Research Statement

Michael D. Adams

In broad strokes, my research area is programming languages, and I aim to help programmers more easily implement, reason about, prove correct, and improve the performance of their programs. Towards this end, my research has particularly focused on the three areas of **static analysis**, **meta- and generic programming**, and **parsing**.

My research straddles the divide between implementation and theory in order to produce tools and techniques that are both practical and theoretically elegant. Typical publication venues for my work include top-tier venues such as ICFP,[1] POPL,[2] OOPSLA,[3] and PLDI.[4] I am the lead developer of Jaam [U-Combinator 2016], a static analysis tool for JVM bytecode that is being developed for the DARPA STAC project. I have been involved in the development of a number of languages and compilers, including the Glasgow Haskell Compiler, the Chez Scheme compiler, the X10 language, the Habit compiler, the Hermit optimization system, and the K Framework. My research has also produced a number of libraries that are released as open-source software [Adams 2010; Adams and DuBuisson 2013; Adams and Ağacan 2016a,b; Adams et al. 2015b, 2016a; U-Combinator 2016].

In Section 1, Section 2, Section 3, and Section 4, I give an overview of my research and my perspective on these topics. For those who are interested, Appendix A, Appendix B, and Appendix C give more details about specific research I have done in these areas.

1 Overview

The overall theme of my research is increasing the ability of programmers to express programs in ways appropriate to their problem domain that are clear, concise, elegant, and efficient. This reduces development time and leads to faster design iteration. It also make programs easier to read, understand, and reason about, which reduces maintenance overhead and makes debugging easier.

Given the current state of the art, there is much foundational work to be done in this area, but the benefits are significant. Advances in this area have the potential to accelerate progress in all areas of technology. They also lower barriers to entry and increase access, as they allow domain specialists to write

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

programs in terms appropriate to the problem domain rather than having to cater to the machine.

My research on static analysis addresses this from the perspectives of both security and performance. My work is part of DARPA's APAC and STAC programs, which aim to increase our ability to detect software vulnerabilities and thus improve software security. As part of this, I have produced several results that reduce analysis time and thus make it more feasible to analyze code for deeper, more-sophisticated security properties.

My research on both meta-programming and parsing addresses this from the perspective of reducing both conceptual and computational costs associated with using advanced programming-language features. To reduce conceptual costs, I have developed techniques that are more composable and easier to reason about than traditional approaches. To reduce computational costs, I have developed techniques that are more efficient than traditional approaches both asymptotically and on real-world benchmarks. These advances aim to make it more practical for ordinary programmers to use advanced language features on realistic programming tasks and thus improve software quality overall.

2 Static Analysis

My research on static analysis applies to optimizing programs as well as detecting (or verifying the absence of) security vulnerabilities. Towards that end, the major advances in my research are foundational developments that improve the applicability, precision, and performance of a wide array of static analyses. For example, my research on flow-sensitive control-flow analysis (see Section A.1 and [Adams 2011; Adams et al. 2011]) shows that such analyses could be more efficiently implemented in $O(n \log n)$ time instead of the usual $O(n^2)$ time and thus scale to larger and more complicated analysis problems. This moves many analyses previously thought to be impractical into the realm of feasibility.

My more recent research on static analysis is part of the recently concluded DARPA APAC[5] program and the currently ongoing DARPA STAC[6] program. These aim to develop techniques to detect and prevent cybersecurity vulnerabilities before they are exploited. This has resulted in both theoretical advances and the Jaam analyzer [U-Combinator 2016] as a practical tool. As part of those programs, I collaborated on research on push-down analyses (see Section A.2 and [Gilray et al. 2016b]) that improved analysis precision and performance simultaneously. We developed a model of the call stack that is provably as precise as possible (given the other approximations in the analysis) and showed how to do this with no asymptotic overhead in either space or time. Finally, I collaborated on research on characterizing polyvariance (see Section A.3 and [Gilray et al. 2016a]) that showed that large families of static analyses can all be described in terms of a single parameter, the allocation strategy. This allows a single analyzer implementation to be flexible enough to accommodate different styles of polyvariance and makes it possible to build hybrid polyvariances that are finely tuned to the task at hand.

[5] Automated Program Analysis for Cybersecurity
[6] Space/Time Analysis for Cybersecurity

These research results are foundational and are applicable to a wide variety of analyses. Together they are initial steps towards improving analysis performance to the point where analyses for new, more-sophisticated properties will soon be practical for common use.

Another future research direction that I would like to explore is user-specified analyses. A critical aspect of any program's design is the abstractions chosen to represent the problem domain, but current static-analysis techniques provide poor support for specifying what these abstractions are and how they should be analyzed. This role is partially filled by sophisticated type systems where the user can encode the relevant properties in the type, but for many properties this is insufficient. It would be better to allow programmers to directly define the abstractions and properties to be analyzed so that these are tailored to their needs.

3 Meta-programming and Generic Programming

Giving the users of a language the ability to extend the language itself allows users to experiment and gradually find the best design choices and idioms. In particular, front-end features such as meta-programming and generic programming can have a profound impact on the programmer's ability to extend a language. This is particularly important with embedded domain-specific languages, where meta-programming and generic programming improve our ability to write programs in terms of concepts that are appropriate to the problem domain.

Giving programmers easy access to this power motivates several aspects of my research. One line of my research (see Section B.1 and [Adams and DuBuisson 2012]) produced an alternative to "Scrap Your Boilerplate" (SYB), the most widely used generic-programming library for Haskell. Unfortunately, SYB is infamously slow and thus cannot be used in many applications where it would otherwise be useful. My research produced a library that had a similar interface to that of SYB but was anywhere from two to twenty times faster than SYB. Later work refined this (see Section B.2 and [Adams et al. 2014, 2015a]) to achieve similar performance improvements with a compiler optimization pass over SYB code rather than a replacement library.

In other research (see Section B.3 and [Adams 2015]), I tackled the issue of formally defining "macro hygiene" (i.e., preventing unintended variable capture in programs that use macros and meta-programming). There is a long history of algorithms implementing this, but my work sought to formally specify the essential property that these algorithms preserve. This formal definition makes it easier to reason about macros and their interaction with other language features and extensions.

Ultimately, I view these as ways to give programmers the power to extend their own language and to customize the language to suit their problem domain. This makes programming a more approachable task for those skilled in a particular domain but not programming-language design. This encourages experimentation with novel perspectives and leads to better programming languages that in turn accelerate progress in all areas of computer science and technology.

4 Parsing

Parsing is at both the interface between human and machine and the front line in security. Though often thought of as a "solved problem", parsing has a surprising number of underexplored areas. This is especially true on the usability side, where existing tools are disappointingly cumbersome to use. For example, debugging a "shift/reduce conflict" or "reduce/reduce conflict" error from Yacc requires deep knowledge of parsing algorithms. As a result, parsing tools are not used in many places where they would be useful. For example, many security vulnerabilities result from improper or incomplete validation of input from untrusted sources.

My research has produced more composable and elegant ways of parsing indentation-sensitive languages like Python, Haskell, and F# (see Section C.1 and [Adams 2013; Adams and Ağacan 2014]). It has also shown how to efficiently implement parsing algorithms that were previously thought to be easy to implement, understand, and use but were too inefficient for practical use (see Section C.2 and [Adams et al. 2016b]). Finally, some of my research currently underway seeks a way to compositionally specify grammatical disambiguation rules while still being compatible with existing parsing technologies (see Section C.3 and [Adams and Might 2015]).

Much of this work is currently foundational, but ultimately the goal with this research is to lift parsing technology to be more usable by the common programmer. This requires tools that make it easier for programmers to specify and reason about parsing while being flexible enough to handle real-world situations.

Appendix A Static Analysis

A.1 Efficient Flow-sensitivity

The flexibility of dynamically typed languages such as JavaScript, Python, Ruby, and Scheme comes at the cost of needing to execute dynamic type checks at runtime. Some of these checks can be eliminated via control-flow analysis. However, traditional control-flow analysis (CFA) is not ideal for this task as it ignores flow-sensitive information that can be gained from dynamic type predicates, such as JavaScript's `instanceof` and Scheme's `pair?`, and from type-restricted operators, such as Scheme's `car`. Yet, adding flow-sensitivity to a traditional CFA worsens the already significant compile-time cost of traditional CFA. This makes it unsuitable for use in just-in-time compilers.

In my dissertation research [Adams 2011; Adams et al. 2011], I developed a fast, flow-sensitive type-recovery algorithm based on the linear-time, flow-insensitive sub-0CFA. The algorithm was implemented as an experimental optimization for the Chez Scheme [Dybvig 2010] compiler where it justified the elimination of about 60% of runtime type checks in a large set of benchmarks. The algorithm processes on average over 100,000 lines of code per second and scales well asymptotically, running in only $O(n \log n)$ time where traditional methods have a complexity to $O(n^2)$.

A.2 Push-down for Free

Traditional control-flow analysis (CFA) for higher-order languages introduces spurious connections between callers and callees, and different invocations of a function may pollute each other's return flows. Recently, three distinct approaches have been published that provide perfect call-stack precision in a computable manner: CFA2 [Vardoulakis and Shivers 2010], PDCFA [Earl et al. 2012], and AAC [Johnson and Van Horn 2014]. Unfortunately, implementing CFA2 and PDCFA requires significant engineering effort. Furthermore, all three are computationally expensive. For a monovariant analysis, CFA2 is in $O\left(2^n\right)$, PDCFA is in $O\left(n^6\right)$, and AAC is in $O\left(n^8\right)$.

My colleagues and I developed a new technique [Gilray et al. 2016b] that builds on these but is both straightforward to implement and computationally inexpensive. The crucial insight is an unusual state-dependent allocation strategy for the addresses of continuations. Our technique imposes only a constant-factor overhead on the underlying analysis and costs only $O\left(n^3\right)$ in the monovariant case.

A.3 Allocation Characterizes Polyvariance

The polyvariance of a static analysis is the degree to which it structurally differentiates approximations of program values. Polyvariant techniques come in a number of different flavors that represent alternative heuristics for managing the trade-off an analysis strikes between precision and complexity. For example, call sensitivity supposes that values will tend to correlate with recent call sites, object sensitivity supposes that values will correlate with the allocation points of related objects, the Cartesian product algorithm supposes correlations between the values of arguments to the same function, and so forth.

My colleagues and I developed a unified methodology for implementing and understanding polyvariance in a higher-order setting (i.e., for control-flow analyses) [Gilray et al. 2016a]. We build on the abstracting abstract machines method [Van Horn and Might 2010] by showing that the design space of possible abstract allocators exactly and uniquely corresponds to the design space of polyvariant strategies. This allows us to both unify and generalize polyvariance as tunings of a single function. Changes to the behavior of this function easily recapitulate classic styles of analysis and produce novel variations, combinations of techniques, and fundamentally new techniques.

Appendix B Meta-programming and Generic Programming

B.1 Template Your Boilerplate

Generic programming allows the concise expression of algorithms that would otherwise require large amounts of repetitive, handwritten code that obscures the essential design of a program. However, many of these systems are implemented in a way that delivers poor runtime performance relative to handwritten,

non-generic code. This poses a dilemma for developers. Generic-programming systems offer concision at the cost of performance. Handwritten code, on the other hand, offers performance but not concision.

My research [Adams and DuBuisson 2012] explored the use of Template Haskell to achieve the best of both worlds. It presents a generic-programming system for Haskell that provides both the concision of other generic-programming systems and the efficiency of handwritten code. Our system gives the programmer a high-level, generic-programming interface, but uses Template Haskell to generate efficient, non-generic code that outperforms existing generic-programming systems for Haskell. In this research, we benchmarked our system against both handwritten code and several other generic-programming systems. In these benchmarks, our system matches the performance of handwritten code while other systems average anywhere from two to twenty times slower.

B.2 Optimizing "Scrap Your Boilerplate"

The most widely used generic-programming system in the Haskell community, Scrap Your Boilerplate (SYB), also happens to be one of the slowest. Generic traversals in SYB are often an order of magnitude slower than equivalent handwritten, non-generic traversals. Thus while SYB allows the concise expression of many traversals, its use incurs a significant runtime cost. Existing techniques for optimizing other generic-programming systems are not able to eliminate this overhead.

My research [Adams et al. 2014, 2015a] presents an optimization that completely eliminates this cost. Essentially, it is a partial evaluation that takes advantage of domain-specific knowledge about the structure of SYB. It optimizes SYB-style traversals to be as fast as handwritten, non-generic code, and benchmarks show that this optimization improves the speed of SYB-style code by an order of magnitude or more.

B.3 Towards the Essence of Macro Hygiene

Hygiene is an essential aspect of Scheme's macro system that prevents unintended variable capture. However, previous work on hygiene has focused on algorithmic implementation rather than precise, mathematical definition of what constitutes hygiene. 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.

My research [Adams 2015] developed such a precise, mathematical definition of hygiene. It explored various kinds of hygiene violation and examples of how they occur. This resulted in an algorithm-independent, mathematical criteria for whether a macro expansion algorithm is hygienic. This characterization corresponds closely to existing hygiene algorithms and sheds light on aspects of hygiene that are usually overlooked in informal definitions.

Appendix C Parsing

C.1 Indentation-sensitive Parsing

Several popular languages, such as Haskell, Python, and F#, use the indentation and layout of code as part of their syntax. A robust syntactic extension or macro facility for these languages should thus be able to integrate with and take advantage of this aspect of the language. Because context-free grammars cannot express the rules of indentation, parsers for these languages currently use *ad hoc* techniques to handle 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. This makes it difficult to extend the syntax of such a language.

This research [Adams 2013] showed how a simple extension to context-free grammars can express these layout rules and how to derive GLR and LR(k) algorithms for parsing these grammars. These grammars are easy to write and can be parsed efficiently.

In addition, I extended [Adams and Ağacan 2014] this work to top-down combinator-based parsing frameworks such as Parsec. That research explores both the formal semantics of and efficient algorithms for indentation sensitivity. It derives a Parsec-based library [Adams and Ağacan 2016a] for indentation-sensitive parsing that is currently used by Idris [Brady 2013] to parse source code.

C.2 Parsing with Derivatives

Current algorithms for context-free parsing inflict a trade-off between ease of understanding, ease of implementation, theoretical complexity, and practical performance. No algorithm achieves all of these properties simultaneously.

Might et al. [2011] introduced parsing with derivatives, which handles arbitrary context-free grammars while being both easy to understand and simple to implement. Despite much initial enthusiasm and a multitude of independent implementations, its worst-case complexity has never been proven to be better than exponential. In fact, high-level arguments claiming it is fundamentally exponential have been advanced and even accepted as part of the folklore. Performance ended up being sluggish in practice, and this sluggishness was taken as informal evidence of exponentiality.

In this research, we reexamined the performance of parsing with derivatives [Adams et al. 2016b]. We have discovered that it is not exponential but, in fact, cubic. Moreover, simple (though perhaps not obvious) modifications to the implementation by Might et al. [2011] lead to an implementation that is not only easy to understand but also highly performant in practice.

C.3 Restricting Grammars with Tree Automata

Precedence and associativity declarations in systems like Yacc resolve ambiguities in context-free grammars (CFGs) by specifying restrictions on allowed

parses. However, they are special purpose and do not handle many other grammatical restrictions that language designers need to resolve things like dangling else, interactions between binary operators and if expressions in ML, and interactions between object allocation and function calls in JavaScript. Often, language designers resort to restructuring their grammars in order to encode these restrictions, but this obfuscates the designer's intent and makes grammars difficult to read, write, and maintain.

This currently underway research [Adams and Might 2015] shows that tree automata can modularly and concisely encode such restrictions. We do this by reinterpreting CFGs as tree automata and intersecting the result with tree automata encoding the desired restrictions. The output of this process is then reinterpreted back into a CFG that encodes the specified restrictions. This process can be used as a preprocessing step before further processing of the grammar and is well behaved. It performs well in practice and never introduces ambiguities or LR(k) or LL(k) shift/reduce or reduce/reduce conflicts.

References

Michael D. Adams. Scrap your zippers, 2010. URL `https://hackage.haskell.org/package/syz`.

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 Thomas M. DuBuisson. Template your boilerplate, 2013. URL `https://hackage.haskell.org/package/TYB`.

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

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 2015a. ISSN 0167-6423. doi: 10.1016/j.scico.2015.09.003.

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. Hermit SYB, 2015b. URL `https://github.com/xich/hermit-syb/`.

Michael D. Adams, Celeste Hollenbeck, and Matt Might. Derp 3, 2016a. URL `https://bitbucket.org/ucombinator/derp-3`.

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 2016b. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908128.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013. ISSN 1469-7653. doi: 10.1017/S095679681300018X.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.

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*, ICFP '12, pages 177–188, New York, NY, USA, September 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364576.

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.

James Ian Johnson and David Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 11–22, New York, NY, USA, October 2014. ACM. ISBN 978-1-4503-3211-8. doi: 10.1145/2661088.2661098.

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`.

David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, September 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863553.

Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In Andrew Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 570–589. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-11957-6_30.