Research Statement
Michael D. Adams

1 General Interests

My research focus is programming languages, with an emphasis on functional programming. My work revolves around the design, implementation, and construction of programming languages, compilers, and software analysis tools. My goal is to help programmers more easily implement, reason about, prove correct, and improve the performance of their code. I have a broad background within the area of programming languages, and my research interests include static-analysis, meta-programming, compilers, type systems, formal verification, and domain-specific languages. My research straddles the divide between implementation and theory in order to produce tools and techniques that are both practical and theoretically elegant. 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 and the Habit compiler.

2 Static Analysis

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 response, I developed in my dissertation research [Adams 2011; Adams et al. 2011] a fast, flow-sensitive type-recovery algorithm based on the linear-time, flow-insensitive sub-CFA. 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 of $O(n^2)$. This compile-time performance and scalability was achieved through a novel combination of data structures and algorithms.
This research also led to several indirect results that have implications beyond the original research. One indirect application is a more efficient representation of control-flow graphs than traditional static single assignment (SSA) representations [Cytron et al. 1991]. Another is an improvement on the constant factors in applicative random-access stacks [Myers 1983]. Finally, this research led to a method of computing forward and backward abstract interpretation [Cousot and Cousot 1992] simultaneously in a single pass.

There are many ways to extend this research. One is generalizing the techniques to higher-degree $k$CFA, points-to analysis, and other species of CFA such as $\Delta$CFA, CFA2, and LFA [Landi and Ryder 1992; Vardoulakis and Shivers 2010; Might 2007; Might and Shivers 2006]. Another is extending these techniques to handle strong update and reflow semantics. This will improve the precision of the analysis for many programs, but in these scenarios, the data structures that optimize path-sensitivity and flow-sensitivity have to be dynamically updated. Doing this efficiently while maintaining the linear-log bound of the original result is an interesting challenge that is likely to lead to new algorithms and techniques for static analysis.

3 Meta-programming and Generic Programming

Giving the users of a language the ability to extend the language itself is essential for maintainable language design and growth [Steele 1999]. This allows users to experiment and gradually find the best design choices and idioms. In particular, front-end features such as generic programming and meta-programming can have a profound impact on the programmer’s ability to play with and extend a language. Giving programmers easy access to this power motivates several aspects of my research.

3.1 Efficient Generic Programming

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 aims to eliminate this dilemma. I have shown how to build meta-programming based systems to implement generic programming systems that match the performance of handwritten code [Adams and DuBuisson 2012], and I am developing optimizations to improve the performance of generic-programming systems [Adams et al. 2013] to produce predictable and robust performance outcomes.
3.2 Macro Hygiene

Macro hygiene ensures that binders scope properly and do not incorrectly capture references. Much has been written on this topic [Kohlbecker et al. 1986; Clinger and Rees 1991; Dybvig et al. 1993], but the definition of hygiene is still somewhat informal compared to, for example, the notion of alpha equivalence. On the other hand, nominal logic [Pitts 2003] gives a rigorous account of the calculus of binders and references. Although macro hygiene and nominal logic are both fundamentally about the same thing (i.e., how binders and references interact), they take very different views of the problem, and I am working on establishing the formal connection between the two. This connection will make reasoning about macros easier and enable further advancements in macro and meta-programming theory.

3.3 Macro Parsing

Though the Scheme macro system is quite sophisticated in how it handles hygiene, the support it provides for parsing the input to a macro is fairly primitive. For example, the define-record-type macro requires a lot of work to parse using the facilities provided by standard Scheme. Recent work [Culpepper and Felleisen 2010] simplifies this to an extent, but even relatively simple-to-parse macros may still lead to unexpected problems. For example, a naively written let* macro takes quadratic time to expand because as each successive binder is processed, the entire binding list is re-parsed by each recursive macro call.

By recasting the problem in terms of regular tree automata, I believe these issues can be avoided. Regular tree automata are well understood, are the natural automata for parsing the inherently tree-based structure of s-expressions and macro syntax, and can be parsed in linear time. While well understood, using them in a macro expander poses novel challenges. Macros are not simply tree parsers. In general, they are tree transducers. In addition, they must interact properly with hygiene. Efficiently handling both of these factors is an area of ongoing research that I am pursuing.

4 Other Research

I have also produced research results in a number of other areas. Some of these were discovered while pursuing other research, but all of these at some level relate to my central interest of allowing greater programmer expressibility and program analyzability.

4.1 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

---

1 The work by Herman 2010 in this area applies only to restricted cases.
Research Statement, Michael D. Adams

4

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.

I have shown how a simple extension to context-free grammars can express these layout rules, and I have shown how to derive GLR and LR(k) algorithms for parsing these grammars [Adams 2013]. These grammars are easy to write and can be parsed efficiently. I plan to extend this work to include other parsing styles such as LL, GLL [Scott and Johnstone 2010] and PEG [Ford 2004].

4.2 Zipper Data Types

The zipper type provides the ability to edit tree-shaped data efficiently in a purely functional setting by providing constant time edits at a focal point in an immutable structure [Huet 1997]. It suffers from two major limitations, however. First, it operates only on homogeneous types. Thus, every node the zipper visits must have the same type. Second, it involves a significant amount of boilerplate code. Thus, a custom implementation must be written for each type the zipper traverses. This is error-prone, and the implementation must be updated whenever the type being traversed changes.

The generic zipper developed in my research overcomes these limitations [Adams 2010]. Using a combination of common type features, it operates over any type and requires no boilerplate code.

4.3 Efficient Applicative Random-Access Stacks

Myers introduced a variant of cons lists that allows indexing to any element of a list in logarithmic time [Myers 1983]. Though other structures like balanced trees, finger trees, and flexible arrays also allow indexing to any element in logarithmic time, the variant introduced by Myers is distinctive in that, except for set-cdr!, it supports all the same operations as a cons list in the same or better asymptotic time than a traditional cons list. These properties made Myers stacks particularly useful as efficient building blocks in other algorithms such as my dissertation work on control-flow analysis. However, while Myers stacks are asymptotically optimal, they do not have optimal constants. In particular, indexing operations in Myers stacks take \(3 \log n + O(\log \log n)\). My research shows that this can be improved to \((1 + \epsilon) \log n + O(\log \log n)\) given an arbitrarily small \(\epsilon > 0\). A paper on this is currently in preparation.

4.4 Efficient Equality

The Revised 6 Report on Scheme [Sperber et al. 2007] requires that the generic equivalence predicate, equal?, terminates even on cyclic inputs. While equal? can be implemented via DFA-equivalence or union-find algorithms, these algorithms usually require an additional pointer to be stored in each object, and
may be unacceptably slow for the small acyclic values that are the most likely inputs to the predicate. I developed a variant of the union-find algorithm that addresses these issues \cite{AdamsDybvig2008}. It performs well on large and small, cyclic and acyclic inputs by interleaving a low-overhead algorithm that terminates only for acyclic inputs with a more general algorithm that handles cyclic inputs. The algorithm terminates for all inputs while never being more than a small factor slower than whichever of the acyclic or union-find algorithms would have been faster. Several intermediate algorithms were also developed, each of which might be suitable for use in a particular application.

4.5 Cache-Oblivious Programming

Some of my earliest research was on expressing matrix algorithms in a way that simultaneously requires minimal effort from the programmer and delivers maximum performance from the machine \cite{AdamsWise2006b, Gottschling2007, AdamsWise2006a}. For these sorts of problems, minimizing cache-miss rates is critical to optimize but also difficult for programmers to manage. My research showed that by changing the representation of the matrix, cache-oblivious techniques can be not only competitive but actually beat hand-tuned implementations in multiple metrics, including cache-miss rates, TLB-miss rates and most importantly total computation time.
References


Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems


