RESEARCH STATEMENT

Thomas Gilray (thomas.gilray@gmail.com)

My primary research interest is precise and scalable static analysis of higher-order and dynamic, multi-paradigm languages. My research in static analysis is motivated by the increasing need for programming which is safe, high-level, and scalable. A problem arises from these conflicting priorities and the compromises which currently must be struck between them. Lower-level code can be highly performant, while, at the same time, more error-prone and difficult for programmers to develop and maintain. Higher-level code can be safer and more natural for a programmer to reason about, but also more difficult for compilers and static analyses to reason about and thus significantly less practical for compute-intensive problems. In my ongoing work, I intend to show how high-level programming can be made fast, while remaining safe, through the application of more sophisticated static analyses.

Static Analysis

Safe, performant, high-level programming can seem an oxymoron in the face of the seemingly inexorable conflict between these three goals. The apparency of this contradiction stems from the difficulty for either a human to understand his or her most complex programs at the computer's level or for a computer to understand them at the programmer's level. Static analysis enters to bridge the divide. Such an analysis aims to statically approximate dynamic program behaviors by examining a program's source text. This is essentially the same process a programmer undertakes when thinking through a program in terms of types, shapes, loop or inductive invariants, ownership and aliases, lifetimes, progress, and other abstract properties, except done with perfect mathematical rigor. When developing a static analysis, three primary concerns present themselves:

1. *Soundness:* A static analysis should provide definite information in the form of hard bounds on program behavior. This ensures that, even where imprecise, an analysis will provide accurate information which can be used to power valid program optimizations or to safely dispatch costly run-time overhead.

2. *Precision:* Analysis designers strive to improve the tightness of established bounds on program behavior and the degree to which an analysis will exclude false positives.

3. *Speed:* Precision must be obtained at a low computational cost (and worst-case complexity) so an analysis can be scaled up to production-sized codebases and efficiently see through layers of high-level abstraction.

The primary class of techniques I use to manage the trade-off between precision and complexity in a static analysis is that of polyvariance. The *polyvariance* of a static analysis is the degree and manner of structural differentiation used to keep different dynamic values distinct in their static approximations. For example, the seminal form of polyvariance was k-call sensitivity [25, 26], a technique keeping values from different calling contexts distinct. A 1-call sensitive analysis merges values propagated from the same most-recent call site, but distinguishes values passed from distinct call sites. A wide variety of strategies for polyvariance have been explored, each of which represents a heuristic for managing the compromise struck between precision and complexity and a gambit on the part of an analysis designer that targets of analysis will tend to behave a certain way. A call-sensitive analysis assumes program values will tend to correlate with their calling context. Another strategy, argument sensitivity [1], assumes function arguments will tend to correlate with one another.

Dynamic languages are notoriously difficult to analyze. Much work has gone into simply defining a correct semantics for Python and JavaScript [14, 24, 28], with perhaps the most compelling effort (at least, for the purposes of constructing a static analysis) being Guha et al.'s λ_{JS} [13] and its successor, Politz et al.'s λ_{S5} [22]. This approach reduces programs to a simple core language consisting of fewer than 35 syntactic forms, reifying the hidden and implicit complexity of full JavaScript as explicit complexity written in the core language. Desugaring is appealing for analysis designers as it gives a simple and precise semantics to abstract; however, it also presents one of the major obstacles to precise analysis as it adds a significant runtime environment and layers of indirection through it. Consider an example the authors of λ_{S5} use to motivate the need for their carefully constructed semantics: `[] + {}` yields the string `"[object Object]"`. Strangely enough, this behavior is correct as defined by the ECMAScript specification for addition—a complex algorithm encompassing

a number of special cases which can interact in unexpected ways [5]. The desugaring process for λ_{S5} replaces addition with a function call to `%PrimAdd` from the runtime environment. `%PrimAdd`, in turn, calls `%ToPrimitive` on both its arguments before breaking into cases. This means that for any uses of addition to return precise results, a k-call sensitive analysis requires an intractable $k \geq 2$ (exponential-time due to the structure of environments in higher-order languages [18, 29]). What seems to be needed are increasingly nuanced and adaptive forms of polyvariance which better suit the targets and goals of an analysis.

Abstract Abstract Machines, Generalizing Polyvariance, and Implementations

Generating models for dynamic, higher-order, and multi-paradigm languages, where a variety of features may interact in non-obvious ways, requires a careful and principled approach. Abstracting abstract machines (AAM) [16, 30] is a systematic method for developing a flow-analytic abstract interpretation [4, 20] given an (operational, small-step [21]) abstract-machine semantics. The concrete (exact) abstract-machine semantics encodes program states as a tuple of mathematical entities (with components such as a control expression or program counter and a stack or continuation). Program evaluation is then encoded as a series of steps within an infinite universe of these machine states. AAM finitizes this machine, yielding an abstract semantics in terms of abstract states and transitions within a finite universe of approximations. Specifically, AAM eliminates recursion and unboundedness within a semantics by using a store-passing-style interpreter which explicitly passes around a finite model of the heap, placing a great importance on the strategy used to allocate *abstract* addresses (addresses in this finite abstract heap which approximate multiple addresses allocated at run-time).

A Unified Methodology for Polyvariance

In my own work, I've developed a unified methodology for polyvariant (e.g., context-sensitive) static analysis in the context of the AAM approach [8, 10]. This originally began as a survey of the polyvariance being used across disparate methods to understand their similarities and differences [7] (awarded best student paper at TFP 2013). I then proceeded to show that the design space of polyvariant techniques exactly and uniquely corresponds to the design space of abstract allocators. This allows a unification and generalization of polyvariance as tunings of a single function. The myriad classic flavors of polyvariance, including call sensitivity, object sensitivity [19, 27], argument sensitivity [1], etc., can be easily recapitulated as strategies for allocation.

Introspective Polyvariance

This perspective opens up a broad design space of strategies for compromising between the complexity and precision of a static analysis and provides a safe and easy method for implementing each point in this space as a simple adjustment of the allocation function. Because every possible tuning of this function can be shown to induce a sound approximation (using a parametric Galois connection [17]), no restrictions need be placed on its behavior and even heuristics which directly introspect on the behavior of a static analysis may be employed to guide the polyvariance used and better adapt it to the target and goals of analysis.

I applied my allocation method to produce a specific adaptive style of polyvariance for continuations [11] which guarantees a perfectly precise modeling of the call stack at no asymptotic cost to analysis complexity, and requiring only a trivial change to analyses already in the AAM style. In fact, for an analysis written in Racket, and another in Scala, the technique was fully implemented by changing only a single line of code. I proved that analyses using this technique exhibit no return-flow conflation of values by formalizing a highly non-trivial bisimulation with an incomputable analysis that uses whole, unbounded stacks to ensure perfect stack precision. I collaborated with Michael D. Adams to mechanize and verify this proof using the Coq proof assistant and I evaluated the technique's impact on a set of Scheme programs, observing perfect return flows and an 80% average reduction in model sizes against a comparable existing technique.

Scaling and High-Performance Algorithms

A simple implementation method for these analyses is to encode them directly as operational interpreters. Production analyses are typically given a custom, hand-optimized implementation in a lower-level language like Java or C++. A more systematic way to optimize these polyvariant AAM-style analyses will be important

for scaling them up to production targets. An increasingly popular approach is to encode analyses in Datalog [31] (i.e., HornSAT and related constraint problems).

Another recent approach is to encode the analysis as linear-algebraic operations [23] (which may be parallelized on GPUs and multi-core systems). Unfortunately, this approach was restricted to a trivially simple (although Turing-equivalent) language with only a single transition rule (binary, continuation-passing-style λ-calculus). I built on this method, extending it to languages with any number of transition rules [9].

In pursuing this extension, I struck up a collaboration with James King whose expertise is in high-performance computing with GPUs. We implemented my encoding in CUDA and saw speedups as large as 20× (over a single-CPU version in C++), however we also observed no improvement on some benchmarks due to overhead from the compressed-sparse-row (CSR) matrix format [12] being rebuilt on every operation. The sparse-matrix format hybrid-ellpack [2] is better suited for dynamic matrix updates, but in the course of our investigation we were able to design a fundamentally better format called dynamic-CSR [15] inspired by traditional reallocating arrays. Our format supports SpMV (sparse matrix-vector multiply) competitive with CSR, exhibits nearly-linear strong scaling when parallelized across multiple GPUs, and can be extended in a time- and memory-efficient manner without restriction (unlike ellpack [6]). We also used this format to develop faster algorithms for SpMM (sparse matrix-matrix multiply) and algebraic multigrid. Our format has garnered interest from researchers working on high-performance sparse-graph applications, including industry leaders like SYSTAP. Our proceedings paper on this format won ISC's PRACE best paper award for 2016.

Ongoing Work

My ongoing work focuses on three orthogonal, but related and supporting, avenues of investigation:

1. *Introspective polyvariance:* I am pursuing further ways to adapt the polyvariance being used to a specific target by directly observing the behavior of an ongoing analysis or by bootstrapping off successive analysis results of intermediate precision. I want to develop an effective strategy of iterative refinement for polyvariant AAM as counterexample-guided abstraction refinement (CEGAR) [3] and similar approaches have done for verification by model checking and predicate abstraction.

2. *High-performance, parallel, and incremental implementations:* I am investigating multiple strategies for encoding flow analyses as linear algebra or as systems of constraints. I am studying incremental approaches to Datalog and have developed an incremental approach for operational interpreters. Incremental algorithms generally have a promising intersection with the bootstrapping approach to introspective polyvariance (successive analyses might be *healed* from existing results).

3. *Compiling abstract abstract machines:* Precision enhancing techniques for static analysis often reify classic compiler optimizations such as inlining, unrolling, and constant propagation within the models they produce. I believe it will be possible to directly perform safe, optimizing compilation (code emission) from sound, already-optimized AAM models.

References

[1] O. Agesen. The cartesian product algorithm. In *Proceedings of the European Conference on Object-Oriented Programming*, page 226, 1995.

[2] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[3] E. M. Clarke, O. Grumberg, S. Jha, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of Computer Aided Verification*, pages 154–169. ACM, 2000.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[5] ECMA. *ECMA-262 (ECMAScript Specification)*. ECMA, 5.1 edition, June 2011.

[6] M. Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: 10.1145/1391469.1391473.

[7] T. Gilray and M. Might. A survey of polyvariance in abstract interpretations. In *Proceedings of the Symposium on Trends in Functional Programming*, May 2013.

[8] T. Gilray and M. Might. A unified approach to polyvariance in abstract interpretations. In *Proceedings of the Workshop on Scheme and Functional Programming*, November 2013.

[9] T. Gilray, J. King, and M. Might. Partitioning 0-CFA for the GPU. *Workshop on Functional and Constraint Logic Programming*, September 2014.

[10] T. Gilray, M. D. Adams, and M. Might. Allocation characterizes polyvariance. *(In submission to) Proceedings of the International Conference on Functional Programming (ICFP)*, September 2016.

[11] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. V. Horn. Pushdown control-flow analysis for free. *Proceedings of the Symposium on the Principles of Programming Languages (POPL)*, January 2016.

[12] J. L. Greathouse and M. Daga. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.68.

[13] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *Proceedings of the European Conference on Object-oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010.

[14] D. Guth. A formal semantics of python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

[15] J. King, T. Gilray, M. Might, and R. M. Kirby. Dynamic sparse-matrix allocation on GPUs. *Proceedings of the International Supercomputing Conference (ISC)*, June 2016.

[16] M. Might. Abstract interpreters for free. In *Static Analysis Symposium*, pages 407–421, September 2010.

[17] M. Might and P. Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 260–274, January 2009.

[18] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 305–315, June 2010.

[19] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, January 2005.

[20] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2004.

[21] G. D. Plotkin. A structural approach to operational semantics. 1981.

[22] J. G. Politz, M. J. Carroll, B. S. Lerner, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *Proceedings of the Dynamic Languages Symposium*, 2012.

[23] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principles of Programming Languages (POPL)*, pages 511–522, January 2010.

[24] J. F. Ranson, H. J. Hamilton, and P. W. Fong. A semantics of Python in Isabelle/HOL. Technical Report CS-2008-04, Department of Computer Science, University of Regina, Regina, Saskatchewan, December 2008.

[25] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[26] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991.

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhotak. Pick your contexts well: Understanding object-sensitivity. In *Symposium on Principles of Programming Languages*, pages 17–30, January 2011.

[28] G. J. Smeding. An executable operational semantics for python. Master's thesis, Universiteit Utrecht, January 2009.

[29] D. Van Horn and H. G. Mairson. Deciding k-CFA is complete for EXPTIME. *ACM Sigplan Notices*, 43(9):275–282, 2008.

[30] D. Van Horn and M. Might. Abstracting abstract machines. In *International Conference on Functional Programming*, page 51, Sep 2010.

[31] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, 2005.