Type Checking Systems Code

Greg Morrisett

Cornell University

Abstract. Our critical computing systems are coded in low-level, type-unsafe languages such as C, and it is unlikely that they will be re-coded in a high-level, type-safe language such as Java. This invited talk discusses some approaches that show promise in achieving type safety for legacy C code.

1 Motivation

Our society is increasingly dependent upon its computing and communications infrastructure. That infrastructure includes the operating systems, device drivers, libraries and applications that we use on our desktops, as well as the file servers, databases, web servers, and switches that we use to store and communicate data. Today, that infrastructure is built using unsafe, error-prone languages such as C or C++ where buffer overruns, format string errors, and space leaks are not only possible, but frighteningly common.

In contrast, type-safe languages, such as Java, Scheme, and ML, ensure that such errors either cannot happen (through static type-checking and automatic memory management) or are at least caught at the point of failure (through dynamic type and bound checks.) This fail-stop guarantee is not a total solution, but it does isolate the effects of failures, facilitates testing and determination of the true source of failures, and enables tools and methodologies for achieving greater levels of assurance. Therefore, the obvious question is:

Why don't we re-code our infrastructure using type-safe languages?

Though such a technical solution looks good on paper and is ultimately the "right thing", there are a number of economic and practical issues that prevent it from happening.

First, our infrastructure is *large*. Today's operating systems consist of tens of millions of lines of code. Throwing away all of that C code and reimplementing it in, say Java, is simply too expensive, just as throwing out old Cobol code was too difficult for Year 2000 bugs.

Second, though C and C++ have many faults, they also have some virtues—especially when it comes to building the low-level pieces of infrastructure. In particular, C provides a great deal of transparency and control over data representations which is precisely what is needed to build services such as memory-mapped device drivers, page-table routines, communication buffer management, real-time schedulers, and garbage collectors. It is difficult if not impossible to realize these services in today's type-safe languages simply because they force one to buy into the "high-level language assumption" in order to realize the benefits of type-safety.

2 Alternatives to High-Level Languages

An alternative to re-coding our infrastructure in a safe language is to change C and C++ compilers to do more run-time checking to ensure fail-stop behavior. For instance, the CCured project [6] at Berkeley has recently shown how to automatically compile C code so that we can ensure memory safety. The approach is based on dynamically tracking the types and sizes of the sequence of elements that a pointer might reference, and ensuring that upon dereference, a pointer is valid. In this respect, CCured is much like Scheme because it relies upon dynamic type tags, dynamic type tests, and a (conservative) garbage collector to recycle memory. And, like a good Scheme compiler, CCured attempts to minimize the dynamic tests and type tags that are necessary to ensure memory safety. Consequently, the resulting code tends to run with relatively little overhead (around 50%), especially when compared to previous research [2, 1] or commercial tools such as Purify where overheads of 10x are not uncommon.

But just as with Scheme or any other high-level language, CCured is less than ideal for building low-level infrastructure because control over data representations and memory management have been taken from the programmer. Depending upon the analysis performed by the CCured compiler, a pointer value may take up one word or two, and data values may be augmented with type tags and array bounds. The programmer has no idea what her data look like and thus interfacing to legacy code requires the wrappers and marshallers of a traditional foreign function interface. Garbage collection may introduce long pauses and space overheads. And finally, errors are detected at run-time instead of compile time. Nonetheless, CCured shows that we can achieve fail-stop behavior in a completely automatic fashion for legacy infrastructure that *should* be written in a higher-level language.

Another approach is to throw static analysis at the problem. However, there are serious tradeoffs in statically analyzing large systems and most current analyses fail in one respect or another. A critical issue is minimizing false positives (i.e., reporting a potential problem when there is none) else programmers will not use the tools. One way to achieve this is to sacrifice soundness (i.e., fail to report some bugs) by choosing careful abstractions that make it easier to find common mistakes. For example, Engler has recently used very simple flow analyses to catch bugs in operating systems [5]. The flow analysis is unsound because, for instance, it does not accurately track alias relationships. Though there is much merit in tools that identify the *presence* of bugs, in contexts where security is a concern, we need assurance of the *absence* of bugs. Otherwise, an attacker will simply exploit the bugs that the tools do not find. At a minimum, we ought to ensure fail-stop behavior so as to contain the damage.

An alternative way to minimize false positives in analysis is to increase the accuracy. Such accuracy often requires global, context-sensitive, flow-sensitive analyses that are difficult to scale to millions of lines of code. Few of these analyses work at all in the presence of features such as threads, asynchronous interrupts, or dynamic linking—features that are crucial for building modern systems.

3 Cyclone

Porting code to high-level languages, using compilers that automatically insert dynamic checks, and using tools to statically analyze programs for bugs each have their drawbacks and merits. The ideal solution is one that combines their benefits and minimizes their drawbacks. In particular, the ideal solution should:

- catch most errors at compile time,
- give a fail-stop guarantee at run time, and
- scale to millions of lines of code

while simultaneously:

- minimizing the cost of porting the code from C/C++,
- interoperating with legacy code,
- giving programmers control over low-level details needed to build systems.

For the past two years, Trevor Jim of AT&T and my group at Cornell have been working towards such a solution in the context of a project called Cyclone [4]. Cyclone is a type-safe programming language that can be roughly characterized as a "superset of a subset of C." The type system of Cyclone accepts many C functions without change, and uses the same data representations and calling conventions as C for a given type constructor. The type system of Cyclone also rejects many C programs to ensure safety. For instance, it rejects programs that perform (potentially) unsafe casts, that use unions of incompatible types, that (might) fail to initialize a location before using it, that use certain forms of pointer arithmetic, or that attempt to do certain forms of memory management.

Of course, once you rule out these potential errors, you are left with an essentially useless subset of the language. Therefore, we augmented the language with new type constructors and new terms adapted from high-level languages. For instance, Cyclone provides support for parametric polymorphism, subtyping, and tagged unions so that programmers can code around unsafe casts. We use a combination of default type parameters and local type inference to minimize the changes needed to use these features effectively. The treatment of polymorphism is particularly novel because, unlike C++, we achieve separate compilation of generic definitions from their uses. To achieve this, we must place restrictions on type abstraction that correspond to machine-level polymorphism. These restrictions are realized by a novel kind system which distinguishes those types whose values may be manipulated parametrically, while retaining control over data representations and calling conventions.

Cyclone also supports a number of different pointer types that are similar to those used in the internal language of CCured. These pointer types can be used to tradeoff flexibility (e.g., arbitrary pointer arithmetic) with the need for bounds information and/or run-time tests. The difference with CCured is that Cyclone requires the programmer to make the choice of representation explicit. This is crucial for building systems that must interoperate with legacy code and to achieve separate compilation in dynamically linked settings. A combination of overloading, subtyping, and local type inference helps to minimize the programmer's burden.

Cyclone also incorporates a region type system in the style of Tofte and Talpin [8, 7, 3]. The region type system is used to track the lifetimes of objects and ensure that dangling pointers to stack allocated objects are not dereferenced. The region type system can also be used with arena-style allocators to give the programmer real-time control over heap-allocated storage. Finally, programmers can optionally use a conservative collector if they are uninterested in the details of managing memory.

The Cyclone region system is particularly novel in that it provides a smooth integration of stack allocation, arena allocation, and garbage-collected heap allocation. The support for region polymorphism and region subtyping ensures that library routines can safely manipulate pointers regardless of the kind of object they reference. Finally, we use a novel combination of default regions and effects on function prototypes, combined with local inference to minimize the burden of porting C code.

All of the analyses used by Cyclone are local (i.e., intra-procedural) so that we can ensure scalability and separate compilation. The analyses have also been carefully constructed to avoid unsoundness in the presence of threads. The price paid is that programmers must sometimes change type definitions or prototypes of functions, and occasionally rewrite code.

4 Status and Future Work

Currently, we find that programmers must touch about 10% of the code when porting from C to Cyclone. Most of the changes involve choosing pointer representations and only a very few involve region annotations (around 0.6 % of the total changes.) In the future, we hope to minimize this burden by providing a porting tool which, like CCured, uses a more global analysis to guess the appropriate representation but unlike CCured, produces a residual program that can be modified and maintained by the programmer so that they retain control over representations.

The performance overhead of the dynamic checks depends upon the application. For systems applications, such as a simple web server, we see no overhead at all. This is not surprising as these applications tend to be I/O-bound. For scientific applications, we see a much larger overhead (around 5x for a naive port, and 3x with an experienced programmer). We belive much of this overhead is due to bounds and null pointer checks on array access. We have incorporated a simple, intra-procedural analysis to eliminate many of those checks and indeed, this results in a marked improvement. However, some of the overhead is also due to the use of "fat pointers" and the fact that GCC does not always optimize struct manipulation. By unboxing the structs into variables, we may find a marked improvement.

We are currently working to expand the region type and effects system to support (a) early reclamation of regions and (b) first-class regions in a style similar to what Walker and Watkins suggest [9]. We are also working on limited support for dependent types in the style of Hongwei Xi's Xanadu [10] so that programmers may better control the placement of bounds information or dynamic type tags.

References

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. Technical Report 1197, University of Wisconsin - Madison, December 1993.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[3] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg Beach, FL, January 1996.

[4] *Cyclone User's Manual*, 2001. http://www.cs.cornell.edu/projects/cyclone/.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[6] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, January 2002.

[7] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[8] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[9] David Walker and Kevin Watkins. On regions and linear types. In *ACM International Conference on Functional Programming*, pages 181–192, September 2001.

[10] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, June 2000.