1 Research Vision

In my vision of an ideal future, programmers would 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 the resulting software artifacts would be clear and concise enough that they obviously have no bugs instead of having no obvious bugs.

My research area is programming languages, and my research is aimed at achieving this future. As such, the thread that connects all of my research is language tools and features that enable programmers to write clear, concise and elegant code and do so without sacrificing other things like performance. 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. The future I describe is a ways off, but it is one that I am excited to work towards.

2 Past Work

2.1 Static Analysis

A major thread 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 [1, 10], I showed how to bring the complexity of type recovery from $O(n^2)$ to be only $O(n \log n)$. 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.

APAC and STAC. Starting in 2014, I worked on two DARPA programs (APAC\footnote{Automated Program Analysis for Cybersecurity. https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity} and STAC\footnote{Space/Time Analysis for Cybersecurity. https://www.darpa.mil/program/space-time-analysis-for-cybersecurity}) aimed at detecting software vulnerabilities in JVM (Java Virtual Machine) and Dalvik (Android) programs before they are deployed. On the theoretic side, this research lead to both the Push-down for Free\footnote{https://www.darpa.mil/program/push-down-for-free} and Allocation Characterizes Polyvariance\footnote{https://www.darpa.mil/program/allocation-characterizes-polyvariance} results.

The Push-down for Free result\footnote{https://www.darpa.mil/program/push-down-for-free} 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., an $O(f(n^2))$ analysis becomes $O(f(n^2)^2)$). Possible future work in this includes generalizing it to other aspects of analysis and other analysis frameworks.

The Allocation Characterizes Polyvariance result\footnote{https://www.darpa.mil/program/allocation-characterizes-polyvariance} 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.
2.2 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 [18]. 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 [13]. 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 [8, 9]. 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.

Indentation Sensitive Parsing. Another way that I have extended parsing theory is to handle indentation [2, 4, 5, 6]. 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 parsing algorithms for parsing these grammars. These grammars are easy to write and can be parsed efficiently. Furthermore, the theory is composable 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.

2.3 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 [17], 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.

First, I developed Template Your Boilerplate [7], 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 [11, 12]. 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 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 [3], 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.

3 Ongoing and Future Work

Beyond the past work that I have done, there are a number of directions that I would like to pursue in my future research. In particular, exploring **new programming paradigms** and how they can allow programmers to express code clearly, concisely and elegantly.

3.1 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.

As an example, consider the problem of computing minimum path-length in a graph. This problem 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 would be able to use just this rule and invent Dijkstra's algorithm for the programmer.

Having a compiler able to operate at this high level would allow it to perform **algorithm-level optimizations**. This is particularly useful with fixed-point computations as small changes in problem representation can require rewriting almost all of the algorithm. For example, I was once writing such a work-list-based algorithm for tree automata. The initial version represented tree automata as lists of productions. This made the algorithm easy to write, but it performed poorly, so after benchmarking I modified the representation to index on non-terminal names. This required rewriting the entire algorithm and made the algorithm more difficult to understand. While that did improve the performance, it was not enough. So I modified the representation to also index in constructor name. This in turn required another rewrite of the algorithm into a more difficult to understand form. After these transformations, it was non-trivial to determine if the function being computed was even the same as what I started with. Specifying the computation at a higher level would have allowed the compiler to perform the work of adapting the algorithm to the representation.
This would allow us to be certain that the function computes the same value as it did before the change of representation.

3.2 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 they make it easier to write clear and concise code. Embedding a DSL in an existing language allows programmers to extend existing languages without having to start from scratch, and one of my research goals is to improve the ways in which a programmer can extend a language. For example, it would be beneficial to allow programmers to safely and soundly extend the type system or introduce domain-specific error reporting.

One area that I would like to pursue 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 is aimed at solving the compositionality problems that arise if the syntaxes of two different language extensions are mixed. This also connects back to my work on Hazel, in which livelits (i.e., live literals) are being developed that allow programmers to extend the language with graphical or GUI elements.

Another area I would like to pursue is allowing extensible and domain-specific optimizations. My work with the Hermit team demonstrated how a domain-specific optimization can be used to eliminate overheads in programs for that domain. I would like to research ways to ensure programmer-provided optimizations are sound. For example, in the Hermit system this was achieved by building up the optimization from correctness-preserving transformations that are provided by Hermit itself. However, there are more general approaches that could be explored. For example, the state graph of a static analysis encodes all possible code paths. From this one could extract optimized source code. Then things like increased polyvariance in the analysis correspond to things like inlining in the optimization. The Allocation Characterizes Polyvariance result allows us turn over the choice of allocation strategy to the programmer while still guaranteeing the correctness of the analysis. Thus, the extracted code is guaranteed to be correct regardless of the allocation strategy. This bypasses the problem in traditional optimizations of proving that an analysis properly corresponds to its transformation. With this approach, the optimization transformation is directly determined by analysis.

3.3 Hazel and Structured Editing

I am currently working as a research scientist on the Hazel project, 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.

My work on Hazel began in Fall 2019, so my work in this area is only just beginning, but I am currently working on concurrent editing. This involves developing conflict-free replicated data types (CRDTs) for representing abstract syntax trees, and I am taking an approach that I hope 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.

References


