type theoretic support necessary to support run-time type analysis, and how it may be extended for more flexibility.

Representing types at run time A particularly important issue about run-time type analysis is how it interacts with the rest of the typed programming language. If the goal of the typed language is to accurately and explicitly model execution, such as in a typed intermediate or target language of a compiler, then the presence of run-time type analysis severely complicates the necessary semantics because types are also run-time objects. As an alternative, Karl Crary, Greg Morrisett, and I proposed the language λ_R which replaced run-time analysis of types with run-time analysis of explicit term representations of those types [3]. In that way, all computation is described by the term level. We connect these term representations to the types through a special form of *singleton type*.

Compiling and encoding type analysis In later work [1], Karl Crary and I discovered that the type system and its associated mechanisms for analysis need not be specifically hard-wired into a special term to perform type analysis. By programming at the type level in a sufficiently rich language, we may encode any type system, and simulate analysis over it. As a result, we may more easily support type analysis in the framework of type-based compilation. When compiling a language that supports type analysis, the target language must be able to represent an analysis of the source language's types. As a sufficiently rich target language, we proposed the LX language.

Based on the observations above, I found that outside of constructs for polymorphism, no other intrinsic facilities are necessary to support run-time type analysis [8]. Using standard encodings of iteration, I showed how to simulate the λ_R language with a version of the polymorphic lambda calculus. With this technique, we no longer need to incorporate sophisticated machinery into the semantics of a language in order to support run-time type-analysis.

Higher-order type analysis Finally, in my most recent work [7], I extend the frameworks for run-time type analysis to type functionals or higher-order types. Many type-directed operations, such as mapping functions, equality functions, pretty printers, and debuggers must be defined in terms of higher-order rather than first-order types. Furthermore, we may also extend existing operations to more complicated types (such as polymorphic types) with higher-order analysis. The important part of this work is viewing type analysis as an interpretation of the type language with the term language.

Applications

I have had the opportunity to apply the above theoretical results and frameworks to some practical problems within the framework of the Typed Assembly Language (TAL) project at Cornell.

The TAL project is an instance of *proof-carrying code*, or the certification and static verification of program executables. TAL extends traditional untyped assembly languages with typing annotations, memory-management primitives, and a sound set of typing rules. These typing rules guarantee the memory safety (the program does not access unallocated memory) and control-flow safety (the program only branches to locations containing valid machine instructions) of TAL programs. Moreover, the typing constructs are expressive enough to encode most source-language programming features and TAL is flexible enough to admit many low-level compiler optimizations. Consequently, TAL may be used as a target platform for type-directed compilers that want to produce verifiably safe code for use in secure mobile-code applications or extensible operating-system kernels. Researchers at Cornell have implemented a variant of TAL for Intel's IA32 architecture called TALx86 and have written a compiler for a safe C-like language called Popcorn to TALx86 [6]. Within the TAL project, I have made two important contributions:

Resource Bound Certification While proof-carrying code systems may automatically certify a large set of important security properties, they did not guarantee bounded termination. These verifiable running-time bounds (and, more generally, of resource-consumption bounds) are essential to many applications, such as active networks and extensible operating-system kernels. To obtain such bounds on resource consumption, code consumers have generally had to rely on operating-system monitoring, which can be costly and which works against the direct access afforded by language-based mechanisms.

With Karl Crary, I extended the Popcorn and TALx86 languages with support for the static verification of execution bounds [2]. This extension involved incorporating time-bound annotations into the source language Popcorn and the target language TALx86, and extending the Popcorn compiler to transform the timing annotations during compilation. These time bounds are represented as a limit on the number of backward jumps in the executable (in TALx86) and are expressed with a very sophisticated type system, simulating dependent types. As a result, we allow the execution time of programs and their subroutines to vary, depending on the values

of their arguments. This type system to track execution time bounds (and bounds on other resources) is derived from the LX language [1] described above: its power comes from the ability to flexibly express running time dependency, just as LX may express run-time type dependency.

Type-safe dynamic linking Dynamic extensibility—the ability to augment a running system with new code without shutting the system down—is a principal requirement in many modern software systems. With Mike Hicks and Karl Crary, I also extended Popcorn and TALx86 with support for dynamic linking and loading, creating the first complete framework for flexible and safe dynamic linking of native code [5]. As the security of the resulting system was important to our design, we did not want to unduly expand the *trusted computing base*. Therefore, at the TAL level, the extension to support dynamic linking was minimal; we added a single new instruction for loading and typechecking code, largely composed of existing functionality. This primitive is flexible enough to support a variety of linking strategies. Using this primitive, along with machinery supporting dynamic types [4], we may implement many existing dynamic-linking approaches. As a concrete demonstration, we used our framework to implement dynamic linking for Popcorn, closely modeled after the standard linking facility for Unix C programs. Aside from the unavoidable cost of verification, our implementation performs comparably with the standard, untyped approach.

Future Research

In the future, I intend both to continue my study of the foundations of typed programming languages and to apply those results broadly to existing and emerging practical problems. Currently, I am interested in the following application areas.

Structural type analysis in practice I would like to incorporate my research on dynamic type analysis into an existing object-oriented programming language, such as Java. While Java uses the names of classes for dynamic type dispatch, my extension would allow the examination of the structure of the class as well. This would provide a principled basis for reflection, and would allow polytypic operations, such as data-structure traversals, object cloning, and structural equality, to be expressed more concisely.

Type-based program verification I have already made contributions in the area of using expressive type systems to specify and verify properties of programs with my work on resource bound certification. A limiting factor in this line of research is flexibility in the security-policy specification. Currently, the security policy is contained and implied by the specific type system used to typecheck the program. In order to make this sort of verification feasible we must separate the policy from the type system of the language. Another line of research that must be considered is the trade-off between user annotation of types and automatic type-inference. How much extra information will users be willing to add to their code? Yet the more sophisticated we make the type-inference engine (which is in essence an automated theorem prover), the less they will understand the reasons why type inference fails to verify their program.

Extension frameworks for statically-typed languages The proliferation of domain specific languages has reinforced the idea that there is no perfect language suited for every task. At the same time, programmers are (rightly so) becoming more dependent on sophisticated development environments, debuggers, and static checkers to aid their development process. Supporting these new facilities for every new “little language” is quite impossible, so some untyped or dynamically typed languages have included support (in the form of a macro system) for extension. However, the challenges of extending a statically-typed language with new type constructs as well as verifying that new term forms always produce well-typed programs have previously prevented the development of similar extension mechanisms.

References

[1] Karl Crary and Stephanie Weirich. Flexible Type Analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[2] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, January 2000.

[3] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional Polymorphism in Type Erasure Semantics. In *Journal of Functional Programming*. To appear.

[4] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional Polymorphism in Type Erasure Semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 301–313, Baltimore, MD, September 1998.

[5] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and Flexible Dynamic Linking of Native Code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001.

[6] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In *Second ACMSIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999. Published as INRIA research report number 0228, March 1999.

[7] Stephanie Weirich. Higher-Order Intensional Type Analysis. In Daniel Le Métayer, editor, *Programming Languages and Systems: 11th European Symposium on Programming, ESOP 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002*. To appear.

[8] Stephanie Weirich. Encoding Intensional Type Analysis. In D. Sands, editor, *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2001.