

# Stephanie Weirich

*Research Statement*

---

## Motivation

The goal of my research is to enhance the reliability, maintainability, and security of software systems through programming language technology. The design of a programming language strongly determines the properties of programs written in that language. A good language will promote the design of trustworthy and robust software and discourage the creation of insecure and inflexible code.

In pursuit of this goal, my research focuses on *statically-typed programming languages*. These languages provide specific compile-time guarantees about the absence of certain program errors. As a result, software written in such languages are immune to particular forms of attack. For example, systems written with type-safe languages cannot be compromised by buffer overruns if all array accesses are statically proven safe. In general, the value of a static type system rests in its ability to express the invariants and consistency properties of software systems.

However, the static type systems of modern programming languages are limited in their expressiveness. Some programming idioms must be ruled out simply because they cannot be shown to be sound by a particular type system. My research explores novel methods to bring more expressive types systems to users: enhancing the capabilities of existing type systems to reason about run-time structure and behavior, reconciling the features of expressive-but-theoretical type systems with existing languages, partially automating the process of type system design so that we may be more confident in the soundness of more complicated type systems, and incorporating programming logics into the design of practical type systems so that application-specific properties may be expressed. Below, I describe this research in more detail as well as my future directions.

## Type-directed programming

One focus of my research is programming language support for *type-directed programming* (TDP), or run-time type analysis. With this form of reflection, a program can analyze type information to determine its behavior. By analyzing the type structure of data, many frequently used operations can be defined once, for all types of data. Not only are these operations easier to express with type-directed programming than with more conventional programming paradigms, but, as software evolves, these operations need not be updated—they will automatically adapt to new data forms. Otherwise, each of these operations must be individually redefined for each type of data, forcing programmers to revisit the same program logic many times during a program's lifetime. This flexibility is even more important in the context of adaptive systems that must dynamically react to changes in their environments.

My research efforts in this area span the interaction between type analysis and the features of several different languages (ML, Haskell, Java) and also include results about the contention between type analysis and type abstraction. This project has been funded by an NSF CAREER award.

**AspectML** Aspect-oriented programming (AOP) is a new mechanism for program modularity that allows the concise and atomic expression of cross-cutting concerns. For example, the code to implement auditing for a particular application may be scattered throughout its source code—AOP allows this feature to be declaratively specified in one location. Security concerns are of particular relevance to AOP because they are pervasive, yet must be consistently applied to be effective.

When aspects are combined with languages that contain parametric polymorphism (such as functional languages like ML and Java, and recent extensions to Java and C#), run-time type analysis is essential for effective programming. In collaboration with professor David Walker and graduate student Dan Dantas at Princeton University, and Penn graduate student Geoff Washburn, I have designed a concise language that cleanly separates the mechanisms for aspect-oriented programming from those of type analysis. We have implemented a prototype of this language as an extension of the ML programming language [DWWW05] including extending the type inference algorithm of ML with support for aspects and type analysis, studied

its use for implementing security protocols [DWWW], and shown that AOP provides a concise solution to the problem of specializing type-directed operations to new datatypes [WW06].

**TDP in Haskell** I have also been working on support for type-directed programming in the functional language Haskell. The Haskell language has a particularly advanced type system, with many features not found in other languages, in particular, type classes, type-level programming (currently supported with functional dependencies), template meta-programming and higher-order polymorphism. However, even though Haskell is a research language incorporating the latest advances, it is also a strong platform for development. There is an existing standard for the language, several industrial-strength implementations, a vibrant user-community, and it has been used in numerous industrial and defense applications. My approach for type-directed programming in Haskell has been different than that for Java and AspectML. In those settings, I sought to support TDP through new, special-purpose language features. However, because Haskell's type system is already so expressive, certain forms of TDP may be expressed as a Haskell library.

My approach to TDP in Haskell is based on encodings of *representation types* in the Haskell type system. These types may be encoded in a number of ways: in my Type-Safe Cast paper [Wei04] I gave an example of a type-directed operation translated from run-time type analysis to Haskell type classes. Dispatch on type information was implemented via the appropriate dictionary. Representation types may also be implemented with a variant of a Church encoding [Wei01]. This encoding can be extended to include the representations of higher-order types, and implement in Haskell [Wei06b]. Both of the above encodings are theoretically interesting, but have practical limitations. However, using the mechanism of Generalized Algebraic Datatypes (GADTs) [PVWW06], I implemented a practical library, called RepLib [Wei06a], for TDP in Haskell. This library is available for download<sup>1</sup>. Based on my experience with this design, I and several other researchers are working on a Common Library for Generic programming for the Haskell language <sup>2</sup>.

**TDP in Java** Although object-oriented languages are widely used in the implementation of modern software systems, they have only primitive support for type-directed programming. As a result, ad hoc measures have been incorporated to provide this facility. For example, Java Reflection provides these facilities in a computationally expensive, error-prone manner. To address this deficiency, I and student Liang Huang designed an extension of a core Java calculus with operations for analyzing both the names and structure of Java types [WH04]. This work was based on a core calculus that I and my students Geoffrey Washburn and Dimitrios Vytiniotis developed for the run-time analysis of generative types [VWW05].

**Type analysis and type abstraction** One problem for both aspect-oriented programming and type-directed programming is that these new programming idioms break existing mechanisms for abstraction and separation of concerns. Because of this, several researchers think that these mechanisms decrease modularity. This need not be so, and I have been examining the interaction between type abstraction and modularity in two different ways.

First, Geoffrey Washburn and I have designed a type system that statically tracks the dependence on run-time type information and can therefore describe when abstraction properties do and do not hold [WW05]. Geoffrey is using this model as the basis of his dissertation work: an implementation of a variant of this language and an exploration of the modularity idioms that it enables. Secondly, an alternate mechanism for type-directed programming is based on representation types [CWM02]—instead of analyzing run-time type information, one may analyze terms that represent that type information. In such languages, it was conjectured that type abstraction properties still hold—if the representation of a an unknown type is not provided, values of that type must be used abstractly. Dimitrios Vytiniotis and I formally showed that conjecture to be true [VW07b] and used an extension of that result to show the partial correctness of a type-analyzing cast function [VW07c].

## Type inference

The type systems of the languages of theoretical study, such as  $F_\omega$  and the Calculus of Inductive Constructions, include expressive features that are not found in advanced functional programming languages, such as ML and Haskell. These advanced languages use aggressive type inference to aid program development: users

---

<sup>1</sup><http://www.cis.upenn.edu/~sweirich/RepLib/>

<sup>2</sup>[http://www.haskell.org/haskellwiki/Libraries\\_and\\_tools/Generic\\_programming](http://www.haskell.org/haskellwiki/Libraries_and_tools/Generic_programming)

need not annotate the types of function definitions, nor explicitly write down the types for polymorphic function instantiation. Importantly, type inference encourages the development of generic code, as polymorphic functions are no more difficult to write or use than non-polymorphic ones.

With Simon Peyton Jones at Microsoft Research, my students Dimitrios Vytiniotis and I extended the leading Haskell compiler, GHC, with three new features. *Higher-rank polymorphism* allows functions to take polymorphic functions as arguments [PVWW06]. *Impredicative polymorphism* makes polymorphic functions truly first-class by also allowing them to be stored in polymorphic data structures [PVWW06]. *Generalized Algebraic Datatypes* (GADTs) add dependency between the data constructors of an algebraic datatype and its type indices. Each of these features are available in the current release of GHC and are being used.

The difficulty with all three of these extensions is backwards compatibility. Complete inference for these features is known to be impossible, so these extensions rely on user annotations. The difficult part was determining a complete specification of where annotations are required that did not require any new annotations for programs that do not use the new features while still assigning a best or “principal” type to each code fragment.

## Mechanizing programming language metatheory

Because program security guarantees are often based on static type checking, confidence in the soundness of static type systems is essential. Yet, as static type systems become more and more expressive, such meta-theoretic results become more complex. Typically proofs about the properties of programming languages are done by hand, despite the length and complexity of these results for modern languages. These proofs are not difficult—they use standard, well-understood techniques—but they are often overwhelming in the details. Researchers in programming languages have long felt the need for tools to help formalize and check their work. There are a number of automated proof assistants being developed within the theorem proving community that seem ready to be applied in this domain—yet, despite numerous individual efforts in this direction, the use of proof assistants in programming language research is still not commonplace.

Therefore, I have been working to make the use of automated proof assistants more common place in the formalization of programming language metatheory. This project has two components: developing community infrastructure (education materials, workshops, libraries, electronic fora) to get researchers to use existing tools, and developing new technologies for programming language representation, specifically, the treatment of binding constructs, to make this process easier. I have been working on this project in collaboration with Benjamin Pierce and Steve Zdancewic here at the University of Pennsylvania as well as Peter Sewell at Cambridge University and Randy Pollack at the University of Edinburgh and a number of Penn students. This project is supported by the NSF CISE Computing Research Infrastructure program.

**Infrastructure** We believe that a modest investment, now, in consolidation, documentation, tool creation, and community building will create the conditions for a dramatic change in the way research is done in programming languages. To that end we have undertaken a number of activities to both assess and advance the current best practices in machine-assisted support for the formalization of programming languages. For the former, we have issued a challenge problem [ABF<sup>+</sup>05] that has received considerable interest, developed online resources for community discussion<sup>3</sup> and have organized a number of gatherings: an informal workshop at POPL 2006 to bring together interested parties. Based on this initial success, I organized and chaired 2006 Workshop on Mechanizing Metatheory, which was held in conjunction with ICFP, and attended by over fifty participants. The second iteration of this Workshop will be held in October 2007, also co-located with ICFP. Finally, at POPL 2008 next January, we will give a tutorial on the use of the Coq proof assistant.

**Research** Our experience with this infrastructure project has also influenced my research. In particular, I have long been interested in the treatment of binding constructs in the implementation and formalization of programming languages [WW07]. To push the state of the art, I and students Brian Aydemir and Aaron Bohannon published work describing an encoding of nominal logic in the theorem prover Coq [ABW06]. Brian, Arthur Charguéraud, Benjamin Pierce, Randy Pollack and I also described our assessment of the concrete approaches to binding [ACP<sup>+</sup>07]. Finally, proof assistants have become an integral part of my

<sup>3</sup><http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/>

research: using this technology, my student Dimitrios Vytiniotis and I are formalizing our current work. All of the results in our most recent paper were formalized in the Isabelle/HOL proof assistant [VW07b].

## Future directions

**Dependent type systems** Static type systems are a popular, cost-effective form of lightweight program verification. They provide a tractable and modular way for programmers to express properties that can be mechanically checked by the compiler, thereby ruling out a wide variety of common memory manipulation and control-flow errors. While extremely helpful for building robust software, the type systems used in practice verify only relatively weak safety properties; they fall far short of what is needed to do full verification of program correctness. This inexpressiveness is partly by design—full program verification is potentially expensive, not fully automatable, and probably unwarranted for non-safety-critical software. Nevertheless, there are many situations in which the ability to specify program properties richer than current type systems permit would be useful. In fact, there is a spectrum of possibilities between simple type safety and full program verification.

Dependent type systems extend the type systems of current practice to allow incremental, lightweight specifications. They work by allowing types that are used as specifications to depend on values computed by program expressions. However, even though the theory behind dependent type systems has been refined over several decades, such type systems have been little used in practical programming languages. The goal of this project is to make these systems work. Specifically, my students Dimitrios Vytiniotis, Brian Aydemir and I are designing a dependently-typed programming language with the following features: a rigorous, type-based distinction between pure and effectful code that allows dependency in the presence of non-logical linguistic features, a flexible, expressive mechanism for predictable type-and-proof inference that reduces the annotation burden of fully specified code, and a system of automatic interface generation between checked and unchecked modules that encourages incremental verification.

This project has been funded by the NSF Foundations of Computing Processes and Architectures and is in its initial stages. Currently, we have a prototype design and implementation [VW07a].

**Manifest Security** This project proposes manifest security as a new architectural principle for secure extensible systems. Its research objectives are to develop the theoretical foundations for manifestly secure software and to demonstrate its feasibility in practice. This project is a collaboration with Robert Harper, Frank Pfenning and Karl Cray at Carnegie Mellon University and Benjamin Pierce and Steve Zdancewic at Penn. It has been funded by the NSF CyberTrust program for the next three years [CHP<sup>+</sup>07].

Manifest security addresses two fundamental problems in extensible software platforms—software systems that can be customized by installing third-party extensions. Such extensions often require access to system resources and sensitive information for their operation, yet unrestricted access compromises the security of the system. It is not clear how to properly *specify* policies about what resources and information an extension may use, nor do we know the best way to effectively *enforce* such policies. Manifest security therefore introduces a novel high-level logical specification language for users to specify the policies that govern any extensions they apply, and uses a combination of static and dynamic methods to determine whether these extensions actually meet these policies.

## References

- [ABF<sup>+</sup>05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *The 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, Oxford, UK, August 2005.
- [ABW06] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, pages 60–69, Seattle, WA, USA, August 2006.
- [ACP<sup>+</sup>07] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. Draft, July 2007.
- [CHP<sup>+</sup>07] Karl Crary, Robert Harper, Frank Pfenning, Benjamin C. Pierce, Stephanie Weirich, and Stephan Zdancewic. Manifest security. Technical report, January 2007. White paper.
- [CWM02] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.
- [DWWW] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. To appear in *Transactions on Programming Languages and Systems*. 59 pages.
- [DWWW05] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 306–319, Tallinn, Estonia, September 2005.
- [PVWW06] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, pages 50–61, Portland, OR, USA, September 2006.
- [VW07a] Dimitrios Vytiniotis and Stephanie Weirich. Dependent types: Easy as PIE. In Marco T. Morazán and Henrik Nilsson, editors, *Draft Proceedings of the 8th Symposium on Trends in Functional Programming*, pages XVII–1—XVII–15. Dept. of Math and Computer Science, Seton Hall University, April 2007. TR-SHU-CS-2007-04-1.
- [VW07b] Dimitrios Vytiniotis and Stephanie Weirich. Free theorems and runtime type representations. In *Mathematical Foundations of Programming Semantics (MFPS XXIII)*, pages 357–373, New Orleans, LA, USA, April 2007.
- [VW07c] Dimitrios Vytiniotis and Stephanie Weirich. Type-safe cast does no harm. Draft, July 2007.
- [VWW05] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 13–24, Long Beach, CA, USA, January 2005.
- [Wei01] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming (ESOP)*, pages 92–106, Genova, Italy, April 2001.
- [Wei04] Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.
- [Wei06a] Stephanie Weirich. Replib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, September 2006.
- [Wei06b] Stephanie Weirich. Type-safe run-time polytypic programming. *Journal of Functional Programming*, 16(10):681–710, November 2006.
- [WH04] Stephanie Weirich and Liang Huang. A design for type-directed Java. In Viviana Bono, editor, *Workshop on Object-Oriented Developments (WOOD)*, ENTCS, pages 117–136, 2004.
- [WW05] Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information flow. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 62–71, Chicago, IL, USA, June 2005.
- [WW06] Geoffrey Washburn and Stephanie Weirich. Good advice for type-directed programming: Aspect-oriented programming and extensible generic functions. In *Workshop on Generic Programming (WGP)*, pages 33–44, Portland, OR, USA, September 2006.
- [WW07] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 17(6), November 2007. To appear. 56 pages.