Research Statement

Frederick Smith

August 8, 2001

My goal is to simplify and speed the development of complex software systems. To that end, I have focused on developing tools that detect run time errors at compile time while allowing programmers to express efficient programs concisely. Specifically, my thesis work combines run-time code generation (an optimization for improving speed) with certifying compilation (a technique for checking at compile time that the resulting code will not crash). In a separate project, Kathleen Fisher, Anne Rogers, and I developed Hancock, a domain-specific language for processing large volumes of data that makes call-processing programs easier to maintain and more efficient. Hancock is used daily within AT&T. Finally, I devised the mostly-copying garbage collection algorithm to improve the efficiency of automatic memory management for garbage-collected programs inter-operating with those using manual memory management (such as most C programs). That algorithm and my implementation (MCC) serve as the basis for the Moby compiler's garbage collector.

Certifying run-time code generation. Portable executable formats, such as Java bytecode, rely on just-in-time compilers to regain efficiency. Transmeta relies on binary translation to convert x86 instructions into Transmeta's native format. This choice allows Transmeta engineers to use the latest hardware technology with only the marginal cost of updating their binary translator. Both of these paradigms rely on aggressive run-time code generation (RTCG) to recover performance. Both require that the dynamically generated code be as reliable as hardware. Reliability can be increased by checking that generated code is type-safe (cannot crash). Typed Assembly Language (TAL) can be used to check that the output of a certifying compiler is type-safe. My research extends this guarantee to programs using RTCG.

Our RTCG system copies code fragments, called templates, sequentially into a code region until a complete well-typed function is produced. At each intermediate step the type of the code region has a post-condition stating its requirements of the next template. A fundamental problem we faced was handling the changing type of the code region.

Traditional type systems assign one type to each location and therefore cannot type-check code regions. Linear type systems permit types to change but only for locations that are not aliased. Modern optimizing compilers rely on register allocation. One effect of register allocation is to create aliases between stack locations and registers, thus violating the key property of linear type systems. David Walker and I developed alias types to solve this problem. Alias types track aliasing explicitly by introducing a level of indirection in the type system. Using this level of indirection, alias types easily accomodate type-changing operations such as code generation and memory deallocation. The formal methods we used to prove alias types sound increased our confidence in the correctness of the TAL implementation.

To evaluate our technology, Dan Grossman and I developed an optimizing compiler for Cyclone, a type-safe C-like language augmented with RTCG constructs. Our compiler translates Cyclone into TAL/T, TAL augmented with RTCG constructs. Because code generation is part of the program's execution time, it is essential to generate dynamic code quickly and effectively. To minimize code generation time, our compiler performs most optimizations (for example, register allocation) at compile time and only assembles pre-computed code templates at run time. Our experiments show that our type-safe system is able to achieve speedups comparable to those achieved by similar un-safe systems.

Most recently, I have worked on evaluating the effectiveness of RTCG and our compiler. The results have been surprising. For a ray-tracer application (based on the winning entry of the ICFP 2000 programming contest) we found little or no benefit. Although seemingly a good candidate for RTCG because the application contains both an interpreter and many invariant matrix-vector multiplies, the benefits of RTCG are negligible because these elements do not account for a significant fraction of the overall running time. On the other hand, I ported the functional simulator from the SimpleScalar Toolset, a widely used simulator in the hardware community, and ran the SPEC CPU95 benchmark suite through it. RTCG was able to eliminate the fetch and decode stages of the instruction pipeline for the simulator. Because these stages are relatively expensive, we obtained an average 2.7 times speedup.

Hancock. AT&T's statisticians are interested in studying telephone call records to detect fraud. Initial trial applications showed that this approach was feasible but the programs were hard to write at the scale of the incoming data (250 million calls per day). The programs were also hard to maintain as native C constructs do not map cleanly onto the abstractions used by the statisticians. Anne Rogers, Kathleen Fisher, and I designed and implemented the domain-specific language Hancock to address these two concerns through the use of special-purpose language constructs and a high-performance database-like run-time system. To ease the migration from C, we designed Hancock to be a strict superset of C. Kathleen Fisher and I implemented the Hancock compiler as a Hancock to C translator. I worked on Hancock through the first public release. Today, you may download Hancock for non-commercial use from the internet. AT&T's production machines run Hancock compiled programs. And, statisticians within AT&T use Hancock to develop new applications.

Mostly-copying garbage collection. Garbage collectors automatically reclaim unusable memory. A garbage collector works by tracing pointers in the heap and reclaiming any data not reachable from static data, the stack or registers. Most type safe language implementations rely on a garbage collector for all memory deallocation. Most garbage collectors are either accurate or conservative. Accurate collectors rely on explicit tags (usually compiler inserted) to detect pointers. Conservative collectors do not use such tags but are usually slower and less precise (they may reclaim less memory) than accurate collectors.

In the process of producing a C backend for the ML compiler TIL, Greg Morrisett and I had to choose a garbage collector. Although our compiler could tag objects in the heap, the stack layout was under the C compiler's control and could not be tagged precisely. Therefore we could not use an accurate collector. At first we used the Boehm-Demers-Weiser conservative collector for C (BDW) but found it to be much slower than the accurate collector we used previously..

Instead we developed and implemented a collection algorithm that we later discovered was an enhanced version of the mostly-copying collection algorithm. Our collector, MCC, uses a hybrid approach. It behaves like an accurate copying collector when possible, and like a conservative mark-sweep collector when necessary. The result is a garbage collector that allows easy inter-operability with C and has good performance. One contribution of our work was a careful analysis that showed MCC outperformed BDW when memory was plentiful but not when space was at a premium. MCC serves as the basis for the Moby compiler's garbage collector. Moby is developed by Kathleen Fisher and John Reppy.

Conclusion. My research has consistently reinforced the importance of four skills. Formal methods help validate claims of correctness and better understand language design principles. Implementation skills are useful for demonstrating practical utility. Analytical and experimental skills are necessary to assess performance. And, collaboration helps provide new perspectives and techniques. For me, research combining these four skills is the most satisfying and the most effective for achieving my goal – simplifying the production of complex software.

In the near term, I intend to investigate techniques for helping programmers express performance intentions. Strong interfaces are a desirable software engineering property but most compilers impose implicit performance penalties on their use. For example, constants are usually not propagated across module boundaries. By ignoring the programmer's performance intention, compilers effectively penalize good software engineering practice. One solution to this problem is to rely on compiler optimizations. From the programmer's perspective this technique is inadequate because it provides no guarantee that the optimization will be performed. A better solution is to make it possible to express performance intentions directly in the language. More generally, we should provide compile-time support for computation. C++ templates inadvertently provide a clumsy way of performing compile-time computation and have been used to great effect for this purpose by matrix libraries.