Research Statement

Michael D. Adams

1 Overview

My primary research focus is on programming languages, with an emphasis on (1) static analyses and security, (2) parsing, and (3) generic programming and meta-programming. However, my interest in programming languages is broad and includes all areas related to the design, implementation, and construction of programming languages, compilers, and software analysis tools. In addition, **a secondary research focus of mine is cybersecurity** due to the crossover with static analysis. My goal is to improve the process of programming and to help programmers more easily implement, reason about, improve the performance of, and secure their code.

I frequently publish in the top venues for my research area including POPL,[1] ICFP,[2] SPLASH[3]/OOPSLA,[4] and PLDI.[5] I also have been involved in the development of a number of languages and compilers including the Glasgow Haskell Compiler [GHC], the Chez Scheme compiler [Chez], the X10 language [X10], the Habit compiler [Habit], and the K Framework [K].

2 Static Analysis

A major thread of my research has been improving the precision, performance, and flexibility of static analysis. The ultimate goal being for static analysis to statically determine the runtime behavior of software and thus not only preemptively detect bugs and vulnerabilities but also enable compiler optimizations.

Type Recovery. In my work on type recovery [Adams 2011; Adams et al. 2011], I showed how to bring the complexity of type recovery analysis from $O\left(n^2\right)$ to be only $O\left(n \log n\right)$. There are a number of future directions I see for this research including generalizing to other analyses and eliminating the quadratic overhead of static single-assignment (SSA) representations of code during compilation.

[1] Principles of Programming Languages
[2] International Conference on Functional Programming
[3] Systems, Programming, Languages and Applications: Software for Humanity
[4] Object-Oriented Programming, Systems, Languages and Applications
[5] Programming Language Design and Implementation

JVM and Jaam. Starting in 2014, I worked on two DARPA programs (STAC[6] and APAC[7]) aimed at detecting software vulnerabilities in JVM (Java Virtual Machine) and Dalvic programs before they are deployed. On the theoretic side, this research lead to both the *Push-down for Free* and *Allocation Characterizes Polyvariance* results. This research also pushed not just theory but also practice, and as part of the project I developed Jaam, the JVM Abstracting Abstract Machine [U-Combinator 2016], as a practical tool for static analysis of JVM bytecode. It continues to be actively developed, and I anticipate continuing to use it as a research platform.

Push-down for Free (P4F). The *Push-down for Free* result [Gilray et al. 2016b] shows how to achieve perfect stack precision in certain analysis frameworks while also not incurring any asymptotic overhead (e.g., a $O(f(n))$ analysis remains $O(f(n))$). Previous methods of achieving perfect stack precision incurred quadratic overheads or worse (e.g., a $O(f(n))$ analysis becomes $O\left((f(n))^2\right)$). Possible future work in this includes generalizing it to other aspects of analysis and other analysis frameworks.

Allocation Characterized Polyvariance (ACP). The *Allocation Characterizes Polyvariance* result [Gilray et al. 2016a] shows how a wide variety of styles of analysis polyvariance can be achieved by choosing an appropriate allocation policy for abstract values. Furthermore, we proved that, when using the appropriate framework, all allocation policies produce sound analyses. Thus, the allocation policy becomes a tunable parameter that can be freely adjusted to match whatever sort of polyvariance is needed to achieve the desired analysis precision. This makes static analysis more expressive and customizable to provide the exact right information for the property being analyzed.

Smart Contracts. Looking forward, I have started working on the static analysis of smart contracts (e.g., Ethereum or other blockchain platforms), and I have an NSF SaTC[8] grant proposal for research in this area that is currently pending review. Security vulnerabilities in this area can easily lead to the loss of funds or assets. Indeed, there have already been $350M lost or stolen due to smart-contract vulnerabilities. However, by taking advantage of the structure of typical smart contracts, our analysis can explore deeper, more sophisticated properties than a typical analyzer. By analyzing these deep properties we can then prove strong security guarantees about the contracts being analyzed.

Relevant Publications:

[6] Space/Time Analysis for Cybersecurity. `https://www.darpa.mil/program/space-time-analysis-for-cybersecurity`

[7] Automated Program Analysis for Cybersecurity. `https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity`

[8] National Science Foundation: Secure and Trustworthy Cyberspace. `https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504709`

1. Thomas Gilray, **Michael D. Adams**, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, **ICFP '16**, pages 407–420. ACM, New York, NY, USA, September 2016. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951936.

2. Thomas Gilray, Steven Lyde, **Michael D. Adams**, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In Proceedings of the 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, **POPL '16**. ACM, New York, NY, USA, January 2016. doi: 10.1145/2837614.2837631.

3. Jan Midtgaard, **Michael Adams**, and Matthew Might. A structural soundness proof for Shivers's escape technique: A case for Galois connections. In Antoine Miné and David Schmidt, editors, Static Analysis, **SAS '12**, volume 7460 of Lecture Notes in Computer Science, pages 352–369. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-331 25-1_24.

4. **Michael D. Adams**, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. Flow-sensitive type recovery in linear-log time. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, **OOPSLA '11**, pages 483–498. ACM, New York, NY, USA, October 2011. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048105.

3 Parsing

Though parsing is sometimes thought of as a solved problem, there has been a recent resurgence of research on parsing. My work has improved both the theoretical and practical performance of existing parsing techniques as well as increased the expressiveness of grammars to more easily express common language patterns.

The Performance of Parsing with Derivatives. Parsing with derivatives is a parsing technique that makes it easy to implement parsers in a number of languages. As a result, it has been ported to a number of languages including Java, Python, Clojure, Haskell, Smalltalk, and miniKanren. However, prior to my work, the lowest computational bound computed for parsing with derivatives was exponential [Might et al. 2011]. In fact, high-level arguments claiming it was fundamentally exponential had been advanced and even accepted as part of the folklore. Performance was sluggish in practice, and this sluggishness was taken as informal evidence of exponentiality.

My research resolved both the theoretical and practical performance issues [Adams et al. 2016]. I showed that the previous complexity bound was overly pessimistic. Parsing with derivatives actually has a cubic bound the same as many other parsing techniques. This then left the question of why parsing with derivatives was slow in practice. I showed that simple (though perhaps not obvious) modifications greatly improved the practical performance leading to an implementation that is not only easy to understand but also performant in practice.

Restricting Grammars with Tree Automata. I have also extended parsing theory to cover situations that are not easily handled by traditional context-free grammars. For example, I have shown how to handle ambiguities by intersecting tree automata with context-free grammars [Adams and Might 2015, 2017]. The result is a modular system for composing grammatical restrictions. This forms a unified theory that subsumes many other ambiguity resolution techniques. Furthermore, I showed how simple tree automata that are expressible in a few lines encode many kinds of restrictions including both classic problems and ones found in real-world languages. In a case study, I found 18 and 16 instances where tree automata could simplify the C and JavaScript standard grammars, respectively. Finally, I showed that tree automata are well behaved and introduce no new shift/reduce or reduce/reduce conflicts when intersected with LR(k) grammars.

Indentation Sensitive Parsing. Another way that I have extended parsing theory is to handle indentation [Adams 2013; Adams and Ağacan 2014, 2016a,b]. Several popular languages including Haskell, Python, and F# use the indentation and layout of code as an essential part of their syntax. In the past, implementations of these languages used ad-hoc techniques to implement layout. These techniques tend to be low-level and operational in nature and forgo the advantages of more declarative specifications like context-free grammars. For example, they are often coded by hand instead of being generated by a parser generator.

I developed an extension to context-free grammars that can express these layout rules and derived LR(k), GLR, and PEG algorithms for parsing these grammars. These grammars are easy to write and can be parsed efficiently. Furthermore, the theory is compossible and allows local indentation definitions. This, in turn, makes it easy to extend the indentation rules of a language in ways not foreseen when the indentation rules were originally chosen.

Relevant Publications:

1. **Michael D. Adams** and Matthew Might. Restricting grammars with tree automata. Proceedings of the ACM on Programming Languages, 1(**OOPSLA ’17**):82:1–82:25, October 2017. ISSN 2475-1421. doi: 10.1145/3133906.

2. **Michael D. Adams**, Celeste Hollenbeck, and Matthew Might. On the complexity and performance of parsing with derivatives. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, **PLDI ’16**. ACM, New York, NY, USA, June 2016. doi: 10.1145/2908080.2908128.

3. **Michael D. Adams** and Ömer S. Ağacan. Indentation-sensitive Parsing for Parsec. In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, **Haskell '14**, pages 121–132. ACM, New York, NY, USA, 2014. doi: 10.1145/2633357.2633369.

4. **Michael D. Adams**. Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule. In Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, **POPL '13**, pages 511–522. ACM, New York, NY, USA, 2013. doi: 10.1145/2429069.2429129.

4 Generic Programming and Meta-programming

Another thread of my research is generic programming and meta-programming. These can have a profound impact on the programmer's ability to play with and extend a language, and giving programmers easy access to this power motivates my research in this area.

Efficient Generic Programming. Generic programming allows programmers to express high-level concepts (such as listing all identifiers in an abstract syntax tree) without having to write the low-level details of tree traversals. This avoids the tedium and repetition of hand writing tree traversals and reduces the chance of mistakes. This leads to higher-level, more-concise programs that allow the programmer to focus on the more important parts of the algorithm.

However, a number of generic programming systems have runtime performance problems. For example, Scrap Your Boilerplate [Lämmel and Peyton Jones 2003], a widely used generic-programming system for Haskell, often runs twenty times slower than non-generic hand-written code. Developers are thus faced with choosing between efficient but verbose hand-written code and concise but slow generic code. However, in my research I showed how to achieve the best of both worlds.

Template Your Boilerplate. First, I developed Template Your Boilerplate [Adams and DuBuisson 2012], which uses meta-programming to simulate the generic programming interface of Scrap Your Boilerplate. Thus programs can be written in the style of Scrap Your Boilerplate without the performance cost.

Optimizing Scrap Your Boilerplate. Later, I improved on Template Your Boilerplate by creating an optimization that works directly on Scrap Your Boilerplate code [Adams et al. 2014, 2015]. Essentially, it is a partial evaluation that takes advantage of domain-specific knowledge about the structure of Scrap Your Boilerplate. As a result, it improves the performance of Scrap Your Boilerplate to match that of hand written code without any special effort on the part of the programmer.

Macro Hygiene. A long-standing challenge in meta-programming systems is that of macro hygiene (i.e., avoiding unintended variable capture). However, while there are decades of research on various ways of implementing hygiene [Kohlbecker et al. 1986; Kohlbecker and Wand 1987; Bawden and Rees 1988; Clinger and Rees 1991; Clinger 1991; Dybvig et al. 1993; Herman and Wand 2008; Herman 2010] and a fairly standard *informal* definition, for a long time there was no formally precise way of specifying what hygiene is and whether a particular algorithm properly implements it. This made it difficult to reason about hygiene and difficult to compare two algorithms. For example, if two algorithms produce different results, it was impossible to say which one was correct. This is in stark contrast with lexical scope, alpha-equivalence and capture-avoiding substitution, which also deal with preventing unintended variable capture but have widely applicable and well-understood mathematical definitions.

In my research [Adams 2015], I developed precise, algorithm-independent, mathematical criteria for whether a macro expansion algorithm is hygienic. This characterization corresponds closely with existing hygiene algorithms and sheds light on aspects of hygiene that are usually overlooked in informal definitions. For example, this included the discovery of a new type of hygiene violation that some existing hygiene algorithms prevent but are overlooked in informal definitions. In future work, this mathematical understanding may lead to ways to make more expressive hygiene systems.

Relevant Publications:

1. William Mansky, Elsa L. Gunter, Dennis Griffith, and **Michael D. Adams**. Specifying and executing optimizations for generalized control flow graphs. **Science of Computer Programming**, 130:2–23, November 2016. ISSN 0167-6423. doi: 10.1016/j.scico.2016.06.003.

2. **Michael D. Adams**, Andrew Farmer, and José Pedro Magalhães. Optimizing SYB traversals is easy!. **Science of Computer Programming**, 112, Part 2:170–193, November 2015. ISSN 0167-6423. doi: 10.1016/j.scico.2015.09.003.

3. **Michael D. Adams**. Towards the Essence of Hygiene. In Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, **POPL '15**. ACM, New York, NY, USA, 2015. doi: 10.1145/2676726.2677013.

4. **Michael D. Adams**, Andrew Farmer, and José Pedro Magalhães. Optimizing SYB is easy!. In Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, **PEPM '14**, pages 71–82. ACM, New York, NY, USA, 2014. doi: 10.1145/2543728.2543730. (Received the **PEPM '14 Best Paper Award**.)

5. **Michael D. Adams** and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In Proceedings of the 2012 ACM SIGPLAN Haskell symposium, **Haskell '12**,

pages 13–24. ACM, New York, NY, USA, 2012. doi: 10.1145/2364506.2364509.

6. **Michael D. Adams**. Scrap your zippers: A generic zipper for heterogeneous types. In Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming, **WGP '10**, pages 13–24. ACM, New York, NY, USA, 2010. doi: 10.1145/1863495.1863499.

References

Michael D. Adams. *Flow-Sensitive Control-Flow Analysis in Linear-Log Time*. PhD thesis, Indiana University, 2011.

Michael D. Adams. Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429129.

Michael D. Adams. Towards the essence of hygiene. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 457–469, New York, NY, USA, January 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677013.

Michael D. Adams and Ömer S. Ağacan. Indentation-sensitive parsing for parsec. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 121–132, New York, NY, USA, September 2014. ACM. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633369.

Michael D. Adams and Ömer S. Ağacan. Indentation sensitive parsing combinators for Parsec, 2016a. URL `https://hackage.haskell.org/package/indentation-parsec`.

Michael D. Adams and Ömer S. Ağacan. Indentation sensitive parsing combinators for Trifecta, 2016b. URL `https://hackage.haskell.org/package/indentation-trifecta`.

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364509.

Michael D. Adams and Matthew Might. Disambiguating grammars with tree automata. In *Proceedings of Parsing@SLE*, October 2015.

Michael D. Adams and Matthew Might. Restricting grammars with tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):82:1–82:25, October 2017. ISSN 2475-1421. doi: 10.1145/3133906.

Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. Flow-sensitive type recovery in linear-log time. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 483–498, New York, NY, USA, October 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048105.

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. Optimizing SYB is easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 71–82, New York, NY, USA, January 2014. ACM. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543730.

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. Optimizing SYB traversals is easy! *Science of Computer Programming*, 112, Part 2:170–193, November 2015. ISSN 0167-6423. doi: 10.1016/j.scico.2015.09.003.

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. On the complexity and performance of parsing with derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 224–236, New York, NY, USA, June 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908128.

Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 86–95, New York, NY, USA, January 1988. ACM. ISBN 0-89791-273-X. doi: 10.1145/62678.62687.

Chez. Chez scheme, 2018. URL `http://www.scheme.com`.

William Clinger. Hygienic macros through explicit renaming. *ACM SIGPLAN Lisp Pointers*, IV(4):25–28, October 1991. ISSN 1045-3563. doi: 10.1145/1317265.1317269.

William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99607.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *LISP and Symbolic Computation*, 5(4):295–326, December 1993. ISSN 0892-4635. doi: 10.1007/BF01806308.

GHC. The glasgow haskell compiler, 2018. URL `http://www.haskell.org/ghc/`.

Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 407–420, New York, NY, USA, September 2016a. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951936.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 691–704, New York, NY, USA, January 2016b. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837631.

Habit. Habit, 2018. URL `http://hasp.cs.pdx.edu/`.

David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, USA, May 2010.

David Herman and Mitchell Wand. A theory of hygienic macros. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78738-9. doi: 10.1007/978-3-540-78739-6_4.

K. K framework, 2018. URL `http://www.kframework.org/`.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319859.

Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 77–84, New York, NY, USA, October 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41632.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN In-*

ternational Workshop on Types in Languages Design and Implementation, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8. doi: 10.1145/604174.604179.

Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: a functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 189–195, New York, NY, USA, September 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034801.

U-Combinator. Jaam: JVM abstracting abstract machine, 2016. URL `https://github.com/Ucombinator/jaam`.

X10. X10, 2018. URL `http://x10-lang.org/`.