Stephanie Weirich

Research Statement

My research focuses on the design of **statically-typed programming languages**. 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. As a result, the compiler can rule out a wide variety of errors and provide more information to refactoring and development tools. For example, systems written with type-safe languages cannot be compromised by buffer overruns if all array accesses are statically proven safe. Furthermore, programmers can modify their code with the assurance that they have not violated critical safety properties.

I explore these designs in the context of **functional programming languages**, such as Haskell and ML. Functional programming languages are an ideal context for type system research; they excel in their capabilities for static reasoning. However, there is need for improvement. Some programming idioms must be ruled out simply because they cannot be shown to be sound by existing type systems. To overcome these limitations, my work investigates type system features in the context of both new languages and existing ones, and evaluates those designs with respect to both theory and practice.

Trellys: Dependently-typed language design

Dependent types promise to dramatically increase the effectiveness of static type systems. They work by allowing types to depend on program values, enabling specifications that are both more flexible and more precise. However, even though dependent type theory has been well studied as a foundation for logical reasoning, these type systems have been little used in practical programming languages.

Together with Aaron Stump (University of Iowa), and Tim Sheard (Portland State), I lead the recently-completed **Trellys project**, which investigated the design and implementation of novel programming languages with dependent types. This Trellys project is unique in that it explores the addition of full-spectrum dependent types to a typical functional programming language (i.e. a call-by-value language with general recursion). Prior work made restrictions on the sorts of dependency or computation allowed.

At Penn, our work has resulted in the development of the **Zombie** language.[1] This language design makes two significant advances:

- **Combining proofs and programs** [CSW14]. The Zombie core language is a type theory that provides a *uniform framework* for programs (which could potentially run forever) and proofs (which must terminate to guarantee logical consistency). These two systems work together, allowing proofs to reason about programs and programs to dynamically manipulate proofs. The type soundness and logical consistency for this core language required the development of a novel proof technique [CSW12], which was applied to the core language using the Coq proof assistant.

- **Programming up-to-congruence** [SW14]. The Zombie source language uses a novel adaption of the *congruence closure* algorithm for proof and type inference. Our implementation is able to automatically construct equational reasoning proofs from available evidence. A benefit of this approach is that we are able to dramatically simplify the semantics of dependently-typed pattern matching without sacrificing expressiveness. Our work includes a specification of the source language type system "up-to-congruence", an algorithm for elaborating expressions to the explicitly type core language, and a proof that elaboration is complete and always produces well-typed terms.

The design of the Zombie language draws from a number of our earlier results with respect to the treatment of equality in call-by-value dependently-typed languages [JZSW10], the interaction between irrelevant computation and equality [SCA+12], and the use of type systems to track termination [SSW10].

[1] Available from `https://code.google.com/p/trellys/`

This work has been recognized through invitations for **keynote addresses** (Programming Languages Mentoring Workshop (PLMW 2014), Federated Conference on Rewriting, Deduction and Programming (RDP 2011, which includes TLCA and RTA), Mathematical Foundations of Program Semantics Conference (MFPS 2011)), **tutorial lectures** at the Oregon Programming Languages Summer School (2014 and 2013), and **invited discussion sessions** at the Programming Languages meets Program Verification Workshop (PLPV 2011 and 2010).

I have also worked to build the community of researchers that study the interaction between dependent types and functional programming. In particular, I organized and chaired two meetings: the DTP 2103 workshop co-located with ICFP, and (with Conor McBride (Strathclyde University, Scotland) and Shin-Cheng Mu (Academia Sinica, Taiwan)) Shonan Seminar 007, in Shonan Village Japan in 2011.

Haskell: Expressive types in practice

Haskell is an advanced purely-functional programming language, designed by a committee of programming language researchers in the early nineties. Since that time, it has served as a testbed for novel language features. I have been an active part of this research community for over ten years. In particular, I have worked with and continue to collaborate with Simon Peyton Jones at Microsoft Research, my students Dimitrios Vytiniotis, Richard Eisenberg, Brent Yorgey and Geoffrey Washburn, and many others, on extensions to the type system of the industrial strength Glasgow Haskell Compiler (GHC).[2]

My work extends the Haskell type system in three directions:

- **Dependently-typed Haskell.** The goal of my most influential extensions has been to make Haskell more like a dependently-typed language.

  My work in this area started with the addition of *Generalized Algebraic Datatypes* (GADTs) to GHC [PWW04, PVWW06]. This feature gives programmers the ability to constrain data structures by invariants encoded in its type system. For example, GADTs can be used to show that an implementation of 2-3 trees stores elements in sorted order and maintains a consistent height of subtrees. GADTs have been a remarkably popular addition to GHC, seeing use in generic programming, modeling objects, and embedding constraints in embedded domain specific languages. Although we did not invent the idea of GADTs (they were independently introduced several times during 2003-2004, and partly inspired by my doctoral research [CW99, Wei00, Wei02, Wei04]), our work was the first to integrate them with an industrial-strength language and coined their now-standard name.

  Since that time, I have worked to extend the expressiveness of type-level computation, the language feature of GHC used to encode program invariants. *Datatype promotion* [YWC$^+$12], makes standard data structures available at the type language, and *Closed type families* [EVPW14] allows the definition of type-level functions over those data structures. I have also shown how to extend Haskell's core language with *kind equalities* [WHE13], which introduces dependent types to type-level computation.

- **Roles.** Sometimes type system features interact poorly. In particular, GADTs, type-level computation and a feature of GHC called `GeneralizedNewtypeDeriving` naïvely combine to make the type system unsound. My work on roles [WVPZ11, BEPW14] resolved this incompatibility and plugged a six-year hole in the GHC type checker.

- **First-class polymorphism.** The Haskell type system was originally based on Hindley-Milner type system, which places restrictions on the use of polymorphic functions in exchange for complete type inference. Our work removes those restrictions, permitting the use of *higher-rank* [PVWS07] and *impredicative polymorphism* [Vyt08, VWP06, VWJ08]. Complete inference for these features is known to be impossible. Therefore, our work defined a specification of required user annotations to enable the new features. Our extensions are backwards compatible: they do 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.

[2] `http://www.haskell.org/ghc/`

My students and I have released several open source libraries that take advantage of these extensions. The `singletons` library [EW12, ES14] directly supports programming with compile-time invariants; for example, it powers the `units` library [ME14], which ensures that units are used consistently in physical simulations. The `RepLib` library [Wei06] enables datatype-generic programming, a mechanism for defining functionality based on type structure. I used this library implement `Unbound`, a library for working with binding structure [WYS11].

This work has been recognized by the academic community through invitations for **keynote addresses** at the International Conference on Functional Programming (ICFP 2014), the 2013 Facebook Faculty Summit, and the Symposium on Functional and Logic Programming (FLOPS 2012). My visibility in the functional programming community is also recognized by my service as the **program chair** of ICFP 2010, the **program chair** of the Haskell Symposium 2009, and as an **editor** of the Journal of Functional Programming, the primary publication venues for research in this field.

Concurrent with my work, the Haskell language has made tremendous growth in industry, arguably due to its uniquely expressive type system. The features described above are implemented in GHC and are being used by nonacademic Haskell practitioners. In particular, I have been in contact with full-time programmers who use GHC for banking, social networking, robotics, aerospace engineering, and consulting. Each year, my students organize the the Hack-φ Hackathon, which brings local programmers together to build and improve Haskell libraries, tools, and infrastructure

I plan to continue my efforts to extend the Haskell type system. In particular, we are currently working to add a true dependent type to GHC, in order to ameliorate the syntactic overhead of singleton-based encodings. I have also been working with an undergraduate research student (Hamidhasan Ahmed, Penn) to add explicit type application. Going further, type inference in GHC is based on constraint solving and implemented by a special purpose constraint solver. To extend the static reasoning capabilities of GHC, I have a NSF grant to explore the integration of SMT (Satisfiability Modulo Theory) solvers into its type inference algorithm.

POPLmark: 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 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.

Therefore, I have been working to make the use of automated proof assistants more commonplace in the formalization of programming language metatheory. Such proof assistants manage the complexity of these proofs; including mechanisms for semi-automatic proof creation, checking and maintenance. This project is a collaboration with Benjamin Pierce and Steve Zdancewic at the University of Pennsylvania as well as Peter Sewell at Cambridge University, Randy Pollack at the University of Edinburgh, and a number of Penn students.

This work 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.

- **Community infrastructure**. In 2005, the POPLmark team issued the POPLmark challenge: a set of design problems to both assess and advance the current best practices in machine-assisted support for the formalization of programming languages [ABF$^+$05]. This challenge was met with enthusiasm, generated many solutions that we gathered on our website and spurred vigorous debate in many venues.

  To encourage the use of proof assistants in the programming language community, I **organized and chaired** the 2006 Workshop on Mechanizing Metatheory. This workshop continued to meet every year during 2007-2010. Furthermore, the Penn group organized a **tutorial** on using Coq for programming

language metatheory at POPL 2008 and I taught this material at the Oregon Programming Languages Summer School in 2008.

- **Binding representations and tools**. Our work also included techniques and tools for working with binding representations. In particular, we developed a new technique for inductive relations over a *locally nameless representation* of syntax [ACP+08]. My student Brian Aydemir developed the LNgen tool [AW10] to aid in the support of locally nameless representation. More recently, Ghent University PhD student Steven Keuchel developed the InfraGen tool to support working with de Bruijn representations of binding structures during a recent six-month visit to Penn [KWS14].

This joint project has had tremendous impact on the community. It is not uncommon for papers submitted to top programming languages conferences, such as POPL and ICFP, to be accompanied by mechanical proofs of the correctness of their results. Benjamin Pierce and I **co-edited a journal special issue** devoted to descriptions of solutions to the POPLmark challenge. Furthermore, I was invited to give a retrospective talk at Cambridge University (Dec 2009) and a **keynote address** at the LFMTP 2012 workshop.

Other projects

HACMS DARPA's High-Assurance Cyber Military Systems (HACMS) project seeks to improve the security of autonomous and semi-autonomous vehicles such as helicopters, automobiles and unmanned ground vehicles. As part of this project, I am part of a team of researchers at Penn and UCLA that are designing attack-resilient control algorithms and building modeling systems to support simulation and high-assurance code generation. In particular, we have used the Coq proof assistant to develop a glue code generator for the ROS platform, and have used the program logic of the Verified Software Toolchain to prove the correctness of this generator [MPS+14].

Type-directed programming and abstraction With type-directed programming, 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 [DWWW08], Haskell [Wei06], Java [WH04]) and also include results about the contention between type analysis and type abstraction. In particular, type-directed 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 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]. 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 [VW07] and used an extension of that result to show the partial correctness of a type-analyzing cast function [VW10].

References

[ABF+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.

[ACP+08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15, January 2008.

[AW10] Brian Aydemir and Stephanie Weirich. Lngen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, June 2010.

[BEPW14] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for Haskell. In *The 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, September 2014. To appear.

[CSW12] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Step-indexed normalization for a language with general recursion. In *Fourth workshop on Mathematically Structured Functional Programming (MSFP '12)*, pages 25–39, 2012.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL 2014: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 33–45, San Diego, CA, USA, 2014.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 233–248, Paris, France, September 1999.

[CWM02] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[DWWW08] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages*, 30(3):1–60, May 2008.

[ES14] Richard A. Eisenberg and Jan Stolarek. Promoting functions to type families in Haskell. In *Haskell Symposium*, 2014. To appear.

[EVPW14] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *POPL 2014: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 671–683, San Diego, CA, USA, January 2014.

[EW12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell Symposium*, pages 117–130, Copenhagen, Denmark, September 2012.

[JZSW10] Limin Jia, Jianzhou Zhao, Vilhem Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 275–286, Madrid, Spain, January 2010.

[KWS14] Steven Keuchel, Stephanie Weirich, and Thomas Tom Schrijvers. Infragen: Binder boilerplate at scale, July 2014. Submitted for publication.

[ME14] Takayuki Muranushi and Richard A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *Haskell Symposium*, 2014. To appear.

[MPS+14] Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee. Verified generation of glue code for ros-based control systems. Submitted for publication., 2014.

[PVWS07] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, January 2007.

[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.

[PWW04] Simon L. Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Practical type inference for generalised algebraic dataypes. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[SCA+12] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogenous equality, and call-by-value dependent type systems. In *Fourth workshop on Mathematically Structured Functional Programming (MSFP '12)*, pages 112–162, 2012.

[SSW10] Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion. In *Workshop on Partiality and Recursion in Interactive Theorem Provers*, pages 76–93, Edinburgh, Scotland, July 2010.

[SW14] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. Submitted for publication, July 2014.

[VW07] 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.

[VW10] Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, type equality and higher-order polymorphism. *Journal of Functional Programming*, 20(2):175–210, March 2010.

[VWJ08] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: first-class polymorphism for Haskell. In *ICFP 2008: The 13th ACM SIGPLAN International Conference on Functional Programming*, pages 295–306, Victoria, BC, Canada, September 2008.

[VWP06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. Boxy type inference for higher-rank types and impredicativity. In *International Conference on Functional Programming (ICFP)*, pages 251–262, Portland, OR, USA, September 2006.

[Vyt08] Dimitrios Vytiniotis. *Practical type inference for first-class polymorphism.* PhD thesis, University of Pennsylvania, August 2008.

[Wei00] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 58–67, Montreal, Canada, September 2000.

[Wei02] Stephanie Weirich. *Programming With Types.* PhD thesis, Cornell University, August 2002.

[Wei04] Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.

[Wei06] Stephanie Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, September 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, August 2004.

[WHE13] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *Proceedings of The 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 275–286, Boston, MA, September 2013.

[WVPZ11] Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *POPL 11: 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, January 26–28, 2011. Austin, TX, USA.*, pages 227–240, January 2011.

[WW05] Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information flow. In *Twentieth Annual IEEE Symposium on. Logic in Computer Science (LICS 2005)*, pages 62–71, Chicago, IL, USA, June 2005.

[WYS11] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 333–345, New York, NY, USA, 2011.

[YWC+12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhaẽs. Giving Haskell a promotion. In *Seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*, pages 53–66, 2012.