Michael D. Adams: Research Statement

1 Research Vision

My research vision is for programmers to be able to communicate their intent to a computer (i.e., write programs) as *quickly* as and at the same *high level* that they can communicate to other programmers and that the resulting software artifacts would be clear enough that they obviously have no bugs instead of having no obvious bugs.

My research areas are **programming languages** and **software engineering**, and my research is aimed at achieving this future. As such, the **thread that connects all my research** is tools that **enable programmers to write clear and elegant code**. I view this as key to improving programmer **productivity** and **code comprehension** and allowing programmers to more effectively design and implement programs. These also have positive effects on other research areas. For example, better languages and tools can make it easier to detect **security** vulnerabilities or prevent them in the first place, and better languages and tools can also **make programming more approachable** and aid in **computer-science education**.

Section 2 discusses my **current and future research** directions, and **Section 3** discusses my **past and continuing research**.

2 Current and Future Work

2.1 Hazel and Structured Editing

I am currently working as a research scientist on the **Hazel** project [17], which focuses on **structured editing**. Structured editing offers leverage on many language design and user interface problems. For example, it can **improve language discoverability** and can bypass parsing and the sorts of parse errors that can particularly frustrate **new programmers**. Structured editing can also **improve feedback provided to programmers** by development environments. For example, in traditional IDEs, the assistance provided by static analyses, style checkers, and type checkers is often available only when the code is syntactically correct. However, with structured editing, not only can these tools run continuously, but they can also take advantage of the record of edit actions performed by the user. For example, the location of the most recent edit could inform where a type error should be reported when it could be located in multiple places.

Within Hazel, I am currently working on **collaborative editing between multiple users**. This involves developing conflict-free replicated data types (CRDTs) for representing abstract syntax trees, and I am taking an approach that will **unify the problems of version control and collaborative editing**. Beyond this work, there are many directions that I plan to continue research with Hazel.

2.2 Lattice-Oriented Programming

There is a large class of algorithms that amount to finding **a least fixed point on a lattice**. Examples include **static analysis**, **parsing** (both generating a parser and the act of parsing itself), and many **graph algorithms**. However, most languages provide little support for these sorts of algorithms. Developers have to write a “work-list” style loop and compute what needs updating on their own. Having a compiler able to operate at the level of lattices would allow it to perform **algorithm-level optimizations**.

As an example, consider computing minimum path-lengths in a graph. This is easily specified by the lattice rule that if the distance between A and B is at most l_1 and the distance between B and C is at most l_2, then the distance between A and C is at most $l_1 + l_2$. With relatively simple heuristics, a lattice-aware compiler could **use just this rule** and **invent Dijkstra’s algorithm *for* the programmer**.

2.3 Domain Specific Languages and Extensible Languages

Domain specific languages (DSLs) allow programmers to use **concepts and notations that match the problem domain**. In doing so DSLs make it easier to write clear and elegant code. Embedding a DSL in an existing language allows programmers to extend languages without having to start from scratch, and one of my research goals is to improve the ways in which **programmers can extend languages**. For example, allowing programmers to safely and soundly extend the type system or introduce domain-specific error reporting.

One area that I am pursuing is the improvement of **syntactic macros**. That is to say allowing the programmer to extend the grammar of the language itself. In fact much of my **parsing research** is ultimately aimed at making this possible. For example, my work on restricting grammars with tree automata [8] is aimed at **solving compositionality problems** that arise when mixing the syntaxes of different language extensions.

3 Past and Continuing Work

3.1 Static Analysis [2, 9, 15, 16]

A major aspect of my research has been improving the **expressiveness**, **precision**, and **performance** of static analysis. The ultimate goal being for static analysis to 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 [2, 9], I showed how to bring the complexity of type recovery from $O\left(n^2\right)$ to be only $O\left(n \log n\right)$. Possible future work in this includes generalizing to other analyses and eliminating the quadratic overhead of static single-assignment (SSA) representations.

APAC and STAC. From 2014 to 2019, I worked on two DARPA[1] programs (APAC[2] and STAC[3]) aimed at detecting software vulnerabilities in JVM (Java Virtual Machine) and Dalvik (Android) programs before they are deployed. This research lead to both the **Push-down for Free** [16] and **Allocation Characterizes Polyvariance** [15] results.

Push-down for Free [16] shows how to achieve perfect stack precision with **no asymptotic overhead**. (Previous methods incurred quadratic overheads or worse.) Possible future work in this includes generalizing it to other aspects of analysis and other analysis frameworks.

Allocation Characterizes Polyvariance [15] shows how a wide variety of analysis polyvariance can be generalized by choosing an appropriate **allocation policy**. Furthermore, **all allocation policies produce sound analyses**. Thus, the allocation policy is a tunable parameter that can be freely adjusted to match whatever sort of polyvariance is needed. This makes static analysis more **expressive** and **customizable** to provide the exact right information for the property being analyzed.

3.2 Parsing [3, 5, 7, 8, 13, 14, 12]

Though parsing is sometimes thought of as a solved problem, there has been a recent resurgence of research on parsing. I have 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. However, prior to my work, the lowest computational bound for parsing with derivatives was **exponential** [18]. My research [13] lowered this bound to a **cubic** bound (the same as many other parsing techniques). I also showed that simple modifications greatly improved **practical performance** leading to an implementation that is both **easy to understand** and **performant in practice** [14].

[1] Defense Advanced Research Projects Agency

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

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

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** [7, 8]. 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 tree automata **expressible in a few lines** encode many kinds of restrictions needed in both classic problems and real-world languages.

Indentation Sensitive Parsing. Another way that I have extended parsing theory is to handle indentation [3, 5]. 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.

I developed an extension to context-free grammars that can express these layout rules and derived LR(k), GLR, and PEG parsing 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 makes it easy to **extend** the indentation rules of a language in ways not foreseen when the indentation rules were originally chosen.

3.3 Generic Programming and Meta-programming [1, 4, 6, 10, 11]

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

Efficient Generic Programming. Generic programming allows programmers to elegantly express high-level concepts (such as listing all identifiers in an abstract syntax tree). This leads to **higher-level programs** that allow the programmer to focus on the more important parts of the algorithm **without having to write low-level details**. However, many generic programming systems have runtime performance problems, being up to twenty times slower than non-generic code. This forces programmers to choose between efficient but verbose code and elegant but slow code. My research showed how to achieve the **best of both worlds**.

First, I developed Template Your Boilerplate [6], 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.

Later, I improved on Template Your Boilerplate by creating an optimization that works directly on Scrap Your Boilerplate code [10, 11]. 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 avoiding unintended variable capture (i.e., macro hygiene). However, while there are decades of research on various ways to *implement* hygiene and a standard *informal* definition, prior to my work there was no *formal* way of specifying **what hygiene is** and **whether an algorithm implements it**. This is in stark contrast to 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 [4], I developed **precise, algorithm-independent, mathematical criteria** for whether macro expansion algorithms are hygienic. This characterization corresponds closely with existing hygiene algorithms and sheds light on aspects of hygiene that are usually overlooked in informal definitions.

References

[1] Michael D. Adams. Scrap your zippers: a generic zipper for heterogeneous types. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 13–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0251-7. doi: 10.1145/1863495.1863499.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] Pierce Darragh and Michael D. Adams. Parsing with zippers (fuctional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP):108:1–108:30, August 2020. ISSN 2475-1421. doi: 10.1145/3408990.

[15] 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 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951936.

[16] 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 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837631.

[17] Hazel. Hazel: A live functional programming environment featuring typed holes, 2019. URL `https://hazel.org/`.

[18] 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.