Research Statement

Jeremy Condit
jcondit@cs.berkeley.edu

My research interests lie at the intersection of *programming languages* and *systems*, an area that has been the subject of many fruitful research projects in recent years. Developers of languages and language tools now have the computational power to analyze interesting properties of real code; meanwhile, system designers look to languages as a means to improve the reliability and security of their code. I am interested in new language techniques and tools that can be usefully applied to real-world systems.

In particular, my research focuses on improving the state of *low-level programming*. Whereas high-level languages avoid memory errors through bounds checking and garbage collection, low-level languages currently sacrifice such guarantees in the name of efficiency and expressiveness. My research focuses on improving the safety and reliability of low-level code without sacrificing its expressive power.

Current Research

The subject of my dissertation research is Deputy [1], a tool that allows programmers to annotate C code using *dependent types*, which are types whose meaning depends on the run-time value of a program expression. Although such types are difficult to reason about at compile time, they are very useful for expressing relationships between variables; for example, they can be used to indicate the relationship between an array and its length or a discriminated union and its tag. Reasoning about such relationships is crucial for understanding low-level code, which makes dependent types a powerful tool in this domain.

From a language standpoint, Deputy offers three main contributions. First, Deputy's dependent type system supports *mutable data* in a more flexible way than previous dependent type systems, which is essential for handling low-level code. Second, Deputy's dependent types use *run-time checks* in order to avoid the difficulties of reasoning about values of program expressions at compile time. Third, Deputy provides *automatic dependencies*, a technique that reduces the annotation burden on the programmer for local variables.

From a systems standpoint, Deputy shows that dependent types provide an effective way to annotate and reason about *real-world code*. In particular, Deputy's types allow programmers to express many common idioms for specifying pointer bounds in C programs, and they also allow us to reason about null-terminated strings and tagged unions. Because Deputy allows programmers to annotate existing data structures without changing them, Deputy preserves the binary interface between each compilation unit. This feature is of critical importance when dealing with large software systems.

Deputy has been used to detect faults in Linux device drivers as part of the SafeDrive project [5]. Deputy provides isolation for device drivers at a fraction of the cost of previous approaches, both in terms of performance and system complexity; similar results should apply to other extensible systems such as the Apache web server. We are in the process of applying Deputy to the Linux kernel itself.

Prior to Deputy, I worked on two additional research projects. The first is Capriccio, a highly scalable thread package for use in Internet servers [4]. At the time, most research suggested that event-based servers were necessary for achieving high performance. Instead, we observed an important duality between the event-based and thread-based programming models, which implied that thread-based servers can achieve performance equivalent to event-based servers (and vice versa). Capriccio is a proof of concept for this notion: it is designed to provide all of the benefits of an event-based system in the context of a threaded programming model. A key component of this work is a language-based technique for limiting the size of each thread's stack while preserving the abstraction of an unbounded stack.

The second project, data slicing, is a type-directed program transformation that splits heap-based data structures into separate regions of memory [2]. This technique was developed for use in CCured [3], which is a tool for enforcing memory safety in C programs. Data slicing allows CCured to separate its metadata from the original program's data structures, thus preserving the original program's data layout. Although Deputy's dependent type annotations have now eliminated the need for such metadata, this technique was very helpful in handling CCured's library compatibility problems. This technique can also be applied to similar problems arising from several other compiler transformations.

Future Research

My long-term research goal is to make low-level programming as safe and reliable as higher-level application programming—if not more so! This problem is challenging because low-level programming languages must avoid imposing any unnecessary constraints on the generated code; for example, data layout and memory allocation should be under the control of the programmer. At the same time, improving the safety of such systems is of the utmost importance, since low-level code drives much of the basic computing infrastructure that we rely on every day.

Deputy addresses many common type-safety problems in C programs. However, there are a number of additional challenges that I would like to address in future research, including concurrency, memory management, specification inference, and user interface design for low-level programming.

First, low-level programming must be able to reason effectively about *concurrency*, particularly given the recent popularity of multi-core processors. Static analysis can make many important contributions in this area; for example, if programmers specify which data is owned by a particular thread and how access to shared data is synchronized, we can verify correct use of this data. Using dependent types for these specifications may help overcome some hurdles that have been encountered by such efforts in the past.

Second, safe and expressive *memory management* is critical for low-level programs. Whereas higher-level applications can rely on garbage collection for safe memory management, low-level programs often need to manage memory themselves. Although many solutions have been proposed, including linear types, regions, and reference counting, none of these solutions is effective in isolation. However, I believe that a careful combination of these techniques may prove more effective than any one of them alone.

Third, *specification inference* is a key component of many of the above challenges. In order for the compiler to effectively verify pointer bounds, concurrency, or memory management, we must allow the programmer to write down detailed program invariants. However, in order to limit the cost of the transition to such a system, we must be able to infer as many of these invariants as possible. Even if the inferred invariants are not always correct, they can provide a useful foundation for the programmer.

Fourth, providing a useful *user interface* for the programmer is a significant upcoming challenge. Textual program representations are very limited in the amount of information they can present without overwhelming the user. I plan to explore more visually-oriented program representations that can show or hide annotations or analysis results according to the programmer's needs. I also plan to explore visual representations of data structures and their invariants. Such a system could allow the programmer to interact with the compiler in a more meaningful way and to express program invariants in a much more intuitive fashion.

Ultimately, I plan to incorporate these ideas into a new language for low-level programming. Much of my research has focused on retrofitting and analyzing existing C code; however, I believe that the best long-term solution to the problems of C is a new language designed from the ground up. As I pursue the research objectives outlined above, I plan to integrate them where possible into a simple and cohesive systems programming language—one that could ultimately replace C and eliminate many of these problems for good.

References

[1] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-Level Programming. In *European Symposium on Programming (ESOP)*, 2007.

[2] Jeremy Condit and George C. Necula. Data Slicing: Separating the Heap into Independent Regions. In *International Conference on Compiler Construction (CC)*, 2005.

[3] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), May 2005.

[4] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *Symposium on Operating System Principles (SOSP)*, 2003.

[5] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Operating System Design and Implementation (OSDI)*, 2006.