David Van Horn – Personal Statement

Biography. I am an Assistant Professor in the Department of Computer Science and UMIACS, which I joined in December, 2013. I received my PhD from Brandeis University in 2009. Before UMD, I was a CRA Computing Innovation Fellow and an Assistant Research Professor at Northeastern University.

Executive Summary

Research. I work toward making the construction of reusable, trusted software components possible and effective using programming language (PL) techniques. My research has spanned program analysis; semantics; verification and model-checking; security; logic; complexity; and algorithms.

I have published 27 peer-reviewed conference or journal papers. This includes one paper in the *Communications of the ACM: Research Highlights* and multiple papers in all four of the flagship SIGPLAN conferences: ICFP (8), OOPSLA (4), POPL (2), and PLDI (2). My ICFP papers have been selected for special issues of the *Journal of Functional Programming* five times. My work on "abstracting abstract machines" has featured prominently in nine PhD theses to date and has been the subject of several invited lectures, research summer schools, and conference tutorials.

Funding. While at Maryland, I have secured over $2M in funding from the National Science Foundation and the Department of Defense to support my research, with $948K as lead investigator.

Teaching and mentoring. While at UMD, I have graduated one PhD student with a second expected this year. The graduated student is now tenure-track faculty at the University of Vermont. I have advised or co-advised five post-docs; four now hold tenure-track faculty positions and one is in an industrial research lab. Each of my papers published since joining UMD features a current or former UMD graduate student or post-doc as lead author; several feature UMD students, including undergraduates, as secondary authors.

I have engaged in significant external mentoring and diversity efforts, including speaking at multiple PL Mentoring Workshops and lectured at the Oregon PL Summer School and the PLT Redex Summer School. I have chaired the ACM Student Research Competition and served on its selection committee several times.

I have taught multiple courses at the freshmen, senior, and PhD levels and have undertaken a complete redesign of the department's first-year programming sequence, CMSC 131 and 132. At the undergraduate level, my course evaluations are consistently above department and college averages on almost all metrics.

Service. Externally, I have served as a referee for every major PL journal; as a program committee member for ICFP (2), POPL, OOPSLA, ESOP, ECOOP, as well as several smaller venues; and as an external review committee member for ICFP and POPL (2). I have served a three-year term on the ICFP Steering Committee. I chaired or co-chaired the 2016 Symposium on Trends in Functional Programming, the 2014 Workshop on Higher-Order Program Analysis, the 2011 NII Workshop on Automated Techniques for Higher-Order Program Verification, and the 2011 New England Programming Languages and Systems Symposium. I have served on two NSF panels. I am currently chairing the SIGPLAN PL Mentoring Workshop at ICFP.

Internally, I have served on or chaired several department committees. I chaired or co-chaired the graduate student review committee for the past four years. I have served on the education committee and the PL/SE/HCI field committee every year since joining UMD. I served on the graduate admission committee for two years. I have served on the space committee (for the Iribe Center) for the past three years.

7/2/18

Research

Since the late 1960s, computer scientists have struggled with what has come to be known as the *software crisis* [33]: an ever increasing reliance of society on computing systems, coupled with the growing gap between the ubiquity and power of these systems and the difficulty of writing useful and efficient programs economically. In his 1972 ACM Turing Award lecture [11], Edsger Dijkstra remarked: "To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem." In the intervening decades, computing systems have radically proliferated, our reliance upon them dramatically deepened, and the "gigantic" computers of Dijkstra's time are infinitesimal compared to today's. In short, the software crisis has flourished and overcoming it is all the more critical.

I believe the solution to this crisis rests in the effective use of programming language (PL) technology, which has the potential to turn the power of computing toward resolving the very crisis it creates. PL history has marched steadily from low-level machine-oriented languages to high-level languages enabling more abstract forms of thinking; computation bridges the gap between them. Today, programming languages and their associated tools can guide good design, categorically eliminate large classes of errors and vulnerabilities, and to provide substantial aid to programmers throughout the software development life-cycle.

I work on the design, implementation, and use of programming languages and program analysis with the goal of making the construction of reusable, trusted software components possible and effective. I focus on modern, high-level languages and automated techniques for analyzing, verifying, and debugging programs.

Computational complexity of program analysis. Program analysis is the art and science of making (software that makes) useful predictions about what a program will do when run. One of the most common forms is *flow analysis* (sometimes called *control-* or *data-flow* analysis, a distinction without a difference in functional and object-oriented languages, which are *higher-order*: they include computational values). Flow analysis predicts what possible values a given expression may take on when run. It is a fundamental form of analysis that essentially underpins any other kind of prediction. Higher-order flow analysis has been widely studied since 1981 [25] with many variants occupying points along a spectrum of precision and performance.

One of the most famous classes of flow analyses is Shivers' kCFA hierarchy, a family of analyses that—for some constant k—distinguishes k-levels of function call contexts before resorting to a coarse-grained approximation. As a PhD student, I was struck by a passage in Shivers' 25-year retrospective on kCFA [39]:"It did not take long to discover that the basic analysis, for any $k > 0$, was intractably slow for large programs. In the ensuing years, researchers have expended a great deal of effort deriving clever ways to tame the cost of the analysis." Despite the extensive literature, very little was known about the computational complexity of performing flow analysis, which would shed light on whether "taming the cost" was even possible.

In my dissertation, I established tight bounds on the complexity of kCFA and related analyses [47]. I proved for any $k > 0$, computing kCFA was complete for EXPTIME [49], demonstrating empirically observed increases in costs can be understood analytically as inherent in the approximation problem being solved. For 0CFA, I proved it PTIME-complete [48], and showed the result was robust for every known variant of 0CFA that made further approximations [50]. I derived a type-based variant of 0CFA for programs adhering to a very restricted syntactic discipline that was in LOGSPACE, which provides some evidence that there's no good 0CFA-like analysis with complexity below PTIME for general programs. While these results appear negative, they yielded insights into better designs. After the EXPTIME-completeness result, we designed an alternative kCFA hierarchy with polynomial-time upper-bounds for any k and empirically showed it computes more efficiently without losing precision compared to Shivers' kCFA [31].

Methodologies for building analyzers. One of the main impediments to building sound software analysis tools is that designing these predictive models traditionally requires highly specialized training, such as a doctorate, not just in PL, but in program analysis specifically. Expressive languages complicate the problem further, limiting its impact on modern high-level languages.

Based on insights gained during my complexity theory investigation, we designed a methodology for developing sound program analyzers using *abstract machines*, a semantic model widely used in the PL research community. The approach, dubbed *abstracting abstract machines* [51], starts from familiar territory—the stuff of undergraduate PL courses—and uses a series of simple program transformations applied to the machine semantics of a language. Each transformation is easily justified as semantics preserving, until a final "finitization" step that incorporates approximation, rendering the models computable. One of the main strengths of AAM is it reduces analysis design, even for sophisticated language features, down to formulating a machine semantics for those features: something the PL community excels at with a large literature on tools and techniques. The original paper includes a series of vignettes applying the technique to analyze stack inspection, garbage collection, laziness, and control operators, each of which would previously be considered a contribution on its own. The AAM technique has thus expanded both the community capable of designing analyzers and the set of language features subject to program analysis.

My research on AAM has been well-received and influential. The original ICFP paper was selected to appear in *Communications of the ACM: Research Highlights* [52], which selects a paper monthly "from all areas of computer science to be highlighted as especially important and relevant for the 80,000+ members of the ACM." It was invited to the special issue of *JFP* devoted to ICFP'10 [53]. It forms the basis of several PhD theses that build upon or employ the approach [1, 5, 12, 17, 19, 26, 27, 38, 42], as well as many papers. It has been the subject of conference tutorials, invited lectures, and research summer schools. The technique has been applied by others to a number of languages such as Scala, Erlang, Java, and JavaScript. It has influenced the design of tools developed at HP Fortify, Github, and Google. I consistently receive feedback praising the work as being one of the most accessible and lucid accounts of how to build an abstract interpreter.

Since conceiving AAM, we have applied and extended it in a number of settings, such as reasoning about concurrency [32], exceptions [30], and detecting malware in Android applications [29, 28]. Thanks to the close correspondence between AAM and the underlying semantics, it is possible to import well-known optimization techniques to speed up analysis [21]. Moreover, the simplicity has enabled strong advances, both in the theory and practice of higher-order program analysis. In particular, we achieved so-called "pushdown" analysis, which essentially replaces a finite-state approximation with that of a pushdown automata [13, 14, 20, 22]. This results in perfectly precise analysis of function calls and returns. Recently, we have shown this added power can be achieved with the same theoretical and observed cost as the finite-state approach, giving a cubic-time algorithm [18]. Finally, we have demonstrated the AAM steps can be applied starting from a high-level compositional interpreter rather than a low-level machine. Remarkably, this tack results in an analyzer that inherits the pushdown property from the defining language rather than through any explicit mechanism [8]. (Two of these paper were invited to special issues of *JFP* [22, 9].)

Verifying behavioral properties of programs. Modern software is developed from reusable components, which communicate in diverse ways. This necessitates well-defined interfaces and mechanisms to discern faulty components when an error occurs. Software contracts [16] express these invariants and agreements between components and ensure they have sensible semantics even in a higher-order setting. Among the subtle issues addressed by contracts is blame assignment, which determines which component is at fault when a contract is violated. Contracts thus form a rich specification language enables a marketplace of reusable software components with a proper account of culpability.

Over the past several years, we have developed techniques for automatically verifying software contracts. The goal has been to leverage contracts to enable a marketplace of *verified* reusable components that are formally proven to satisfy their contracts. Two paramount technical obstacles needed to be overcome to achieve this goal. First, the expressivity of contracts, while crucial for the construction of reliable components, thwarts static reasoning about programs and incurs significant run-time monitoring costs. Second, the expressivity of higher-order languages, a mainstay of modern industrial software, thwarts static reasoning about contracts, despite the availability of mature automated tools and techniques.

To overcome these obstacles, we developed a novel symbolic semantics for modularly executing programs with contracts at its interface boundaries [44, 43]; one of the key contributions was a treatment of higher-order symbolic functions. When combined with existing abstraction techniques, such as AAM, the symbolic semantics becomes an effective automated verification engine for proving the absence of run-time errors, including contract failures [34]. (This paper was accepted to a special issue of *JFP* [36].) Despite the source language's use of higher-order values, the verification technique is able to side-step the need for a higher-order solver, thereby leveraging powerful off-the-shelf SMT solvers. The approach is also useful for generating concrete, potentially higher-order, counterexamples—inputs that witness a run-time failure—for programs, and we proved a strong relative completeness result demonstrating counterexample generation depends only upon the power of a first-order solver for the base types of a language [37]. Recently, the approach has been extended to handle stateful programs effectively [35]. This work is formalized and proved sound with mechanically checked proofs; prototypes were accepted by artifact evaluation committees at PLDI and POPL. The empirical evaluation shows the approach effective in eliminating 99.94% of run-time checks in a suite of realistic programs. Currently, we are developing *termination contracts* as a means to verify total, rather than partial, correctness.

Verified and extensible analyzers. Critical software systems require high-assurance tools to verify the absence of undesirable behavior such as crashes, security vulnerabilities, or privacy lapses. While many of these tools exist, few are verified, calling in to question the trustworthiness of their results, and consequently, the reliability of the critical systems. This situation persists despite several decades of research and investment in independent areas of mechanic verification and sound program analysis. The main problem is each aspect is on its own considered a difficult undertaking, technically and economically.

We have made progress toward a solution in two important regards: we integrated techniques from mechanical verification and program analysis allowing existing correct-by-construction methods for designing analyzers to be carried out in a dependently-typed proof assistant with ability to extract certified implementations; we have developed a theory and mechanism for extensible program analysis construction that enables analyzers to be constructed correctly and automatically out of a combination of existing analysis components.

Abstract interpretation (AI) is a theory of sound approximation widely used in semantics, formal verification, and static analysis. Since its debut in the late 1970s [3, 4], efforts to combine AI and mechanized verification have achieved limited success, either sacrificing generality of the theory or the ability to extract certified analyzers from existing proofs. Our theory of *constructive Galois connections* achieves both [6]. (This paper was accepted to a special issue of *JFP* [7].) The key insight was to use monadic discipline to isolate and navigate between specifications and implementations. We were able to carry out two case studies of deriving certified analyzers in a dependently typed programming language. Monadic transformers were employed in our work on *Galois transformers* to achieve modular and extensible program analyzers that makes it possible to design analysis components that are reusable in their implementation and metatheory [10]. We are currently exploring the use of program synthesis to make building analyzers even easier.

Other work. In addition to the above, I have worked on incremental computation [24], online-verification validation [23], probabilistic languages [46], mechanical theorem proving [54], programming pedagogy [45], gradual refinement types [55], semantics of laziness [2], and temporal model checking [40, 41].

Teaching, Mentoring, and Service

My approach to teaching CS is based on the proposition that writing programs is the most precise form of thinking. As such, everybody should—and can—be taught to program as part of a college education.

I have taught UMD courses at all levels. My graduate course *CMSC 631: Program Analysis & Understanding* introduces the complementary areas of *PL* and *program analysis* and exposes students to the basic principles of research processes in CS. It covers the theory and practice of PL and techniques to mechanically reason about programs and trains, while also training students to articulate questions and recognize elements of solutions.

At the upper undergraduate level, my course *CMSC 430: Introduction to Compilers* introduces students to one of the most powerful and fruitful ideas in computer science, which is that often the best way to solve a problem is to develop a new language that makes the solution easy to express correctly, succinctly, and maintainably. Student develop the skills needed to design and implement their own programming language. Throughout the course, students design and implement several related languages, and will explore parsing, syntax querying, data-flow analysis, compilation to byte-code, type systems, and language inter-operation.

At the introductory level, I designed and taught a two course sequence: *CMSC 131A, 132A: Systematic Program Design I & II*. The first course introduces computing and programming. Its major goal is to teach students principles of systematic problem solving through programming. It exposes students to the fundamental techniques of program design: an approach to the creation of software that relies on systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. It uses interactive, distributed, multi-player games to engage students based on *Realm of Racket*, a book I co-authored with a group of undergraduates [15]. The second course studies class-based program design and abstractions for reusable software and libraries. It covers the principles of object orientation and examines the relationship between algorithms and data structures. Both courses help students develop critical thinking and general problem solving skills. They learn how to structure their ideas and articulate complex concepts to themselves and to machines. This sequence was offered on an experimental basis, working closely with the Associate Chair of Undergraduate Education to evaluate its success.

I have helped place my PhD students and post-docs in tenure-track faculty positions at the U. of Vermont, U. of Alabama at Birmingham, U. of Colorado at Boulder, the Madrid Institute of Advanced Studies, and in research and engineering positions at Microsoft Research and Google.

I have served a three year term on the steering committee for ICFP. I have served on the PC and ERC of 23 conferences and workshops, including three flagship SIGPLAN conferences (POPL (3), ICFP (3), and OOPSLA). I was the program and general chair for the Symposium on Trends in Functional Programming and for the Workshop on Higher-Order Program Analysis. I have co-organized international meetings at NII in Japan and the regional New England Programming Languages Symposium and served on two NSF review panels. I have served on the ACM SIGPLAN Student Research Competition committee at ICFP and PLDI and chaired the competition at ICFP. I have participated in PLMW (and currently chair it), OPLSS, the PLT Redex Summer School, and Student Research Competitions; given guest lectures in *CMSC 396H: Undergraduate Honors Seminar*. For the past four years, I have chaired or co-chaired the committee that organizes our annual assessment of graduate students. I've served multiple times on the committees for graduate admission, Middle States evaluation, education, and the Iribe Center currently under construction.

Conclusion

I enjoy being a professor at Maryland. The environment provides me the inspiration, freedom, and support to pursue creative scholarship and education in pursuit of my goal of making the construction of reusable, trusted software components possible and effective. My students, colleagues, and I have achieved many notable successes and I look forward to continuing to contribute to the computer science community through research, teaching, mentoring, and service, thereby advancing the mission of the university.

References

[1] Peter Aldous. *Noninterference in Expressive Low-Level Languages*. PhD thesis, University of Utah, 2017.

[2] Stephen Chang, David Van Horn, and Matthias Felleisen. Evaluating call by need on the control stack. In *Proceedings of the Symposium on Trends in Functional Programming*, 2010. Best student paper award.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1979.

[5] David Darais. *Mechanizing Abstract Interpretation*. PhD thesis, University of Maryland, 2017.

[6] David Darais and David Van Horn. Constructive Galois connections. In *ICFP '16: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2016.

[7] David Darais and David Van Horn. Constructive Galois connections. *Journal of Functional Programming*, 2018. To appear.

[8] David Darais, Nicholas Labich, Phúc C. Nguyễn, and David Van Horn. Functional pearl: Abstracting definitional interpreters. In *ICFP '17: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, September 2017.

[9] David Darais, Nicholas Labich, Phúc C. Nguyễn, and David Van Horn. Functional pearl: Abstracting definitional interpreters. *Journal of Functional Programming*, 2018. In preparation.

[10] David Darais, Matthew Might, and David Van Horn. Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis. In *OOPSLA '15: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 552–571, New York, NY, USA, 2015. ACM.

[11] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[12] Emanuele D'Osualdo. *Verification of message passing concurrent systems*. PhD thesis, University of Oxford, 2015.

[13] Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *Workshop on Scheme and Functional Programming*, 2010.

[14] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2012.

[15] Matthias Felleisen, Barski M.D. Conrad, David Van Horn, and Eight Students of Northeastern University. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, San Francisco, CA, USA, 2013.

[16] Robert B. Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2002.

[17] Thomas Gilray. *Introspective Polyvariance for Control-Flow Analyses*. PhD thesis, University of Utah, 2017.

[18] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *POPL '16: Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles in Programming Languages*, January 2016.

[19] Dionna A. Glaze. *Automating abstract interpretation of abstract machines*. PhD thesis, Northeastern University, 2015.

[20] Dionna A. Glaze and David Van Horn. Concrete semantics for pushdown analysis: The essence of summarization. In *Workshop on Higher-Order Program Analysis*, June 2013.

[21] Dionna A. Glaze, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In *ICFP '13: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2013.

[22] Dionna A. Glaze, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 24, May 2014.

[23] Matthew Hammer, Bor-Yuh Chang, and David Van Horn. A vision for online verification-validation. In *GPCE '16: The 15th International Conference on Generative Programming: Concepts & Experience*, November 2016.

[24] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *OOPSLA '15: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, October 2015.

[25] Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1981.

[26] Vineeth Kashyap. *Configurable and Sound Static Analysis of JavaScript: Techniques and Applications*. PhD thesis, University of California, Santa Barbara, 2015.

[27] Shuying Liang. *Static analysis of Android applications*. PhD thesis, University of Utah, 2014.

[28] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2013.

[29] Shuying Liang, Matthew Might, and David Van Horn. AnaDroid: Malware analysis of Android with User-Supplied predicates. In *Proceedings of Tools for Automatic Program Analysis*, June 2013.

[30] Shuying Liang, Weibin Sun, Matthew Might, Andrew Keep, and David Van Horn. Pruning, pushdown Exception-Flow analysis. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014.

[31] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2010.

[32] Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent Higher-Order programs. In *Static Analysis*, volume 6887. Springer Berlin Heidelberg, 2011.

[33] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[34] Phúc C. Nguyên, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2014.

[35] Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. In *POPL '18: Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles in Programming Languages*, pages 51:1–51:30, New York, NY, USA, January 2018. ACM.

[36] Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming*, 27:e3, 2017.

[37] Phúc C. Nguyễn and David Van Horn. Relatively complete counterexamples for higher-order programs. In *PLDI '15: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2015.

[38] Jens Nicolay. *Purity Analysis for Higher-Order Imperative Languages: An Abstract Machine Approach*. PhD thesis, Vrije Universiteit Brussel, 2016.

[39] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In *Best of PLDI 1988*, volume 39. ACM, 2004.

[40] Christian Skalka, Scott Smith, and David Van Horn. A type and effect system for flexible abstract interpretation of Java (extended abstract). *Electronic Notes in Theoretical Computer Science*, 131, May 2005.

[41] Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(02), 2008.

[42] Quentin Stiévenart. *Scalable Designs for Abstract Interpretation of Concurrent Programs: Application to Actors and Shared-Memory Multi-Threading*. PhD thesis, Vrije Universiteit Brussel, 2018.

[43] Sam Tobin-Hochstadt and David Van Horn. Semantic solutions to program analysis problems. In *FIT Session, The ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI'11)*, June 2011.

[44] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *OOPSLA '12: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2012.

[45] Sam Tobin-Hochstadt and David Van Horn. From principles to practice with class in the first year. In *International Workshop on Trends in Functional Programming in Education*, June 2013.

[46] Neil Toronto, Jay McCarthy, and David Van Horn. Running probabilistic programs backwards. In Jan Vitek, editor, *Programming Languages and Systems*, pages 53–79. Springer Berlin Heidelberg, 2015.

[47] David Van Horn. *The Complexity of Flow Analysis in Higher-Order Languages*. PhD thesis, Brandeis University, 2009.

[48] David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2007.

[49] David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2008.

[50] David Van Horn and Harry G. Mairson. Flow analysis, linearity, and PTIME. In *Static Analysis*, volume 5079, chapter 17. Springer Berlin / Heidelberg, 2008.

[51] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2010.

[52] David Van Horn and Matthew Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM*, 54, September 2011.

[53] David Van Horn and Matthew Might. Systematic abstraction of abstract machines. *Journal of Functional Programming*, 22(Special Issue 4-5), 2012.

[54] Niki Vazou, Joachim Breitner, William Kunkel, David Van Horn, and Graham Hutton. Theorem Proving for All: Equational Reasoning in Liquid Haskell. *ArXiv e-prints*, June 2018.

[55] Niki Vazou, Éric Tanter, and David Van Horn. Gradual liquid type inference. In *OOPSLA '18: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2018. To appear.