

# Research Statement

Michael D. Adams

February 28, 2011

## 1 General Interests

My research relates to the design, implementation, and construction of programming languages, compilers, and software analysis tools that help programmers more easily implement, reason about, prove correct, and improve the performance of their programs. A major direction of my research and the focus of my dissertation is control-flow analysis (CFA) and other forms of static analysis. However, I have a broad background and interests within the area of programming languages including compilers, macros, type systems, formal verification, and domain specific languages. These areas might seem quite disparate, but I view them all as facets of how to express complex ideas elegantly and derive masterful complexity from masterful simplicity. This is a common thread connecting my interests.

## 2 Control-Flow Analysis

My dissertation research is focused on improving the asymptotic bounds of flow-sensitive and predicate-awareness control-flow analysis. It shows how to add flow-sensitivity and predicate-awareness to sub-0CFA [4] while increasing the computational complexity from  $O(n)$  to only  $O(n \log n)$  where traditional methods would increase the complexity to  $O(n^2)$ . This is achieved by a novel combination of known (but uncommon) algorithms. This research is part of an effort to develop an analysis with a low worst-case cost suited for use as part of a dynamic type-check elimination optimization to be integrated as a standard optimization in the Chez Scheme compiler [8]. More details can be found in the draft paper at:

[http://www.cs.indiana.edu/~adamsmd/papers/fast\\_flow\\_sensitive\\_cfa/](http://www.cs.indiana.edu/~adamsmd/papers/fast_flow_sensitive_cfa/)

### 2.1 Extensions

There are many ways that I plan to extend my dissertation research. For example I want to extend it to other levels of the CFA hierarchy. Preliminary examination indicates that the techniques directly translate to 0CFA. However for 1CFA and higher degree kCFA, the situation is more complex. 1CFA is

EXPTIME complete [16] so the cost of traditional methods for flow-sensitivity is comparatively small in the worst case. However, the common case for 1CFA is significantly cheaper than the worst case. In these cases the extra cost of traditional flow-sensitivity may be worth eliminating using the techniques from my dissertation work. In addition, ideas analogous to the algorithms used to efficiently represent flow sensitivity may improve the representations of context sensitivity.

I also plan to explore the applicability of these techniques in other species of CFA such as  $\Delta$ CFA, CFA2, LFA [17, 12, 13]. In particular it is worth exploring the applicability of these techniques to Points-to analysis [11] as in that domain higher-degree context sensitivity is still polynomial time [14]. Thus improving the efficiency by a linear factor can have a noticeable impact.

Another avenue to explore is handling strong update and reflow semantics. When a closure containing a strong update is called that update must be reflected in the context where it is called. This changes the flow structure of the program's variables. To account for this the data structures from my dissertation research that optimize flow-sensitivity have to be dynamically updated. Doing this efficiently while maintaining the linear-log bound of the original result is an interesting challenge which will likely require new algorithmic techniques.

## 2.2 Indirect Applications

The techniques developed during my dissertation research also have applicability to areas other than CFA. For example, the traditional method for selecting join points for flow information is to use static single assignment (SSA) and  $\varphi$ -nodes [7]. But SSA may produce quadratically many  $\varphi$ -nodes and thus cannot be used in a linear-log time algorithm. In order to maintain a linear-log time bound for CFA, I developed an alternative method for choosing join points that selects only linearly many places but is still sound for the analysis. This alternative method may be applicable in other cases as a replacement for SSA.

Another direction I plan to pursue is applying the techniques from my dissertation research to forward and backward abstract interpretation [6]. The skipping functions used to move type information efficiently from one place in the program to another provide a model for efficiently computing both forwards and backwards abstract interpretation simultaneously.

## 3 Static Analysis

My research in CFA should be understood in the broader context of static analysis and how it can allow programmers to better express, analyze, and verify properties of their programs. Programmers often design their programs with a particular abstraction of the problem domain in mind. In order to better aid the programmer and verify that the program is using the domain concepts sensibly, I want to improve the programmer's ability to instruct a static analyzer as to the abstract domains that are most relevant to the problem.

This is already done to an extent in most languages with the ability to declare new types and have the compiler verify that values are used in a type-safe way. Many techniques have been developed allowing the programmer to encode complex judgments into various type systems. My work on encoding heterogeneous generic zippers in Haskell [1] is an example of this and I hope to continue advancing these sorts of techniques. However, encoding certain properties into types can be excessively complicated. This is where allowing the programmer to declare properties of the program that should be analyzed is useful. A programming language's type system is essentially one sort of static analysis, but many properties are not easily expressed in a type system. As it stands, analyzing new program properties usually requires a significant amount of development time and theoretical knowledge. Contrast this with types, where little is required for the programmer to declare new types. Lowering the barriers to the use of static analysis in everyday programming is a major research direction I wish to pursue. This involves not only efficient analysis algorithms such as in my dissertation research but also making the definition of a new analysis expressible in terms approachable by the average programmer.

## 4 Other Research

Though my recent research focus is static analysis with an emphasis on control-flow analysis, I have been involved in research over a broad range of topics. Following are some of my past research results. 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 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 [10]. It is used in a number of applications and is widely applicable for manipulating tree-shaped data structures.

The traditional zipper suffers from two major limitations, however. First, it operates only on homogeneous types. That is to say, every node the zipper visits must have the same type. In practice, many tree-shaped types do not satisfy this condition, and thus cannot be handled by the traditional zipper. Second, the traditional zipper involves a significant amount of boilerplate code. A custom implementation must be written for each type the zipper traverses. This is error prone and must be updated whenever the type being traversed changes.

The generic zipper developed in my research overcomes these limitations [1]. It operates over any type and requires no boilerplate code to be written by the user. It does this by encoding the context portion of a zipper as an existential type using common Haskell type features.

## 4.2 Cache-Oblivious Programming

Some of my earliest research was on expressing matrix algorithms in a way that simultaneously requires a minimum effort for the programmer and delivers maximum performance from the machine [3]. For these sorts of problems, minimizing cache-miss rates is critical to optimize but also difficult for programmers to manage. Previous work on cache-oblivious programming developed methods for automatic cache management and had great success on algorithms such as the fast Fourier transform [9]. However, these techniques did not deliver performance competitive with hand-tuned libraries when translated to matrix algorithms such as Cholesky factorization [5]. My research showed that by slightly 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.

## 4.3 Empirical Evaluation of Algorithms for Equality

Other work focused on efficiently implementing the “equal?” predicate which the Revised<sup>6</sup> Report on Scheme [15] requires to terminate on all inputs including cyclic data. While “equal?” can be implemented via a DFA-equivalence or union-find algorithm, these algorithms usually require an additional pointer to be stored in each object, are not suitable for multi-threaded code due to their destructive nature, and may be unacceptably slow for the small acyclic values that are the most likely inputs to the predicate.

Along with Kent Dybvig, I developed a variant of the union-find algorithm for “equal?” that addresses these issues [2]. 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.

## 4.4 Future Research

Besides my research in CFA and static analysis, I am currently involved in ongoing research in both the areas of macros and parsing technology. I expect results from that research to be published in the next year. Research directions I hope to pursue in the future include formal methods, advanced type systems including dependent type systems, and domain specific languages.

## 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.
- [2] Michael D. Adams and R. Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 179–188, New York, NY, USA, 2008. ACM.
- [3] Michael D. Adams and David S. Wise. Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 41–50, New York, NY, USA, 2006. ACM.
- [4] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):845–868, July 1998.
- [5] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast matrix multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 13(11):1105–1123, November 2002.
- [6] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, January 1999.
- [7] 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 (TOPLAS)*, 13(4):451–490, October 1991.
- [8] R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.
- [9] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.
- [10] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.
- [11] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.
- [12] Matthew Might. Logic-flow analysis of higher-order programs. *ACM SIGPLAN Notices*, 42(1):185–198, January 2007.

- 
- [13] Matthew Might and Olin Shivers. Environment analysis via  $\Delta$ CFA. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 127–140, New York, NY, USA, 2006. ACM.
  - [14] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the  $k$ -CFA paradox: illuminating functional vs. object-oriented program analysis. *ACM SIGPLAN Notices*, 45(6):305–315, June 2010.
  - [15] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised<sup>6</sup> report on the algorithmic language Scheme, September 2007.
  - [16] David Van Horn and Harry G. Mairson. Deciding  $k$ CFA is complete for exptime. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 275–282, New York, NY, USA, 2008. ACM.
  - [17] 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.