Research Statement

Matthew Might

Fall 2007

Modern computing stands at a triple inflection point. With the cost of information *in*security spiraling, the shift to multi-core architectures now widespread, and the need for reliable software systems firmly entrenched, progress in programming languages is not a matter of convenience. It is an imperative.

Acting as polestar for my research, this imperative guides my search for solutions in the form of static analyses, linguistic constructs and program optimizations. These three pursuits reify my research philosophy—a philosophy which pushes responsibility deeper into the compiler and further away from the programmer.

Security Much of the blame for the mushrooming information security crisis is attributable to defects at the language-design level, *e.g.*, buffer overflows and injection attacks. Many of these defects could have been prevented outright (or at least ameliorated) when the programming language in use was designed. Viewing security as the combined responsibility of the compiler and the language lends itself to a two-pronged approach in my research: designing expressive, efficient, secure linguistic constructs for new languages; and building security-minded static analyses for languages with entrenched code bases.

Parallelism The trend toward multi-core architectures and away from pushing up clock rates means that the continuance of Moore's law now depends on the average programmers' ability to fully harness this newfound parallelism; this is a feat which, with present tools, seems unlikely to happen. Once more, my research takes a two-front approach: developing linguistic

constructs and paradigms that support the natural construction of either implicitly or explicitly parallel programs; and building static analyses that capture the requisite properties for automatic parallelization, *e.g.*, aliasing and dependencies.

Verification Software systems have become intertwined in every facet of modern society, and yet, when it comes to soundly predicting the behavior of software, we are toiling in the dark ages. While some progress has been made, the vast majority of software enters into use with no rigorous guarantee that it will meet a formal specification. (In fact, most software lacks even a formal specification to begin with.) My research in verification consists of transferring results from my work in higher-order flow analysis into the domain of model checking, where they can be used to verify safety and liveness properties.

Static analysis

Static analysis forms the backbone of my research into security, parallelism and verification. That a single analysis often has applications in all three areas is what makes it possible to tackle them simultaneously: I am not trying to climb three mountains; I climb only one.

In static analysis, the important metrics for the frameworks I develop are:

- *Speed*. The average time that it takes an analysis to run.
- *Precision*. The degree to which an analysis excludes false positives.
- *Power*. The kinds of questions an analysis can answer soundly.[1]

ΔCFA [10, 11], ΓCFA [9, 12, 7] and LFA [5] mark my published explorations in the space measured by these axes. My recent work extends the reach of these frameworks to cover more language features (*e.g.*, pointer arithmetic), and develops techniques to move each of these further outward on the aforementioned metrics.

[1] For example, asking the question, "Could a `String` object flow to the variable `x`?" requires a less powerful analysis than asking the question "Are all values which flow to the variable `i` in bounds for the array `a`?" In this same sense, must-alias analyses are more powerful than may-alias analyses.

ΔCFA: Extracting environment structure from the stack

ΔCFA, my first foray into static analysis, reasons about environment structure in higher-order programs [10, 11]. It was a direct response to fundamental limitations suffered by k-CFA, a framework that has been instrumental in the field of higher-order and object-oriented flow analysis for nearly two decades. Chief among these limitations is k-CFA's inability to make sound judgments about the equivalence of environments (and, less generally, objects and aliases) that arise during program execution. Shivers' seminal work on k-CFA [15] termed this the *environment problem*.

As a research community, we have been stumbling over uses for an environment solution—from Super-β inlining and rematerialization [4] to precise must-alias analysis and coroutine fusion [16]—for almost as long as k-CFA has been known. ΔCFA drew many of these optimizations and analyses into reach for the first time.

ΔCFA determines equivalence between environments by enriching procedure strings into *frame strings* and then statically bounding their shape. Whereas procedure strings capture procedure call and return events, frame strings are tuned to pick up push and pop actions on the program stack. This shift in perspective brings two immediate benefits: (1) stack motion has a consistent interpretation in places where call and return lose their meaning or fail to nest; and (2) algebraically, frame strings form a group where procedure strings formed a monoid. This shift to a group places ΔCFA atop a firm theoretical foundation, enabling a sound formalization of the deep connection between stack and environment behavior.

Next steps for ΔCFA The original ΔCFA was only an opening salvo on frame strings and environment analysis. Since ΔCFA's creation, I have extended its precision and power by generalizing the concept behind abstract garbage collection and abstract counting (covered in the next section under ΓCFA) beyond heaps and environment structure to abstract frames themselves; ultimately, this allows more of the group-theoretic structure of frame strings to be recovered and exploited during analysis [13]. At present, my work in ΔCFA focuses on transforming it into a CFL-reachability problem, which will allow both an on-demand formulation (critical for large code bases and modularity) and a reduction in analysis run-time. Meanwhile, I have found that ΔCFA's results can drive both a Harrison-style dependency analysis [2] for automatic parallelization and a generalized escape analysis

that works in the presence of full, first-class continuations. Eric Knauel at the University of Tübingen has integrated the enhanced ΔCFA into Scheme 48, and his early results with respect to the generalized escape analysis have been highly encouraging—I expect to report more on ΔCFA shortly [13].

ΓCFA: Exploiting reachability and cardinality in analysis

My next analysis, ΓCFA, strikes at the core problem with abstract interpretation: the loss in precision that comes from mapping the infinite state-space through which a computation evolves onto a smaller (often finite) abstract state-space.[2] This compaction has two effects: (1) it bounds the room through which the abstract interpretation may run, ensuring its termination; but (2) it also creates false positives by using a single abstract element to represent multiple concrete elements. ΓCFA, as a higher-order flow analysis, makes more efficient use of this abstract space by applying the abstract analog of a long-proven technique—garbage collection—to the *environment structure* of the analysis.

Experimentation reveals that abstract garbage collection leads to order-of-magnitude improvements in precision. But, surprisingly, more than precision improves: analysis run times also fall by an order of magnitude. In subsequent investigation, I found that the tightened precision (fewer false positives) leads to fewer spurious forks into impossible state-space during abstract interpretation (Figure 1, Figure 2). By avoiding these false branches, the running time of the analysis drops. For these benefits, ΓCFA has also since evolved into a system for software model checking [7].

Beyond boosting speed and precision, abstract garbage collection made a second solution to the environment problem, abstract counting, feasible. The crux of the environment problem is determining when the concrete counterparts of two abstract values are equal. Each abstract value represents a set of concrete values. Abstract counting upper-bounds the cardinality of these sets during the analysis. (The upper bounds used in ΓCFA are zero, one, and more than one counterpart.)

Abstract counting drives a simple principle to determine the equality of concrete elements (such as environments, objects and aliases) from the equality

[2]For example, imagine mapping the state-space of a Turing machine onto the states of an NFA. The resulting NFA is a conservative over-approximation of the original Turing machine, that is, an abstract interpretation.

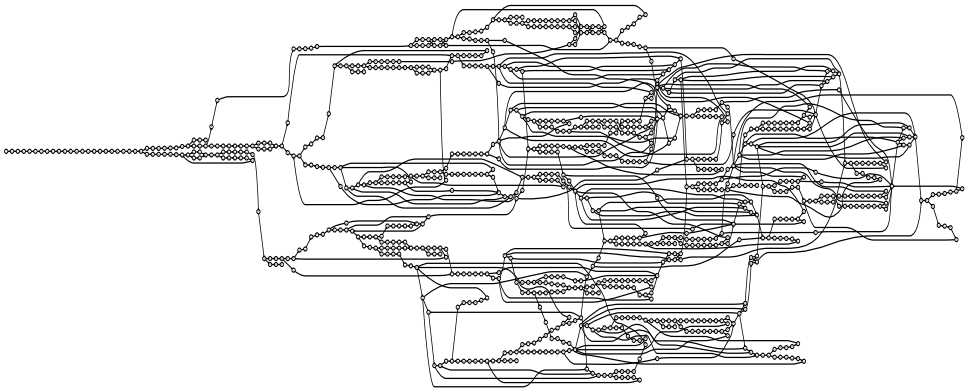

Figure 1: State graph for a doubly nested loop generated *without* abstract garbage collection.

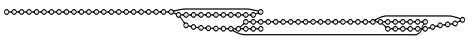

Figure 2: State graph for a doubly nested loop generated *with* abstract garbage collection.

of their abstractions: If two sets A and B are equal, *and* each set is a singleton, then any element in the set A is equal to any element of the other set B and *vice versa*.

Even without abstract garbage collection, abstract counting is still a sound enrichment of a flow analysis. However, without abstract garbage collection, the approximate cardinalities quickly spiral toward the useless "more than one." With abstract garbage collection in play, these cardinalities periodically reset to zero as resources are collected, and this often leads to the magic count of "one counterpart."

Next steps for ΓCFA Given its benefits for precision, ΓCFA has already been integrated into compilers by Cameron Zwarich at the University of Waterloo and, once again, by Eric Knauel at the University of Tübingen. Early empirical results from these efforts have been impressive both in terms of precision and in the size of the code bases analyzed (up to the largest available benchmark of 20,000 lines), but they have also revealed even more opportunities to improve speed.

In recent work with Pete Manolios at Northeastern University, I have gen-

eralized the philosophy behind abstract garbage collection into a philosophy that calls for management of abstract resources in accordance with optimizing speed and precision. Under the Cousots' thirty-year-old formulation of abstract interpretation [1], context-sensitive approaches felt like the natural barrier beyond which no analysis could remain sound. So, in order to pursue "precision-sensitive" abstract resource allocation, we first had to prove that doing so was sound. Capping our efforts to prove soundness, I discovered a completeness theorem for polyvariant strategies [8] that, against prevailing intuition, guarantees that all strategies (including non-deterministic and probabilistic ones) can be proven sound. Armed with this completeness theorem, we are now actively exploring precision-sensitive abstract interpretations, and already, I have found that a large class of existing context-sensitive analyses (chiefly, the CFA family) can be made faster at no cost to precision.

Logic-Flow Analysis: The best of both worlds

My next static analysis framework, logic-flow analysis (LFA) [5], began with the question, "What would enable a flow-analytic abstract interpretation to rule out the presence of buffer-overflow vulnerabilities in higher-order programs where the proof of safety is neither trivial nor lexically apparent?"

A flow analysis adequately handles the task of sorting out prerequisites such as aliasing for arrays and (due to my prior work) environment structure. But, a flow analysis is too coarse-grained to determine whether anything but a trivial array access is in bounds.

LFA recovers the requisite information by performing a propositional abstract interpretation concurrently. In this interpretation, each machine state abstracts to a set of propositions in first-order logic. This proposition-based interpretation is fine-grained enough to capture the relevant relationships between indices and bounds. But, by itself, it is too imprecise to reason about the requisite control- and data-flow properties.

Algorithmically, LFA is actually the reduced product of these two abstract interpretations. The end result is an analysis that is simultaneously more powerful and more precise than what either interpretation achieves individually. Critically, it is precise enough to rule out the presence of buffer overflows, even in situations as complicated as vertex arrays.[3]

[3] I spend extra attention on buffer overflow, because according to the U.S. C.E.R.T.

The propositional abstract interpretation can actually capture and verify a variety of properties beyond buffer-indexing constraints, from lock-usage invariants all the way to user-proscribed pre- and post-conditions, making it fertile ground for further investigation.

Next steps for LFA LFA was not conceived in a vacuum; it was created with security in mind, and as a result, I have worked to bring it to bear on industrial languages such as C, C++ and Java. Much of this has occurred in the context of a startup company that I co-founded, Diagis,[4] which is creating a static analyzer to perform software security audits. At present, we have a working prototype implemented for the LLVM compiler framework, a GCC-front-ended compiler system sponsored by Apple. This prototype will also be instrumental in delivering empirical validation for my ongoing research.

The academic-to-industrial adaptation process proved itself a rich source of further research problems. Naturally, we had to find a way to handle aliasing soundly in the presence of pointer arithmetic, structures and unsound type casts. To solve this problem, I drew upon the Peano-theoretic model of arithmetic to create a Peano-theoretic model of pointer arithmetic: our analyzer maintains two additional abstract heaps that dynamically construct address-adjacency information [14].

As with the artificial soundness constraints that resulted in the completeness theorem for polyvariance, we found ourselves needlessly ham-strung by the conventional correctness framework: conventional correctness forced abstract garbage collection to needlessly hemorrhage propositions each time it collected. To side-step this barrier, I created the notion of *soundness modulo congruence* [6]. Under this correctness regimen, the concrete semantics are made into an infinitary, branching non-deterministic semantics, where each state branches to every state with a congruent heap. Correctness then requires that at least one branch of this infinitary concrete semantics be simulated.

When it comes to gritty, real-world problems, clean, elegant solutions like Peano pointer arithmetic, soundness modulo congruence and the polyvari-

vulnerability database, it is the most common vulnerability, and it accounts for more than a third of the most severe vulnerabilities by itself.

[4]With myself as principal investigator, Diagis has received a $50,000 grant from the Georgia Research Alliance and a $100,000 SBIR grant from the National Science Foundation. Diagis currently has five employees.

ance completeness theorem serve as examples of what I look for.

Parallelism

To date, I have begun two concerted efforts in tackling parallelism: (1) a static analysis for recovering interprocedural dependency information in higher-order languages, and (2) a coroutine-based approach to multi-core utilization.

Dependency analysis from continuation marks

Since my work on ΔCFA, I have been conscious of the need to abstract the program stack in a precise and efficient fashion. This precise modeling of the stack can, as a side benefit, capture the interprocedural dependencies that must be known in order to perform coarse-grained automatic parallelization. It is roughly true that whatever procedures have frames live on the stack when a side-effectable resource is read or written have a dependency on that resource. (It is roughly false in that this naïve view does not account for tail-call optimization.) I recently adapted my analytic framework to produce dependency information from its continuation-based modeling of the stack [3]. By using continuation marks, this framework can also still exploit proper tail-call optimization, which ends up improving speed and precision in abstract interpretations. An implementation of this technique is underway, and I expect to report that the same order-of-magnitude boost to precision that abstract garbage collection brought to other analyses will be replicated for interprocedural dependency analysis.

Coroutine fusion

With regard to linguistic constructs for parallelism, my research efforts to date have centered on the coroutine. A coroutine-based program is assembled as a (potentially non-linear) pipeline of communicating processes. UNIX users make use of this paradigm every day when issuing commands such as `find . | grep foo | wc -l`.

While coroutines are not new, programmers have shied away from them in industrial contexts to avoid the penalty of a per-coroutine program stack and

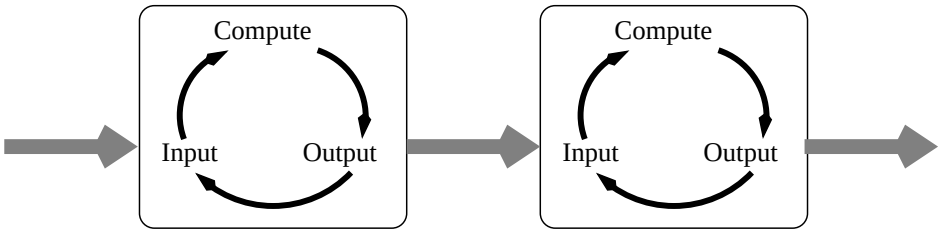

Figure 3: Before coroutine fusion.

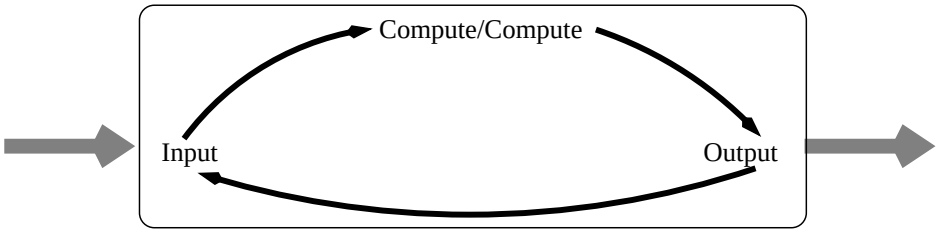

Figure 4: After coroutine fusion.

an inter-coroutine communication overhead. However, for certain software systems, *e.g.*, graphics pipelines and network protocol stacks, coroutines represent a compelling design abstraction over the tedious and error-prone hand-merging of layer upon layer.

Coroutine fusion [16] is a program optimization that melts two communicating processes (Figure 3) into a single process (Figure 4), thereby eliminating both the stack and communication overhead. By reasoning about and optimizing data-flow, coroutine fusion can turn a clean but inefficient "many-copy" architecture into a zero-copy system for free.

The beauty of the coroutine fusion optimization lies not in its complexity, but rather in its uncompromising simple-mindedness: turning *everything* into λ. Coroutine fusion builds inter-process communication channels out of continuations, and then uses continuation-passing-style conversion to turn continuations into λ terms. The resulting program uses a single construct to encode every control transfer mechanism—from procedure call and return to exception throw and coroutine switch—call to λ.

Once in this form, the program is subjected to battery of optimizations for

the λ calculus, most of which have been known since the 1930s. Once these classical optimizations drive the program to a fixed point, only ΔCFA or ΓCFA can enable the crucial, environment-sensitive Super-β class of optimizations, whose effect is to melt the processes together.

With access to coroutine fusion, parallelization for programs written in this paradigm becomes a four-step process: (1) Profile the program. (2) Tile the coroutine network with coverings equal to the number of processors. (3) Fuse each covering into a single coroutine. (4) And, run each fused coroutine on its own processor, with communication buffers between processes.

What else is next?

At present, the challenges and opportunities stacked up in my research pipeline should keep me occupied for years to come. Below, you'll find a sampling of these opportunities. Taken together with my previous work, they paint a picture of where my long-term research arc is taking me next.

ΘCFA LFA is powerful, but it is also expensive. Fortunately, my recent work indicates that it is far stronger than what one needs for ruling out buffer overflows for common situations. Consequently, my efforts are geared toward distilling from LFA only the essential properties required for this task. So far, this distillation hints at a fusion of ΓCFA with a generalization of Wand and Steckler's invariance sets [17]. What I view as an ancestor to ΘCFA now appears in my ongoing work on soundness modulo congruence [6]. Once completed, ΘCFA will create another point in the speed-precision-power landscape: more powerful than ΓCFA, faster than LFA, and just right for removing buffer overflows.

Unifying abstract interpretation and constraint-solving for CFA Mitch Wand at Northeastern University and I share an interest in unifying constraint-solving and the abstract interpretation approaches to higher-order control-flow analysis. Each approach possesses advantages—speed and simplicity for constraint-solving, and power and flexibility for abstract interpretation. A means of inter-converting these approaches should light the way for exchanging these advantages as well. With Mitch starting from constraint-solving and approaching abstract interpretation, and myself start-

ing from abstract interpretation and working toward constraint-solving, we hope to meet in the middle.

Quantum computation and the λ calculus While I expect most of my work to be centered in programming languages, I do reserve a small portion of my time for investigations into neighboring fields. Sparked by the constraints imposed in quantum computation, my interests there involve the development of a typed quantum λ calculus, and the exploitation of continuations to provide a clean, efficient framework for reversible computing.

For the typed λ calculus, my work is looking for a twist on the traditional type-safety guarantee. Instead of "No well-typed program has an error," the type-safety theorem for this language adds another constraint: "No well-typed program *attempts* to do the physically impossible." Van Tonder's recent work [18] pointed at the importance of using linearity for quantum resources. My investigations suggest that only a combination of linear type systems and dependent type systems can make such a guarantee while remaining universal for quantum computation.

The formalization of frame strings as a group in ΔCFA hinted at another route to explore: exploiting the continuations flowing through CPS not only as the arbiters of control, but also as the arbiters of reversibility. The effect of applying a continuation, in frame string terms, is to add the group-theoretic inverse of the frame string spanning from some point in the past to the present; the evolution of a quantum computation looks like the successive application of unitary transformations, so continuation application could be modeled as applying the sequence of inverse transformations described by the frame string. Because of the frequent opportunities for reversing computation, this could lead to the development of a qubit-efficient quantum compiler.

The opportunities outlined above are just a slice of the challenges ahead. The broad applicability of static analysis makes me optimistic that my work going forward will be done in concert with colleagues, as we explore and uncover the space that lies between our fields.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.

[2] L. Harrison. "The Interprocedural Analysis and Automatic Parallelization of Scheme Programs." *Lisp and Symbolic Computation: An International Journal*, Vol. 2, No. 3/4, pp. 179-396, 1989.

[3] Matthew Might. "Automatic parallelization from dependency analysis of continuation marks." Unpublished. Available from http://matt.might.net/papers/unpublished/.

[4] Matthew Might. "Environment Analysis of Higher-Order Languages." Ph.D. dissertation, Georgia Institute of Technology, August 2007.

[5] Matthew Might. "Logic-flow analysis of higher-order programs." *Proceedings of the 34th Annual ACM Symposium on the Principles of Programming Languages* (POPL 2007). Long paper category. Nice, France. January, 2007. pages 185–198.

[6] Matthew Might. "Abstract ignorance is bliss." Unpublished. Available from http://matt.might.net/papers/unpublished/.

[7] Matthew Might, Benjamin Chambers and Olin Shivers. "Model Checking via ΓCFA." *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation* (VMCAI 2007). Nice, France. January, 2006.

[8] Matthew Might and Panagiotis Manolios. "The polyvariance completeness theorem: Enabling precision-sensitive abstract interpretation." *Submitted to PLDI 2008.* Available from http://matt.might.net/papers/unpublished/.

[9] Matthew Might and Olin Shivers. "Improving flow analyses via ΓCFA: Abstract garbage collection and counting." *Proceedings of the 11th ACM International Conference on Functional Programming* (ICFP 2006). Portland, Oregon. September, 2006. pages 13–25.

[10] Matthew Might and Olin Shivers. "Environment analysis via ΔCFA." *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages* (POPL 2006). Charleston, South Carolina. January, 2006. pages 127–140.

[11] Matthew Might and Olin Shivers. "Analyzing environment structure of higher-order languages using frame strings." *Journal of Theoretical Computer Science*. 2007. To appear.

[12] Matthew Might and Olin Shivers. "Exploiting reachability and cardinality in higher-order flow analysis." *Journal of Functional Programming*. 2007. To appear.

[13] Matthew Might, Olin Shivers and Eric Knauel. "Generalizing escape analysis in the presence of continuations." Unpublished. Available from http://matt.might.net/papers/unpublished/.

[14] Matthew Might, Olin Shivers, Benjamin Chambers and T. Stephen Strickland. "Abstract interpretation of imperative programs via garbage-collectable arithmetic." Unpublished. Available from http://matt.might.net/papers/unpublished/.

[15] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. dissertation, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145, School of Computer Science.

[16] Olin Shivers and Matthew Might. "Continuations and transducer composition." *Proceedings of the 27th Conference on Programming Language Design and Implementation* (PLDI 2006). Ottawa, Canada. pages 295–307. June, 2006.

[17] Mitchell Wand and Paul Steckler. "Selective and Lightweight Closure Conversion." *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages* (POPL 1994). Portland, Oregon. January, 1994. pages 435-445.

[18] André van Tonder. "A Lambda Calculus for Quantum Computation." *SIAM Journal on Computing*. 33 (5) pages 1109-1135.