Thirty Years is Long Enough: Getting Beyond C

Eric Brewer Jeremy Condit Bill McCloskey Feng Zhou

Computer Science Division, University of California at Berkeley
{brewer,jcondit,billm,zf}@cs.berkeley.edu

Abstract

Thirty years after its creation, C remains one of the most widely used systems programming languages. Unfortunately, the power of C has become a liability for large systems projects, which are now focusing on security and reliability. Modern languages and static analyses provide an opportunity to improve the quality of systems software, and yet adoption of these tools has been slow.

To address this problem, we propose a new language called Ivy that has an evolutionary path from C. The mechanism for this evolutionary path is a system of *extensions* and *refactorings*: extensions augment the language with new features, and refactorings assist the programmer in updating their code to use these new features. Extensions and refactorings have a wide variety of applications, from enforcing memory safety to detecting user/kernel pointer errors. We also demonstrate Macroscope, a tool we have built to enable refactoring of existing C code.

1 Introduction

Since the time of their creation, the relationship between Unix and C has been symbiotic: C matured because of its link to Unix, and Unix flourished because C was a quantum leap beyond its predecessor, assembly language. Thirty years after its creation, C is now deeply entrenched in the operating system community—but it is showing its age. We believe that good languages lead to good systems; thus, it is time for new language technology to drive new systems research. Unfortunately, rescuing existing systems from the perils of C is no mean feat.

One possible approach to improving language technology for systems is to focus on an entirely new language. Modern languages such as Java and ML provide stronger static guarantees, such as type and memory safety, at a slight cost in expressiveness. This trade-off may be desirable for some systems, which emphasize reliability, security, and availability over raw performance. However, these languages lack a number of useful features of C, such as manual memory management and bit-level data layout. Also, it is impractical to rewrite existing systems in an entirely new language—with millions of lines of C code running critical infrastructure, we cannot afford to simply start over.

A second possible approach to this problem is to use static analysis to root out software problems. The benefit of analysis is that it finds bugs without requiring code to be rewritten in a new language or a new model. However, static analysis tools are difficult to write and often difficult to use. Since C imposes no restrictions on where and when programs can write to memory, tools must make very conservative assumptions about program behavior, or else pay a huge cost in the complexity of the analysis. And because all analyses are conservative in some way, they usually yield large numbers of false positives, which make real bugs more difficult to detect. These false positives, combined with long analysis times, make it difficult to integrate static analysis directly into the build process of a program, which in turn hinders the ability of these tools to have a lasting impact on source code quality.

We propose a third approach that offers an *evolutionary path* from C to a new language called Ivy. This approach incorporates the advantages of both of the previous ones. First, Ivy is a programming language as opposed to an analysis tool; it will provide sound guarantees to programmers using a checker that will be integrated into the compiler. Second, Ivy will provide a transition path from existing code by means of *extensions* and *refactorings*. Extensions will add new language features such as sophisticated data layout, concurrency control, and memory management, each of which can be enabled or disabled individually. Extensions may add language features, but they may also disable them. For example, the memory safety extension will forbid some uses of casting and pointer arithmetic while adding mechanisms such as regions and built-in reference counting. Refactorings will assist programmers in the transition by analyzing existing code to find patterns that could be better expressed with a specific language extension. Working in tandem, extensions and refactorings will en-

able a transition to modern language features; indeed, they will also allow future evolution as new language technology emerges.

This paper presents our vision for the future of systems programming. First, we discuss the problems of C, and we describe a number of features that we would like to see in a new systems programming language (Section 2). Then, we discuss in more detail our evolutionary path toward this new language (Section 3) as well as a few examples of extensions and refactorings (Section 4). Finally, we present our initial results, which demonstrate that it is possible to migrate code to a more solid foundation and to apply useful refactorings (Section 5) with modest programmer effort.

2 Requirements for a Replacement

C succeeded for many years because systems written in C were safer, more portable, and more maintainable than those written in assembly. Equally important, C programs performed nearly as well as their assembly counterparts. But as time has passed, new languages have picked up the standards of safety and reliability, while C has not progressed. The most obvious gap has been in memory safety, where languages like Java and ML provide much stronger guarantees than C. Less obviously, but just as important, C fails to provide the programmer with tools for concurrency control, safe data layout, and other system-specific tasks.

In this section, we discuss some of the key features that we would like to see in a successor to C. We believe that these changes will have a positive impact on the safety and security of systems programs.

Type and memory safety. Memory safety is a crucial property for safe and secure systems. The Cyclone language [10] permits programs to use a number of safe, flexible memory management policies, such as region-based memory management, reference counting, and garbage collection, all within the context of a C-like language. Also, the CCured tool [3] analyzes pointer usage to introduce efficient run-time memory safety checks. These tools demonstrate that memory safety is a reasonable goal in a C-like language.

Besides catching bugs, type safety makes other analyses easier to write. In a type safe language, two memory locations cannot be aliased if their types differ. C lacks this property, making it more difficult to develop tools. Memory management disciplines like regions also make analysis easier, since they refine the type system further, reducing possible aliasing relationships. In general, we believe that increased memory safety will have the additional benefit of making programs easier to analyze.

Concurrency. A modern systems programming language must have native support for concurrency for a number of reasons. First of all, integrating threads and atomic sections [6] into the language makes it easier for programmers to write safe and portable code. Indeed, Boehm has shown that implementing threads without some compiler support is unsafe [2]. Secondly, built-in support for concurrency makes it easier for the compiler to check properties of concurrent programs. Most tools have difficulty processing concurrent software, since they fail to take into account all possible thread interleavings; however, code written using atomic sections should be easier to analyze, since the interactions between threads are spelled out explicitly. The Calvin-R checker makes use of atomicity in this way [8].

API adherence. Systems software often must comply with complex interfaces. Tools such as the metacompilation (MC) system [9], SLAM [1], and ESP [4] ensure that code adheres to a given interface. Unfortunately, these tools have difficulty with pointers and aliasing. We believe that some of these problems can be eliminated with the introduction of stronger type systems and memory management disciplines in the language, as mentioned above.

Data layout. Modern languages that strictly enforce type safety, such as Java and ML, almost always require data to be formatted according to conventions specified by the language or the compiler. C allows *a priori* data layout, where the programmer can control data formatting down to the bit level, which is important for systems applications that must be compatible with existing libraries, file formats, or network protocols. Unfortunately, data layout in C is not safe, so we intend to supply a mechanism to allow type-safe *a priori* data layout using a dependent type system [14]. Dependent type systems have been studied in the context of functional languages, but new research is necessary to make them work in an imperative language like C.

3 The Ivy Platform

In general, new technologies often fail for two very important reasons. The primary cause is the lack of an evolutionary path to the new platform. Although it is tempting to make a clean break with the past, users will rarely choose to adopt a technology that makes their old software obsolete. A secondary cause of failure is a lack of extensibility, which allows a platform to be updated to meet changing requirements. Therefore, the Ivy platform that we propose is both extensible and evolutionary.

New Ivy features will be implemented via language extensions, which will plug into the compiler and provide new syntax, type checking rules, and code generation options. Language extensions will implement

system-specific checking, similar to the checks provided by MC [9] or SLAM [1]. These extensions will also give the programmer flexibility in choosing what rules the compiler should check, since extensions may be enabled or disabled selectively. For example, programmers will not need to enable the memory safety extension until their code conforms to the specific rules about regions and reference counting required by that extension.

The Ivy platform will also include refactoring tools so that programmers can evolve legacy code. The first step in refactoring a program is to convert it to Ivy without any extensions enabled. We describe this step in detail in Section 5; in brief, the goal is to eliminate use of the C preprocessor, which is not present in Ivy. Once a C program has been converted to Ivy, refactoring tools will enable the use of new language extensions for code that was not originally written with those extensions in mind. For example, a tool might use a region inference algorithm to make region-based memory management explicit, thereby allowing the memory safety extension to certify the code as memory safe. Unlike language extensions, which are run each time the compiler is used, refactorings are applied only once in the lifetime of the code. We expect that each language extension will be bundled with a refactoring to enable that extension on legacy code. Refactorings may require a small amount of user guidance, but they will be mostly automatic.

This approach has several advantages over designing a new language or creating more bug-finding tools:

- There is no need for manual translation of programs. Languages like Java and ML provide attractive features like memory safety, but they require that existing C code be rewritten. Even Cyclone, which is similar to C, requires extensive manual intervention. Ivy's refactoring tools will make use of program analyses, like the one found in CCured [3], to make these changes automatically. User guidance will be necessary but relatively rare. Although refactorings may be somewhat heavy-weight, they will be only applied once. Our goal is to shift much of the burden of static analysis out of a compiler or checker, which is run frequently, to a single-use refactoring. Since they are only run once, refactorings will have a somewhat larger "budget" of running time and user interaction that traditional bug-finding tools.

- Language extensions will build on each other. We expect that the guarantees provided by one extension will be used by other extensions. For example, many program analyses for C assume memory safety in order to operate correctly. Ivy extensions can make this requirement explicit by depending on the Ivy memory safety extension. As a more concrete example, an extension for checking protocol adherence (like MC, SLAM, or ESP) would be much easier to write if it could assume that all memory accesses respect types, that data is segregated into explicit regions, and that concurrent memory accesses occur only inside atomic sections. These guarantees would be provided by underlying Ivy extensions for memory safety and concurrency.

- Programmers can choose which extensions they need. Refactorings might be difficult to apply, since they make substantial changes to source code. In some cases, it may not be possible to apply all extensions to an extremely old and baroque code base, even with the aid of refactorings. But since Ivy extensions will be enabled selectively, programmers may use only those that are practical.

- Researchers can develop custom extensions and refactorings. Since systems software is constantly changing, it may be necessary in the future to create new Ivy extensions, as well as the refactorings that enable them. Ivy will be built to make this process as simple as possible.

4 Extension Examples

In this section, we present two examples that show how our new programming platform will help developers to write better systems software. In particular, these examples demonstrate the power of automated refactorings and language extensions. These examples are only theoretical, but we believe that they represent realistic uses of our proposed language.

CCured. The CCured project [3, 11] analyzes pointer usage in large software systems in order to add low-cost memory safety checks. CCured infers a pointer "kind" for each pointer in the program, and this pointer kind is the basis for the static checks and run-time instrumentation performed by CCured. Unfortunately, a significant amount of manual intervention is required in order to "cure" real programs, since CCured sometimes fails to understand why a particular piece of code is memory-safe. Worse, this manual intervention must be repeated with each subsequent release of the software package that is being cured. With our framework, CCured will be implemented as an extension and a refactoring: the refactoring will infer pointer kinds to the best of its ability, and the extension component will do the corresponding type-checking and instrumentation every time the software is compiled. Because the refactoring produces human-readable code, the programmer will have a chance to improve the results of CCured's analysis. Also, because this

annotated code becomes the master copy in the repository, the programmer need not repeat this procedure after every update to the project.

Linux's Sparse. Linux creator Linus Torvalds wrote a static analysis tool called Sparse [12], which is specifically tailored for checking the Linux kernel. It uses extra annotations added by programmers in order to verify certain kernel-specific properties. For example, programmers can annotate pointers that should contain user-space addresses, and Sparse will verify that none of these pointers are dereferenced directly by the kernel. With our framework, a refactoring will discover pointers that contain user-space addresses (much like the existing CQual project [7]), and an extension will be responsible for checking, at each compilation, that these pointers are never dereferenced. This approach makes annotations explicit in the code and thus integrates this check into the development process.

5 Macroscope: A First Step

Although "vanilla" Ivy code, without any extensions, is similar to C, there are some differences. The most important one is the lack of the C preprocessor (CPP) in Ivy. CPP poses a great challenge for refactoring tools: because the preprocessor is token-based rather than syntax-tree-based, refactoring tools cannot parse CPP code directly. Since refactorings are so critical to the Ivy platform, the first step in the translation to Ivy is the elimination of CPP. In its place, Ivy includes a flexible macro system that is based on syntax trees rather than tokens. Thus, refactoring tools can operate on Ivy macros directly, without first expanding them. This feature is critical to the success of refactoring tools, since they must preserve the readability and human understanding of the code in order to be useful. To solve this problem, we have developed a tool called Macroscope, which transforms a C program into an Ivy program while preserving the vast majority of the program's macros.

Macroscope translates macros, conditional compilation, and include files into equivalent Ivy constructs. The latter two cases are straightforward, since Ivy supports compile-time conditionals and modules. Macros, however, are quite difficult to handle, since they often consist of arbitrary sequences of tokens. In such cases, Macroscope translates the tokens to complete syntactic units using a variety of heuristics. It understands the entire CPP language, including token pasting, stringization, recursive macros, and varargs macros. In some cases, it may make a construct less general (in order to convert it to a complete syntactic unit); these cases are called imperfect translations. For example, if a macro expands to an identifier that is used sometimes as a variable name and sometimes as a type name, then Macroscope will generate two different Ivy macros for the one CPP macro. However, it will always produce Ivy code that is equivalent to the original CPP code. Additionally, Macroscope's output is readable by humans and extremely similar to the original input.

Program	Lines	Imperfect macro translations	Imperfect #ifdef translations
gzip	7,324	7 (0.5%)	18 (2.0%)
rcs	17,178	10 (0.6%)	59 (5.8%)
OpenSSH	55,153	10 (0.1%)	41 (1.9%)
Linux 2.6.10	163,154	88 (0.6%)	62 (2.7%)

Table 1: Benchmarks demonstrating the feasibility of translating CPP code to Ivy. In the case of Linux, only a minimally configured kernel was translated. An imperfect translation is any construct that is not a nearly token-for-token translation of the C preprocessor code. Imperfect translations nevertheless produce correct code.

Macroscope's execution is divided into two phases: expansion and extraction. In the expansion phase, macros, conditionals, and include files are rewritten to C code as they would be by CPP. Macroscope keeps a record of each expansion. Next, the code is parsed into a syntax tree using a standard C parser. Afterwards, the extraction phase backs out each expansion in reverse order using the expansion records. To extract a given construct, Macroscope identifies the lowest syntax tree node that encompasses all of the tokens from the expansion record. This node is replaced with an Ivy construct that closely resembles the original CPP construct that was expanded, using several heuristics that allow us to generate good Ivy constructs. Every Ivy macro that Macroscope generates is built from an entire node in the syntax tree, which ensures that Ivy macros are complete syntactic units. This feature is crucial, since it allows Ivy macros to be parsed and analyzed as is.

We have tested Macroscope on a set of open source programs that we believe is representative of the systems programs that users will need to translate. Our largest test case is a minimally configured Linux 2.6.10 kernel. We also applied Macroscope to `gzip`, `rcs`, and `OpenSSH`. Table 1 shows the results for these benchmarks. Imperfect translations may be the result of Macroscope generating an Ivy construct that is less general than a given CPP construct, which is undesirable but sometimes necessary. Imperfect translations nevertheless result in correct Ivy code. Based on these results, we believe that eliminating the preprocessor with Macroscope is completely feasible.

To demonstrate that Ivy code is not difficult to refactor,

we have developed a proof-of-concept refactoring tool that operates on the code produced by Macroscope. We have applied the tool to the Ivy version of `gzip` 1.2.4, which contains a well-known buffer overflow vulnerability involving the `strcpy` function. The tool is designed to fix `strcpy`-based buffer overflows. It replaces each call to `strcpy` with a safer version that will not overflow its destination buffer. This safer function requires that the size of the destination buffer be passed as an argument. The refactoring tool attempts to infer the size of the buffer statically. If it fails, the user must supply the size argument manually. When we refactored `gzip`, the tool automatically inferred the buffer size about 70% of the time. The static analysis that the tool uses is currently extremely simple, but in the future we hope to scale it up for use on much more sophisticated properties.

6 Related Work

A number of projects have attempted to craft a successor to C. For example, Cyclone [10] is a C-like language that is type-safe and that provides advanced language features in a systems programming environment. In addition, BitC [13] is a language that attempts to combine C's expressiveness with the rigor of a modern functional language. Unfortunately, neither of these languages provides a reasonable evolutionary transition path for existing software, without which it is difficult to make a real-world impact. In addition, our extensions and refactorings will allow programmers to customize Ivy for specific projects and to incorporate future language features, neither of which is supported by these alternatives.

Dawson Engler's research group has produced a number of program analysis tools, such as the MC system [9] and RacerX [5], both of which use static analysis to find bugs in large software systems. These projects have all been successful in uncovering serious bugs in real code; however, in order to scale to such large systems, they must make potentially unsound assumptions about properties of the code. Thus, these tools generate many false positives and occasional false negatives, making it impractical to incorporate them directly into the build process as we would like.

The SLAM project at Microsoft [1] is a program analysis tool that uses model checking to detect errors in Windows drivers. This tool provides even stronger guarantees about certain safety properties; however, it is currently used for individual drivers in isolation. A similar project at Microsoft, ESP [4], also can check that a program adheres to a given protocol. It is more scalable, but uses a weaker form of path sensitivity than SLAM. Nevertheless, we are encouraged by the results of MC, SLAM, and ESP, and we may use them as the basis for extensions and refactorings to check interface compatibility.

Finally, projects such as CCured [3] and Linux's Sparse [12] analyze existing software to improve memory safety and find defects. As mentioned earlier, these systems can be better implemented as part of our framework: the inference portion becomes a refactoring, and the static checks (as well as any corresponding run-time checks) become an extension.

7 Conclusion

The C language has a long and venerable history. Even today, despite its flaws, most systems software is written in C. However, in recent years, the flexibility of C has proven to be a liability as system designers focus more on reliability and security. We have proposed a new programming platform, Ivy, that provides the features of a modern, safe language along with an evolutionary path that will allow us to bring existing code up to date. Program transformations, called refactorings, are used to improve the safety and security of legacy code, and language extensions perform the necessary compile-time checks on refactored code. An initial translation step, which eliminates CPP and is mostly automatic, moves legacy code onto the platform, where the power of static analysis, refactoring and extensions can be fully applied. We hope Ivy will serve as a safe, modern platform for future systems research.

Acknowledgements: Thanks to George Necula, Rob von Behren and David Gay (of Intel) for their help on this project.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103–122, 2001.

[2] H.-J. Boehm. Threads cannot be implemented as a library. Technical Report HPL-2004-209, Hewlett Packard, 2004.

[3] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[4] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[5] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[6] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 338–349. ACM Press, 2003.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1–4, 1999.

[8] S. N. Freund and S. Qadeer. Checking concise specifications for multi-threaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[11] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[12] D. Searls. Linus & the Lunatics, Part I. *Linux Journal*, November 2004. `http://www.linuxjournal.com/article/7272`.

[13] J. Shapiro, S. Sridhar, S. Doerrie, M. Miller, and E. Northup. The BitC language specification. `http://www.coyotos.org/docs/bitc-spec/bitc-spec.html`.

[14] H. Xi and F. Pfenning. Dependent types in practical programming. In ACM, editor, *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.