Conditional Compilation is Dead, Long Live Conditional Compilation!

Paul Gazzillo
University of Central Florida
paul.gazzillo@ucf.edu

Shiyi Wei
University of Texas at Dallas
swei@utdallas.edu

***Abstract*—Highly-configurable systems written in C form our most critical computing infrastructure. The preprocessor is integral to C, because conditional compilation enables such systems to produce efficient object code. However, the preprocessor makes code harder to reason about for both humans and tools. Previous approaches to this challenge developed new program analyses for unpreprocessed source code or developed new languages and constructs to replace the preprocessor. But having special-purpose analyses means maintaining a new toolchain, while new languages face adoption challenges and do not help with existing software. We propose the best of worlds: eliminate the preprocessor but preserve its benefits. Our design replaces preprocessor usage with C itself, augmented with syntax-preserving, backwards-compatible dependent types. We discuss automated conditional compilation to replicate preprocessor performance. Our approach opens new directions for research into new compiler optimizations, dependent types for configurable software, and automated translation away from preprocessor use.**

***Keywords*—conditional compilation, preprocessor, C language, variability, dependent types**

I. Introduction

Highly-configurable software such as the Linux kernel and the Apache web server form our most critical infrastructure, underpinning everything from high-performance computing clusters to Internet-of-Things devices. Keeping these systems secure and reliable is essential. This software's variability, i.e., its ability to be configured and compiled with various combinations of features, enables one codebase to target different hardware and application-specific requirements. The Linux kernel, for instance, runs web servers, cell phones, refrigerators, and more.

Some of the largest and most configurable software is written in C. Linux, for instance, has more than 14,000 configuration options. Unfortunately, standard practice for these and most C software is the decades-old practice of implementing variability ad-hoc with brittle tools. In particular, developers need to *hand-code* configurations directly in source code with extensive preprocessor encodings, which appear as much as every couple of lines [1]. The preprocessor supports conditional compilation, where the compiler, given configuration options, produces different object code from the same source code. Conditional compilation is what enables the extreme customizability of a single codebase like the Linux kernel.

Supported in part by NSF CCF Awards #1840934 and #1816951

What is wrong with using the preprocessor for implementing variability? Due to the lack of support for variability in the C language itself, developers have to hand-code conditional compilation with the preprocessor, which opens the door to errors appearing in any variation of the compiled code. The tight coupling between the preprocessor and C language makes it difficult to understand, debug, and maintain the software. Due to this difficulty, software tools for C rarely consider the preprocessor usage fully [2], [3], [4], [5].

We argue the use of the preprocessor has created a dilemma: (1) the preprocessor is so entrenched in C development that it is integral to the C language, and (2) the preprocessor is such a serious impediment to the quality of C code that software tools cannot be expected to work well with unpreprocessed code. To the best of our knowledge, our community has tried to resolve this dilemma with two main research directions.

The first direction is to improve the state of the art in software tools for C. In recent years, researchers have made significant progress in developing *variability-aware analyses* that work on unpreprocessed C code, including new and modified algorithms for parsing [6], [7], [8], data-flow analysis [9], [10], [11], type-checking [12], and rewriting [13]. Nevertheless, there is still a lack of tools that can effectively detect critical bugs in real-world configurable software such as Linux across all configurations. More importantly, we believe this direction is not sustainable. It does not directly tackle the fundamental issue in developing configurable C software, i.e., hand-coding configurations with the preprocessor lead to bugs and performance issues. In addition, research on "variability-oblivious" program analysis marches on. Trying to maintain a variability-aware toolchain in parallel means having to play catch up with substantial engineering effort.

The second direction is to develop new preprocessors [14], [15] or adopt better development practices for configurable system software. Feature-Oriented Software Design (FOSD) [16] has brought new programming language constructs and paradigms, such as variational data structures [11], the choice calculus and variational programming [17], and variational execution [18], [19]. These approaches make features, i.e., configuration options, explicit in the language, which enables easier analysis on configurable code [16]. Language adoption, however, depends on more than just good science; it depends on social factors as well [20]. In spite of its known issues, C remains a popular language [21].

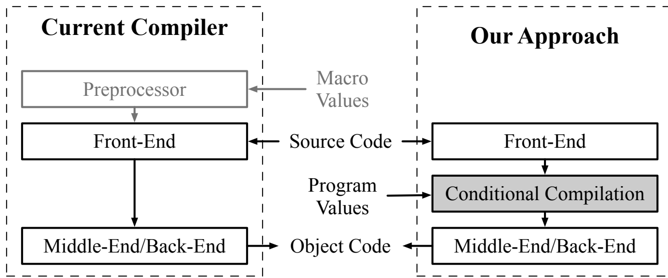

Fig. 1: Comparison of compiler phases.

Moreover, even if all new projects adopted good development methodologies, there are still pervasive C codebases like the Linux kernel that benefit from continued tool development.

We instead outline a new direction that resolves the dilemma by striking a balance between *replacing the preprocessor* (to enable tool support) and *preserving the use of C* (to ease adoption). Ideally, we would like to eliminate preprocessor usage altogether. The preprocessor's value proposition, however, is high-performance object code. Our approach uses existing C constructs plus a syntax-preserving extension to the type system to support some common usage of the preprocessor. To replicate the performance benefits of the preprocessor, we present a new conditional compilation phase in the compiler.

Our approach entails the following changes to the use of C for configurable code. (1) Replace the use of preprocessor conditionals with C conditionals when used within functions, as recommended by GCC coding standards [22]. (2) Replace the use of preprocessor macros with C variables for configuration options. (Section II). (3) Extend the C type system with *dependent types* that control the existence of declarations via expressions of C variables (Section III).

Due to the changes above, we propose a new compiler phase that supports automated conditional compilation by taking C variable values at compile-time. The compiler uses these values to resolve dependent type expressions and C conditionals as much as possible to produce efficient object code. A side benefit of this approach is that there is no distinction between compile-time and run-time variability. The developer is free to use the same program for either. The compiler is then free to optimize as much at compile-time as possible and add instrumentation to support run-time configuration.

Finally, our approach introduces interesting new research questions for our community to investigate (Section IV), including new compiler optimizations and automated translation.

II. Overview

Our goal is to replace preprocessor usage with *automated conditional compilation* that results in good object file size and performance. Figure 1 compares the compilers before and after removing the preprocessing phase and adding our proposed conditional compilation phase. The difference is that compile-time configuration options are no longer passed via preprocessor macros. Instead, C variable values may be passed at compile-time. The conditional compilation phase uses these values to optimize the resulting object code by removing code infeasible for a specific configuration.

```
1 #ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
2  if (imajor(inode) == 10)
3    i = 31;
4  else
5 #endif
6    i = iminor(inode) - 32;
```

(a) Implementation with the preprocessor.

```
1 bool CONFIG_INPUT_MOUSEDEV_PSAUX;
2 if (CONFIG_INPUT_MOUSEDEV_PSAUX) {
3  if (imajor(inode) == 10)
4    i = 31;
5  else
6    i = iminor(inode) - 32;
7 } else {
8  i = iminor(inode) - 32;
9 }
```

(b) Implementation with our approach.

Fig. 2: Configurable control flow. Adapted from Linux v2.6.33.3 drivers/input/mousdev.c.

The benefits of this approach are that there is only a single language used to implement configurable code, and conditional compilation is instead automated by the compiler. The preprocessor allows implementation of just about any program semantics, leading to poor tool support. Without the preprocessor, tools work with a simpler semantics and need not support the full power of the preprocessor.

We believe this approach can be implemented in an existing compiler, since most of the C language definition remains unchanged. There are two possible places to insert the phase in an existing compiler: just after parsing or just after type checking. Since we reuse existing syntax, the parser requires little or no change. In this case, the new phase will take all configuration options at compile-time and produce a transformed abstracted syntax tree with any configuration conditions resolved. In the latter case, C's static semantics are augmented with dependent types and a new phase is added after type-checking. This phase takes uses any given configuration options to resolve conditionals at compile-time, but produces run-time instrumentation to select types according to runtime program values. A combination of both phases may be possible.

To illustrate how this approach works, take the code snippet in Figure 2a. This example from the Linux kernel source code uses an `#ifdef` to alter the program, depending on whether the preprocessor macro `CONFIG_INPUT_MOUSEDEV_PSAUX` is defined or not. When the macro is defined, the program contains a complete if-then-else construct, i.e., lines 2-4 and 6. Line 6 is outside of the preprocessor conditional, so it is in all configurations. When the macro is not defined, the resulting object code contains no branch; only the assignment from line 6 is included.

Figure 2b shows how this code snippet would be implemented in our approach. The macro is replaced with the C variable declared on line 1, while the preprocessor conditional

is replaced with a C conditional on line 2. Notice that the two versions of the program in Figure 2 are equivalent in meaning. The if-then-else branch conditioned on `imajor` is only evaluated when `CONFIG_INPUT_MOUSEDEV_PSAUX` is defined, and the assignment from line 6 of Figure 2a still appears in all configurations, i.e., in both branches on lines 6 and 8. The use of C conditionals is more restrictive than `#ifdefs`. We consider this a benefit, since the "anything goes" freedom of the preprocessor leads to poor tool support.

The downside is that the performance benefits of the preprocessor are lost, because the resulting object code for Figure 2b is larger and slower compared to the `#ifdef` version. Our version has an extra branch due to line 2, which the preprocessor would have resolved at compile-time. Furthermore, the branch on line 3 will always appear in the object code, unlike the preprocessor version, because the configuration option is a program variable with a value unknown at compile-time.

To replicate the performance benefits of the preprocessor, our proposed compiler phase allows program variables to be set at compile-time, akin to partial evaluation. The compiler can remove the code for disabled configuration options without relying on preprocessor directives. The branches on lines 2 and 3 in Figure 2b are optimized away via constant propagation and dead code elimination, as long as the value of `CONFIG_INPUT_MOUSEDEV_PSAUX` is known at compile-time. With constant folding, even expressions involving C variables (e.g., `CONFIG_USB && CONFIG_PAGES > 0`) can be evaluated at compile-time and used to perform conditional compilation via dead code elimination.

GCC coding standards [22] even recommend using C conditionals over preprocessor conditionals when possible, and Linux developers have been converting `#ifdefs` to C conditionals[1]. This practice, however, only converts control structure, not the conditional expressions, which are still preprocessor macros. Macros are still used to provide constants to the compiler.

Overall, our approach expands on best practices that prefer C conditionals over preprocessor conditionals by replacing preprocessor usage entirely. C program variables are used in place of macros, which requires the compiler to accept program values at compile-time. This solution allows the compiler to optimize away code controlled by unselected configuration options.

III. Dependent Types for Conditional Compilation

Transformation of `#ifdefs` to C conditionals is not enough, because `#ifdefs` may appear anywhere in C code, including around declarations and function definitions. Since we use C variables as configuration options, type declarations can now include C variables. The existence of a declaration is predicated on C variables, which leads us to adopt the use of dependent types [23] in order to support configurable code.

Figure 3a, another code snippet extracted from the Linux kernel source, shows a common use of the preprocessor to configure the fields of a struct. `i_reserved_quota` is only a field of the struct when the macro `CONFIG_QUOTA` is defined. This can be seen as a variational data type [11].

```
1 struct {
2   u16 i_inline_size;
3 #ifdef CONFIG_QUOTA
4   qsize_t i_reserved_quota;
5 #endif
6 }
```

(a) Implementation with the preprocessor.

```
1 bool CONFIG_QUOTA;
2 struct {
3   u16 i_inline_size;
4   qsize_t __attribute__((config (CONFIG_QUOTA)))
        i_reserved_quota;
5 }
```

(b) Implementation with our approach.

Fig. 3: Configurable struct definition. Adapted from Linux v4.18 fs/ext/ext4.h.

```
1 #ifdef CONFIG_SMP
2 extern void cpu_load_update_active(struct rq *this_rq);
3 #else
4 static inline void cpu_load_update_active(struct rq *
      this_rq) { }
5 #endif
```

(a) Implementation with the preprocessor.

```
1 bool CONFIG_SMP;
2 extern void __attribute__((config (CONFIG_SMP)))
      cpu_load_update_active(struct rq *this_rq);
3 static void __attribute__((config (!CONFIG_SMP)))
      cpu_load_update_active(struct rq *this_rq) { }
```

(b) Implementation with our approach.

Fig. 4: Configurable function definition. Adapted from Linux v4.18 kernel/sched/sched.h.

Figure 3b shows how our approach can support such usage without the preprocessor. We specify the condition on the struct field in line 4 using an attribute specifier[2] in the type declaration. Attributes are part of the GCC version of the C grammar and are used for optimization, but we repurpose this syntax to express predicates. A Boolean expression of program variables describes the conditions under which the declaration exists. In this example, the expression is `CONFIG_QUOTA`. Note that this version of the program will compile with a standard C compiler, if it ignores the attributes. Our proposed compiler, however, can produce more efficient object code for certain configurations, since it does not have to lay out memory for the struct field when its predicate does not hold.

The type predicates also allow for multiple type declarations of the same identifier. Figure 4a shows two different declarations of the same function. The first (line 2) is an `extern` declaration, while the second (line 4) declares the function to be `static` and defines its (empty) body. Using the attribute

[1] https://lkml.org/lkml/2015/5/20/484

[2] https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html

specifiers, Figure 4b shows how different configurations of the function are declared with the mutually exclusive predicates `CONFIG_SMP` and `!CONFIG_SMP`. While these declarations are valid C syntax, this program will not compile with the current C compiler due to the multiple declarations of the same identifier. This requires our conditional compilation phase to compile correctly.

Our approach uses dependent types to represent the variability previously encoded with the preprocessor. The use of dependent types has been discussed before but dismissed as too limited for expressing variability, and because they are undecidable in general [24]. When used only for conditional compilation, however, this dependent typing scheme is decidable, because any program values used in type specifications are provided at compile-time.

Allowing dependent types for expressing configurable code opens the door to more sophisticated uses. Suppose we allow compilation without the values necessary to resolve configuration conditions at compile-time. The compiler then has the opportunity to perform static type checking and optimization across multiple configurations simultaneously. Also, blurring this line between compile-time and run-time configuration is possible because our approach makes no distinction between the two. Previous work on dependent types has shown that adding run-time checks can still make type-checking decidable [25]. This usage of dependent types enables run-time variability: for configuration options not provided at compile-time, the compiler can add instrumentation to read and evaluate configuration conditions at run-time, much like variability-aware execution [19]. Moreover, by falling-back to run-time variability, the user building the software is free to choose between compile-time and run-time configuration without having to rewrite any code.

IV. Research Directions

Several research questions emerge from our approach that guide future work: *What new compiler optimization algorithms can improve conditional compilation?* Dead code elimination works in some cases, but new optimizations may be necessary to match hand-coded preprocessor usage. *How much existing C code can be translated automatically?* Previous work on parsing and transforming unpreprocessed C code supports the possibility for automatic translations in such cases [6], [7], [13]. Empirical studies of preprocessor use in real-world code will help guide new translation algorithms that infer dependently-typed C. *How easily can existing program analysis and bug-finding techniques be repurposed?* While problems with analyzing unpreprocessed C can be eliminated, the semantics of the new type system may still need new theory and development for existing C analyses. *What are the formal semantics of the new type system, and what correctness properties can be proved?* Proving safety properties, such as the lack of null-pointer errors, would be useful across all configurations. Adding dependent types to existing formal semantics for C can enable verification of configurable C code. *How can the compiler efficiently support both compile-time and run-time variability at the same time?* New optimization algorithms can make the compiled object-code efficient, even when configuration options are not known at compile-time.

V. Conclusion

In this paper, we present a new research direction to resolve the dilemma of preprocessor usage, showing how some preprocessor usage can be replaced with C itself while still retaining the benefits of conditional compilation. Our position is that *both* compile-time and run-time variability should be represented in the same language. We argue that C code without the preprocessor will be easier to understand and debug, while preserving C syntax will help adoption. Our design leads to dependent types for configurable type declarations and compiler optimizations to automate conditional compilation. Our research opens new optimization, translation, and verification challenges for our community to investigate.

References

[1] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE TSE*, pp. 1146–1170, Dec 2002.

[2] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef confirmed harmful: Promoting understandable software variation," in *IEEE VL/HCC*, 2011, pp. 143–150.

[3] S. Schulze, E. Jurgens, and J. Feigenspan, "Analyzing the effect of preprocessor annotations on code clones," in *IEE SCAM*, 2011, pp. 115–124.

[4] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *ECOOP*, 2015, pp. 495–518.

[5] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, and A. Wasowski, "Variability through the eyes of the programmer," in *ICPC*. IEEE Press, 2017, pp. 34–44.

[6] C. Kästner *et al.*, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *OOPSLA*, Oct. 2011, pp. 805–824.

[7] P. Gazzillo and R. Grimm, "SuperC: Parsing all of c by taming the preprocessor," in *PLDI*. ACM, 2012, pp. 323–334.

[8] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *ICSM*, Sep. 2005, pp. 379–388.

[9] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years," in *PLDI*. ACM, 2013.

[10] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *ESEC/FSE*. ACM, 2013, pp. 81–91.

[11] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden, "Variational Data Structures: Exploring Tradeoffs in Computing with Variability," in *Onward!* ACM, 2014, pp. 213–226.

[12] C. Kästner, K. Ostermann, and S. Erdweg, "A Variability-aware Module System," in *OOPSLA*. ACM, 2012, pp. 773–792.

[13] A. F. Iosif-Lazar, J. Melo, A. S. Dimovski, C. Brabrand, and A. Wasowski, "Effective Analysis of C Programs by Rewriting Variability," *CoRR*, 2017.

[14] B. McCloskey and E. Brewer, "ASTEC: A New Approach to Refactoring C," in *ESEC/FSE*. ACM, 2005, pp. 21–30.

[15] C. Kästner, "Virtual separation of concerns: Toward preprocessors 2.0," Magdeburg, Germany, 5 2010.

[16] S. Apel, D. Batory, C. Kstner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[17] S. Chen, M. Erwig, and E. Walkingshaw, "A Calculus for Variational Programming," in *ECOOP*, 2016, pp. 6:1–6:28.

[18] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *ICSE*, 2014, pp. 907–918.

[19] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," in *OOPSLA*. ACM Press, 2018.

[20] L. A. Meyerovich and A. S. Rabkin, "Socio-PLT: Principles for Programming Language Adoption," in *Onward!* ACM, 2012, pp. 39–54.

[21] IEEE, "Interactive: The Top Programming Languages 2018," last accessed Sep 30, 2018. [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

[22] GNU, "Coding Standards," last accessed Sep 30, 2018. [Online]. Available: https://www.gnu.org/prep/standards/standards.html#Conditional-Compilation

[23] P. Martin-Löf, "Constructive mathematics and computer programming," in *Logic, Methodology and Philosophy of Science*, vol. 104, 1982.

[24] S. Chen, M. Erwig, and E. Walkingshaw, "Extending Type Inference to Variational Programs," *ACM TOPLAS*, 2014.

[25] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent Types for Low-Level Programming," R. De Nicola, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 520–535.