David Van Horn
Research statement

I am interested in all aspects of program analysis and its applications to programming languages, software engineering, verification, and security. Specifically, I work on the design, implementation, and use of programming languages to enable the production of software that can be mechanically reasoned about, with the ultimate goal of enabling the construction of reusable, trusted software components. I have developed new techniques for designing, implementing, and verifying automated reasoning tools that work on programs written in modern, high-level languages; applied these techniques to build state of the art security vetting systems for analyzing Android smartphone applications; resolved important discrepancies between the functional and object-oriented communities' understanding of analysis; and proven novel lower and upper bounds on the complexity of analyzing programs.

Past I: Mechanically Verifying Temporal Properties of Programs

My work began by developing an approach to the verification of temporal properties of higher-order programs. With Scott Smith (Johns Hopkins University) and Christian Skalka (University of Vermont), I designed a programming language that enabled programmers to write *temporal assertions* and formulated a mechanical verification procedure for statically proving such assertions hold (JFP'08). Temporal assertions, which are predicates in a temporal logic stating a property of a program's trace up to the point of the assertion being made, can express a wide range of safety properties such as safe locking behavior, resource usage policies, and history- and stack-based security policies. We created an algorithm for soundly inferring temporal models of program events and assertions, and verified programmer-supplied temporal specifications using off-the-self modal logic model checkers.

Past II: Understanding the Computational Complexity of Program Analysis

Subsequently, I worked on understanding the complexity of basic program analysis. I sought to give an analytic understanding of how difficult it is to mechanically reason about the run-time behavior of programs. I studied a family of program analyses known as k-CFA, a well-known and widely used flow analysis framework. Flow analyses such as k-CFA are a ubiquitous and fundamental component of automated program reasoning—whether in the service of optimization, verification, or bug finding—and form the basis of almost all other kinds of program analysis. In its simplest form, it answers basic questions such as "what functions can be applied?" and "to what arguments?" In more sophisticated forms, it can answer complex queries about the behavior of programs. For example, the analysis I have developed for security vetting of smartphone applications can answer questions such as "can the microphone recorder ever be enabled while the push-to-record button is *not* pressed?" or "can the contents of the contact list ever be sent over the net to blacklisted domains?"

The "k" in k-CFA is an integer characterizing the extent to which the analysis is context-sensitive. 0-CFA is context insensitive, while 1-CFA makes context-sensitive distinctions based

on the most recently called procedure, 2-CFA makes distinctions based on the two most recent calls, and so on. Olin Shivers, k-CFA's inventor, wrote in a PLDI retrospective on it [3]:

> 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.

A natural question, then, is whether the empirically observed increases in costs can be understood analytically as *inherent* in the approximation problem being solved, or as a shortcoming in algorithm design. My dissertation resolves this question with an exact characterization of the complexity of k-CFA. For $k = 0$, the analysis is PTIME-complete (ICFP'07), but for any constant $k > 0$, it is EXPTIME-complete (ICFP'08). The EXPTIME lower bound validates empirical observations and proves there is no tractable algorithm for k-CFA, despite the great deal of effort to tame its cost.

The results of my PhD work sparked interest in the object-oriented (OO) community where k-CFA-style analyses have also been widely used. The interest came from the apparent contradictions of what was known in these two communities: for OO programs, k-CFA could be done in polynomial-time, yet was EXPTIME-hard for functional programs. With Matthew Might (University of Utah) and Yannis Smaragdakis (University of Athens, Greece), we investigated this seeming paradox (PLDI'10). Using my dissertation's insights about the relation of analysis to evaluation, we resolved the tension by boiling it down to subtle assumptions about the representation of behavioral values, i.e., the difference in closure and object representation accounts for the disparity in cost. The result yielded immediate practical applications: by mapping the object representation on to closures, we designed a novel, context-sensitive hierarchy of analyses that are polynomial in complexity.

I also turned to upper bounds, working with Jan Midtgaard (University of Aarhus, Denmark), and improved the best known 0-CFA algorithm by a logarithmic factor (HOSC, to appear). The result was an important correction to the theoretical understanding of 0-CFA. In a landmark paper by Heintze and McAllester [2], 0-CFA was shown to be as hard as a problem whose best known algorithm, unimproved since 1968, was $O(n^3)$. This result had been viewed as strong evidence of a "cubic bottleneck" in flow analysis that could not be overcome. Midtgaard and I proved this was incorrect, giving a subcubic 0-CFA algorithm. Our result reopens a seemingly closed problem for further research.

Past III: Designing Better Program Analyzers

The insights gleaned from the complexity investigation of my dissertation continue to guide my work on automated reasoning systems. One key goal has been to narrow the technical gaps between interpreters and analyzers. This should make it easy to design, implement, and verify analysis frameworks; make analysis more accessible to the non-specialist; and open the door for directly applying runtime technology to yield improvements in analytic technology.

With Might, I developed a systematic construction, which we called *abstracting abstract machines*, for deriving program analyses. The central idea is that a programming language's semantics can be transformed, by a simple turn-the-crank construction, into an analysis for soundly reasoning about programs written in that language. Using our approach, we were able

to analyze a number of language features often considered beyond the pale of existing analysis approaches. We derived analyses for reasoning about space consumption in a lazy language, security violations in a language with stack-inspection, and control effects in a higher-order imperative language with first-class control, to name a few (ICFP'10). The work was well received; it was nominated and accepted to appear as a research highlight in *Communications of the ACM* (CACM'11), and invited to appear in a special issue of *Journal of Functional Programming* (JFP'12). Moreover the simplicity of the approach has inspired other researchers: Luke Ong's group at Oxford analyzes the safety of highly concurrent Erlang programs using the approach,[1] and Harvard's course on optimizing compilers devotes a week to the idea.[2]

Might and I used the technique to analyze security vulnerabilities in Android applications, with the goal of helping Department of Defense programmers vet mobile applications for highly sensitive deployments. We use analysis and specification languages to verify Dalvik bytecode programs are free of malware.

We have also shown how to adapt the method to concurrent settings (SAS'11), and how to model so-called "pushdown" abstractions of programs that approximate programs by pushdown automata rather than finite state machines. The latter results is a much more precise characterization of the run-time stack, especially when combined with a notion of *abstract* garbage collection (ICFP'12). We have also applied the technique to derive sound analyses of the latest JavaScript standard: EcmaScript 5.0, including analysis of programs that use `eval`, a notoriously difficult feature to reason about. This work is being carried out by two undergraduates at Northeastern, which speaks to the accessibility of our approach.[3]

Together with a PhD student, I have recently developed a method for optimizing implementations of analyzers developed using the abstracting abstract machines approach. The resulting implementations are two to three orders of magnitude faster than previous implementations, making the analysis not only easy to construct, but easy to make highly performant.[4]

Current & Future: Mechanically Verifying Behavioral Software Contracts

I have also begun work on other kinds of verification; in particular, I have been working on the static verification of behavioral software contracts with Sam Tobin-Hochstadt (Northeastern). Behavioral contracts specify pre- and post-condition invariants on components. Such contracts are not merely assertions, as they are an agreement between a component's producer and consumer and establish which party is to blame the case a guarantee is not met. These guarantees are monitored at runtime and signal blame when contracts are violated.

We developed an approach that statically verifies software components meet their contracted specification by using a novel symbolic execution framework (OOPSLA'12). For this first step, we have implemented an interactive verification environment for a featureful language of contracts and used it to verify distributed, interactive video games that were developed in our undergraduate courses. A key lesson of this work is that run-time mechanisms for enforcing program invariants can be put to use for improving automated static reasoning.

[1] `http://mjolnir.cs.ox.ac.uk/soter/`

[2] `http://www.eecs.harvard.edu/~greg/cs252rfa12/`

[3] `https://github.com/dvanhorn/LambdaS5`, `http://arxiv.org/abs/1109.4467`

[4] `https://github.com/dvanhorn/oaam`, `http://arxiv.org/abs/1211.3722`

Moving forward, I plan to use my work on analysis and verification as a foundation for making advances along several axes on constructing reliable and trustworthy software.

My primary thesis is that language mechanisms that capture design knowledge can be leveraged to qualitatively improve automated reasoning about programs. The work, therefore, naturally follows two intertwined threads: (1) research on language mechanisms that capture design knowledge, and (2) research on leveraging such mechanisms for automated reasoning.

My work on contract verification is a first step in this direction, but much work remains. I am currently advising (jointly with Tobin-Hochstadt) a student who has made progress on increasing the power of our symbolic execution approach to incorporate control and data paths into the semantics so that the analysis engine can reason about control branches and data structure accesses. The work has important applications to type inference of "occurrence-typing" systems and is likely to lead to contract *inference* for higher-order languages.

Concurrently, I am looking into integrating more sophisticated tools in support of the simple theorem prover we currently employ. With Panagiotis (Pete) Manolios (Northeastern), I have been working on using ACL2—an industrial strength theorem prover and winner of the 2005 ACM Software System Award—to do a better job reasoning about first-order data and contracts. I have also begun to explore how ACL2 can be used as an engine for generating concrete counter-examples from abstract contract failures and how to use these results for counter-example guided abstraction refinement (CEGAR), a topic that has received little attention in the functional language community despite its successes in imperative program verification.

With Nikhil Swamy (Microsoft Research, Redmond), I am also working on using the F* framework and powerful Z3 model checker to automatically prove full correctness of programs with respect to their contracts. I developed an embedding of Contract PCF, a core model of behavioral contracts, into F*, a dependently typed language in the ML tradition, and used its powerful type checker to verify contract correctness.

Finally, I also plan to investigate novel contract systems that allow the expression of *temporal* properties. While the work with Skalka and Smith on this topic was successful, I would like to enhance the program logics to take advantage of the recent breakthroughs in so-called "nested word automata" [1] that enable the expression of properties involving balanced procedure call and returns. Coupled together with my work on pushdown analysis, there is a strong potential for verifying rich temporal contracts. Since temporal properties have important applications to security, I plan to investigate this work (with Might) in the context of security contract systems for Android applications.

References

[1] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):1–43, 2009.

[2] Nevin Heintze and David Mcallester. On the cubic bottleneck in subtyping and flow analysis. In *LICS '97: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, page 342. IEEE Computer Society, 1997.

[3] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In Kathryn S. McKinley, editor, *Best of PLDI 1988*, volume 39, pages 257–269. ACM, 2004.