SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C

Zachary Patterson
The University of Texas at Dallas
USA
Zach.Patterson@utdallas.edu

Zenong Zhang
The University of Texas at Dallas
USA
zenong@utdallas.edu

Brent Pappas
University of Central Florida
USA
pappasbrent@Knights.ucf.edu

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

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

ABSTRACT

Variability-aware analysis is critical for ensuring the quality of configurable C software. An important step toward the development of variability-aware analysis at scale is to transform real-world C software that uses both C and preprocessor into pure C code, by replacing the preprocessor's compile-time variability with C's runtime-variability. In this work, we design and implement a desugaring tool, SugarC, that transforms away real-world preprocessor usage. SugarC augments C's formal grammar specification with translation rules, performs simultaneous type checking during desugaring, and introduces numerous optimizations to address challenges that appear in real-world preprocessor usage. The experiments on DesugarBench, a benchmark consisting of 108 manually-created programs, show that SugarC supports many more language features than two existing desugaring tools. When applied on three real-world configurable C software, SugarC desugared 774 out of 813 files in the three programs, taking at most ten minutes in the worst case and less than two minutes for 95% of the C files.

CCS CONCEPTS

• **Software and its engineering** → **Preprocessors**; **Source code generation**; *Software product lines.*

KEYWORDS

C preprocessor, syntax-directed translation, desugaring

ACM Reference Format:
Zachary Patterson, Zenong Zhang, Brent Pappas, Shiyi Wei, and Paul Gazzillo. 2022. SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3512763

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3512763

1 INTRODUCTION

The preprocessor is widely used when developing real-world C programs to enable flexible configuration and reuse of the software. Its extensive usage has significant impacts on the performance and reliability of C software. Past studies have shown that bugs exist only under some configurations of the software (i.e., *variability bugs*) [1], and the preprocessor makes bug detection challenging since only one configuration can be tested at a time [34].

Most research on bug detection in configurable C software focuses on a combinatorial testing approach [9, 14, 33, 47], which samples the large software configuration spaces, and then applies the testing and analysis techniques on each of the samples. However, the sampling-based approach lacks the guarantee of program correctness under all configurations. This limitation makes a sound static analysis more desirable in many applications.

Variability-aware analyses have been developed to detect parsing errors [16, 24], type errors [26], and run-time defects such as double free errors [42]. Viewing real-world C programs as being written in two languages (i.e., C and preprocessor), the variability-aware analyses analyze C software as a whole using data structures that represent the combined language. However, the development of the variability-aware analyses as specialty analyses of the two languages makes it infeasible to reuse many existing C static analysis tools which focus on analyzing only C (i.e., *variability-oblivious tools*). This explains the large gap in terms of bug detection capabilities between the state-of-the-art variability-oblivious tools (which can perform inter-procedural analysis using techniques such as separation logic [20] and model checking [7]) and variability-aware tools [42]. As research on variability-oblivious analysis marches on, variability-aware analyses lag behind, because they have to constantly maintain parity with the state-of-the-art, while supporting the preprocessor, which amplifies the engineering effort.

Our goal is to close the gap by transforming real-world C software that uses both C and preprocessor into pure C code, i.e., to *desugar* unpreprocessed C into a simpler subset of the language. Desugaring converts the preprocessor's difficult-to-analyze compile-time variability into run-time variability represented in pure C. This desugared code can then be analyzed by existing variability-oblivious tools which do not support unpreprocessoed C, or it can be used as a common intermediate representation for developing new variability-aware analyses.

The key challenge to our approach is that preprocessor usage often has no direct equivalence in C. The transformation tool needs to account for all interactions of the preprocessor with C usage and scale to all variations of the source code created by these interactions. Moreover, real-world C programs are not ideal, having syntactic and type errors in untested configurations [1], an obstacle to correct and complete transformation.

Prior work on the desugaring approach has limited support for real-world C. C Reconfigurator [21] is an effort to conduct variability-aware verification by transforming a subset of unpreprocessed C constructs. But it specifies its transformation on an idealized language instead of C; as such its implementation lacks support for many common C constructs including structs and some function definitions. Hercules [12, 41, 43] handles unpreprocessed C by traversing abstract syntax trees (ASTs) produced by the TypeChef [25] variability-aware C parser and type checker. Hercules' transformation is informally described and is also not over the complete C grammar; thus, its implementation has unsound support for C constructs, including structs, functions, and some expressions [41]. Moreover, it relies on the strict assumption that no type errors are present in any configuration, limiting its capability on real-world C.

In this paper, we introduce a newly designed desugaring transformation and implement it in a new tool called SugarC that is capable of desugaring real-world C programs. Because real-world C is not guaranteed to be type-safe or even syntactically valid in all configurations, our desugarer performs simultaneous type checking and transformation and preserves syntactic and type errors in the desugared output as run-time errors. SugarC is specified as a novel syntax-directed translation of unpreprocessed C to pure C, where we augment *C's formal grammar specification* with translation rules. This approach combines both the soundness of a formal grammar specification and the realism of using C's actual grammar specification. Using C's own grammar makes our support for C constructs explicit, as well as for those constructs we intentionally omit, such as the now-uncommon K&R-style functions [22].

SugarC's translation is defined by annotating each grammar construct's production with a semantic action that specifies the translation of that construct and by associating each construct with a semantic value. We represent semantic values in a data structure, called a *multiverse*, that has several generic operators, including the product operator. Product simplifies the specification of many transformation rules by encapsulating variation, with some constructs, such as statements, representable as identity transformations over multiverse values. However, naive desugaring of more complex C syntax, such as user-defined types, can cause exponential explosion in the desugared output. To realize SugarC as a practical tool, we employ novel optimizations that improve scalability for constructs such as structs, unions, and enums declarations, which can have multiple variations in real-world C.

To evaluate support for desugaring C constructs, we develop a benchmark we call DesugarBench. It consists of 108 hand-created programs, covering a wide range of constructs drawn from C's grammar specification. We evaluate SugarC as well as C Reconfigurator and Hercules, to compare support for unpreprocessed C. While no tool supports all constructs, we show that SugarC supports many more constructs, especially the kinds of challenging

```
1 #ifdef UINT
2 unsigned int x;
3 #endif
4 #ifdef CHAR
5 char x;
6 #endif
7
8 printf("%u\n", x);
```

(a) Unpreprocessed C code.

```
1  const bool __UINT, __CHAR;
2
3  int __x_1;
4  char __x_2;
5
6  if (__UINT && ! __CHAR) {
7    printf("%u\n", __x_1);
8  }
9  if (!__UINT && __CHAR) {
10   printf("%u\n", __x_2);
11 }
12 if ((__UINT && __CHAR) ||
13     (!__UINT && ! __CHAR)) {
14   __type_error();
15 }
```

(b) Desugared code.

Figure 1: An example of desugaring unpreprocessed C code.

cases found in real-world C. In addition, we perform an empirical evaluation of SugarC's scalability to desugar three real-world configurable programs (axTLS, Toybox, and BusyBox). SugarC desugared 774 out of 813 files in the three programs, taking at most ten minutes in the worst case and less than two minutes for 95% of the C files.

This paper makes the following contributions:

- The design and specification of a syntax-directed translation that simultaneously type checks and desugars unpreprocessed C. (Section 2)
- The realization of our desugarer in a new tool called SugarC that incorporates optimizations for scaling to real-world usage of C constructs. (Section 3)
- A new benchmark suite, DesugarBench, that measures support for desugaring unpreprocessed C. (Section 4)
- An evaluation that compares SugarC with prior work on DesugarBench, and on three real-world C programs that demonstrates the scalability of SugarC. (Section 5)

Significance. Some of the most critical software infrastructures are implemented as highly-configurable C programs, and this configurability increases the challenges of maintaining software quality while also rendering traditional bug detection and testing infeasible. SugarC is an important step in developing automated analyses that are capable of scaling variability-aware analyses to large, real-world programs. It creates the foundation for both leveraging existing variability-oblivious analysis tools and accelerating the development of new variability-aware analyses to increase the reliability and security of our software infrastructures.

We have made our SugarC specification and implementation, DesugarBench, and all experimental data available [36].

2 DESIGN OF SUGARC

The preprocessor adds several constructs to C: macro definition (`#define`) and expansion, header inclusion (`#include`), and conditional compilation (`#ifdef`). These constructs have subtle semantics when combined with C. For instance, making multiple declarations of the same symbol is not possible in C, but is legal in unpreprocessed C. Figure 1a has two declarations of `x` (lines 2 and 5). Since

these declarations are guarded by preprocessor conditionals, this program is legal as long as only one of the macros UNIT or CHAR is defined. Otherwise, there will be a type error, either an undeclared symbol error on line 8 (if neither macro is defined) or redeclaration error (if both macros are defined).

Figure 1b shows the desugared code. In general, SugarC represents C constructs affected by the preprocessor's compile-time behavior as equivalent run-time C behavior. Configuration macros (such as UNIT and CHAR) are represented as **const** bool C variables (line 1), and preprocessor conditionals are desugared depending on their context. The multiple declarations of x in the unpreprocessed code cannot be within C conditionals because of C's scoping rules and because there is no C syntax for conditionals outside of function bodies. Instead, we desugar these declarations of the same symbol as single declarations of different symbols, __x_1 and __x_2 (lines 3 and 4). Then the use of the original symbol x is multiplexed into all of its possible variations, guarded by C conditionals (lines 6–15). Note that our desugarer also represents compile-time errors at run-time, via calls to a __parse_error or __type_error function that only applies to the errant set of configurations, ensuring all variations of the unpreprocessed source are preserved (line 14).

We specify the desugaring as a *syntax-directed translation* of the unpreprocessed source to pure C. Syntax-directed translation is a classic transformation technique that works by augmenting a formal grammar with rules guiding the translation [2]. The translation is defined by (1) annotating each grammar construct's production with a *semantic action* governing the translation of the construct and (2) associating a *semantic value* that holds the result of the translation of that construct. In our case, each C construct's semantic action produces a pure C version of all variations of the unpreprocessed construct, storing the pure C code as a semantic value. This approach allows us to use a formal grammar specification to model the effects of the preprocessor, while still following the actual grammar of C. Our syntax-directed translation is specified as a *bison* grammar [17] with semantic actions. These semantic actions are read by the underlying parsing framework (we use SuperC [16] in our implementation) to generate a parser that dispatches control to the semantic actions after parsing each construct.

SugarC also performs simultaneous type checking during the transformation to preserve type and syntactic errors in the desugared output as run-time errors. Doing this is important for handling real-world code, because unpreprocessed source code is not guaranteed to be type-safe in all configurations. By type checking during desugaring, SugarC is able to weaken the assumption of type correctness for all configurations that prior works (Hercules and C Reconfigurator) rely on, while still desugaring the type-safe variations of the unpreprocessed source. Preserving these errors in the desugared output ensures that type-unsafe configurations are not silently transformed into valid variations of the source code.

2.1 Representing Semantic Values

To record all variations of a desugared construct, we use a *multiverse* object for semantic values. Similar to prior work on variational data structures [44], choice calculus [11], and Makefile static analysis [15], SugarC's multiverse holds a set of values, each tagged with a *presence condition*, i.e., a logical expression that represents the configurations in which the value appears.

We define a multiverse M as a set of n pairs of a source code construct s and presence condition p.

$$M = \{(s_1, p_1), (s_2, p_2), ..., (s_n, p_n)\}$$

M is always a finite set, because there are a finite number of possible variations of unpreprocessed source code due to preprocessor conditionals. For example, the multiverse for all variations of the variable usage of x in Figure 1a on line 8 would be:

$$\{(\text{`__x\_1'}, \_\_\text{UINT} \wedge \neg \_\_\text{CHAR}),$$
$$(\text{`__x\_2'}, \neg \_\_\text{UINT} \wedge \_\_\text{CHAR}),$$
$$(\text{`__type\_error()'}, \_\_\text{UINT} \wedge \_\_\text{CHAR} \vee \neg \_\_\text{UINT} \wedge \neg \_\_\text{CHAR})\}$$

To ensure all possible variations of each construct are captured, SugarC maintains two invariants of the presence conditions in the multiverse. First, the presence conditions must be *disjoint*, i.e., each pair of presence conditions should be mutually exclusive. This reflects the determinism of the preprocessor: one configuration yields one variation of the source code. Second, the presence conditions must be *covering*, i.e., the union of the presence conditions is logical True. This ensures that the transformation will record all variations of the input source code. The above multiverse satisfies these two invariants because a type error is present if __UINT and __CHAR are both defined or undefined, and a variation of x is present when only one of the macros is defined; the presence conditions are disjoint and cover all four interactions between the two conditions.

We define several generic operations on multiverses to ease the specification of transformation rules. The *product* operator lifts any binary operator, such as string concatenation, to multiverse operator by taking the cartesian product of all elements in the multiverse. For instance, as part of the transformation for the declarations on lines 2 and 5 in Figure 1a, the semantic action needs to concatenate the resulting desugarered declarations that are emitted on lines 3–4 in Figure 1b. Since there are two variations each, there are four possible variations of their concatenation.

The product of two multiverses A and B over a scalar operator $\circ$ is defined as follows:

$$A \times^{\circ} B = \{(a \circ b, p \wedge q) \mid \forall (a, p) \in A, \forall (b, q) \in B\}$$

This formal definition can be found in [15] in the context of Makefile analysis, albeit it was only implemented for string concatenation in that context. SugarC, however, embodies the generic version of this operator, automatically lifting string concatenation, list concatenation, declarator construct composition (used when collecting declarations), among other operators. The product operator allows for many semantic actions to be specified as a straightforward identity transform, with the multiverse encapsulating correct handling of multiple variations (see Section 2.3).

In addition, we define new operators including *scalar product*, which allows for one operand to be a scalar value instead of a multiverse; *filter* for trimming elements that have infeasible presence conditions; *deduplicate* for unioning elements with identical values; and *transfom*, a unary operator, which is effectively a map.

2.2 Representing the Symbol Table

SugarC maintains a symbol table for all variations for two reasons. First, it stores the renamings of identifiers, since multiple declarations of the same symbol may legally occur in unpreprocessed C code. Second, it stores type information so that the desugarer can also perform type checking. While configuration-aware symbol tables [13] and type checking [24] have been addressed in prior work, SugarC is the first to perform type checking and desugaring simultaneously, which addresses the challenge of handling real-world code that has no guarantee of type safety in all configurations.

Since we need to preserve all variations of the unpreprocessed source, we resolve this by generating a unique renaming for each variation of the symbol. This enables SugarC to choose the right renaming for the symbol wherever it is used. In the case of multiple possible renamings, SugarC generates a C conditional to ensure each variation of the identifier use is preserved (lines 6-11 in Figure 1b).

We define symbol table S as a structure that maps m identifiers v_i (from the unpreprocessed source code) to a multiverse of pairs of type $\tau_j^{v_i}$ and renaming $r_j^{v_i}$ for each identifier.

$$\begin{aligned} S = \{ & v_1 \mapsto ((\tau_1^{v_1}, r_1^{v_1}), p_1), ((\tau_2^{v_1}, r_2^{v_1}), p_2), ...), \\ & v_2 \mapsto ((\tau_1^{v_2}, r_1^{v_2}), p_1), ((\tau_2^{v_2}, r_2^{v_2}), p_2), ...), \\ & ..., \\ & v_m \mapsto ((\tau_1^{v_m}, r_1^{v_m}), p_1), ((\tau_2^{v_m}, r_2^{v_m}), p_2), ...)\} \end{aligned}$$

A typical type checker will use the lack of an entry in the symbol table to determine whether a symbol has not yet been declared, e.g., to prevent multiple declarations, and use-before-declaration. Our configuration-aware symbol table, however, may have an entry in some configurations while the symbol has not been declared in others. To account for this, we add special type entries to each symbol's multiverse, called *undeclared* and *error*. With these entries, the symbol table captures in which configurations the symbol is defined or has a type error. This ensures that each symbol's associated multiverse of definitions meets the covering invariant.

2.3 Semantic Actions for Desugaring

We use pseudo-code to show the semantic actions. An action's function is named after the grammar construct, e.g., WhileLoop or Declaration, while its parameters are the components of the grammar rule. For instance, a WhileLoop construct has the following context-free grammar rule:

WhileLoop → 'while' Expression Statement

Its semantic action function thus has the following pseudo-code signature:

WhileLoop(Expression, Statement)

Semantic actions in our implementation have access to *global parsing state*. In particular, this includes the symbol table for the current scope, which is called `symtab` in the semantic actions below.

We now highlight a few illustrative semantic actions from a variety of constructs. The complete grammar with all semantic actions can be found in the anonymized, released artifact.

Algorithm 1 The semantic action for transforming while loops.

Input: Multiverse values for the Expression and Statement.
Output: A desugared while loop as a multiverse.
1: **function** WhileLoop(Expression, Statement)
2: **return** ('while' $\times^+$ Expression) $\times^+$ Statement

2.3.1 Statements. Desugaring statements in the unpreprocessed C language involves no more than taking the cross-product of all variations of the components of statements. This is possible because transforming a multiverse of statements only requires surrounding each multiverse element with a C conditional.

Algorithm 1 shows the semantic action for while loops. Line 1 defines the semantic action function WhileLoop which takes Expression, a multiverse of conditional expressions, and Statement, a multiverse of C statements. Line 2 is the cross-product of all component symbols of the WhileLoop construct, including the 'while' keyword and semi-colon. The $\times^+$ product operator lifts the string concatenation scalar operation to all pairs of multiverse elements, updating the presence conditions accordingly. Note that scalar semantic values such as 'while' are not multiverses and we use a special scalar product operator to take the product of a scalar and multiverse.

The representation and preservation of all variations of the source program is encapsulated by the multiverse, enabling a straightforward definition of the translation rule. For instance, if the Expression construct has two variations due to a preprocessor conditional, while all other constructs have only one variation. The repeated applications of the product operator will yield two complete variations of the entire WhileLoop which are stored as a single multiverse object.

The desugaring of compound statements can be expressed similarly using a lifted concatenation. In practice, this can yield an exponential explosion of variations. This explosion can be avoided by simply swapping out the semantic value of the compound statement. Instead of using a string, our specification using a list of strings and lifts list concatenation instead of string concenation. This allows for the translation rule to remain simple, while improving performance in practice.

2.3.2 Declarations. Handling declarations is more complicated than statements, because (1) preprocessor conditionals around declarations are not equivalent to C conditionals due to scoping rules, (2) multiple declarations of the same symbol are prohibited, and (3) there is no language support for conditionals outside of function bodies. Instead, SugarC produces one, unconditional declaration for each variation, but renames the symbol. In order to ensure the renaming is applied to all uses of the symbol, the symbol table records all variations of the symbol simultaneously.

Algorithm 2 shows the semantic action for declarations. C declarations contain a type specifier, which can be primitive types, structs, unions, etc., and a declarator, which can both give the symbol its name as well as declare compound types such as functions and pointers. The semantic action takes a multiverse of values for the type specifier and declarator. Line 2 first combines the two multiverses into all possible pairs of type specifiers and declarators by

Algorithm 2 The semantic action for transforming declarations.

Input: Multiverse values for the TypeSpec and Declarator.
Output: Desugared declarations, i.e., multiverse of declaration renamings.

```
1: function Declaration(TypeSpec, Declarator)
2:   D = TypeSpec ×^pair Declarator
3:   for each ((name_i, τ_i), p_i) ∈ D do
4:     symtab[name_i] ← Update(symtab[name_i], τ_i, p_i)
5:   A = Rename(symtab, TypeSpec)
6:   B = Rename(symtab, Declarator)
7:   return A ×^+ B ×^+ ';'
```

lifting an operator that creates a pair of two elements, i.e.,

$$A \times^{\text{pair}} B = \{((a,b), p \wedge q) \mid \forall (a,p) \in A, \forall (b,q) \in B\}$$

Lines 3 and 4 update the configuration-aware symbol table with each declaration. Conceptually, the update operation replaces the prior multiverse entry for the given (name_i, τ_i) pair. This update needs to account for both redeclaration errors and also ensure that each declaration only affects the configuration defined by the presence condition p_i. The function ensures this by conjoining the presence condition p_i of the new declaration with that of the old, p_{old}. The Update function is defined as follows:

$$\textsc{Update}(M, \tau_i, p_i) = \bigcup_{(\tau_{\text{old}}, p_{\text{old}}) \in M} \begin{cases} \{(\text{error}, p_{\text{old}})\} & \text{if } \tau_{\text{old}} = \text{error} \\ \{(\tau_i, p_{\text{old}} \wedge p_i), & \\ \quad (\text{undeclared}, p_{\text{old}} \wedge \neg p_i)\} & \text{if } \tau_{\text{old}} = \text{undeclared} \\ \{(\text{error}, p_{\text{old}} \wedge p_i), & \\ \quad (\tau_{\text{old}}, p_{\text{old}} \wedge \neg p_i)\} & \text{if } otherwise \end{cases}$$

There are three cases to consider when updating a symbol table entry: (1) If the previous entry was a type error, then the entry remains a type error. (2) If the entry was undeclared, then the entry is replaced with the newly-given type. Since there may still be some configurations left undeclared, the undeclared entry remains under the configurations where the new entry does not overlap, i.e., $p_{\text{old}} \wedge \neg p_i$. (3) If the entry was already declared, the entry becomes an error in the configurations that overlap between the old and new declarations. For non-overlapping configurations, the entry remains the same. C's type checker has subtle rules for redeclaration not reflected in this pseudo-code but are present in our implementation: redeclarations are prohibited in the function-local scopes but are permitted in the global scope as long as the types are identical (which C allows to support having separate function definitions from function declarations in headers).

Once the symbol table is updated for all variations of the declaration, the action produces the desugared version of each variation (lines 5-7). Our actual implementation creates a new identifier for each new variation of a symbol's declaration and stores it alongside the type in the symbol table. We abbreviate the symbol table here for brevity, and instead use Rename in place of the process of looking up the renamings in the table.

The TypeSpecifier also needs renaming because of user-specified type names. Declaration syntax in C is complicated, supporting not only variable declarations, but also additional specifier syntax for structures, unions, and enums, user-defined types (**typedef**), separate declarator syntax specifying compound types (functions, pointers, and arrays), lists of declarators in one declaration, differences between locally- and globally-scoped symbols, and more. User-defined structures and types have consequences for the scalability of desugaring, and we leave a discussion of our optimized handling of these in Section 3.1.

2.3.3 Expressions. Like statements, expression translations are also specified with an identity transform. For instance, a binary C expressions take the cross-product of each element of the multiverses from the left and right expressions, i.e., Left $\times^{\text{op}}$ Right. Unlike statements, however, expressions have type. In order to perform type-checking and desugaring simultaneously, the semantic value of an expression is a multiverse of pairs (s, τ) to hold both the desugared expression s and its type τ. The product $\times^{\text{op}}$ lifts the C construct's operator itself, both concenating the desugared expression and checking type compatibility, i.e., $A \times^{\text{op}} B =$

$$\{((a+\text{op}+b, \tau_{\text{op}}(\tau_a, \tau_b)), p \wedge q) \mid \forall ((a,\tau_a),p) \in A, \forall ((b,\tau_b),q) \in B\}$$

This desugaring operation is particularly subtle, because the operator is used both as a string value for producing a source translation $a + \text{op} + b$ and as a function type $\tau_{\text{op}}(\tau_a, \tau_b)$. Essentially, this product operation takes each combination of variations of the left side and the right side, then for each combination produces both the desugared string translation and the result of checking the type compatibility of the operands for the operator. Observe that this formulation gracefully handles type errors, since the error type is carried along just for the type-unsafe configurations until it can be replaced at the statement-level with a `__type_error()` call.

3 REALIZING SUGARC IN PRACTICE

Using cross products from the ground up and killing parsing error branches would create a correct output. But this approach would lead to incredibly large desugared source files. To reduce the resulting size of the transformation and create output that is amenable to static analysis, we perform several novel optimizations that make desugaring feasible with real-world code. These optimizations involve the user-defined types created by structs, unions, and enums as well as our design choices for error representation.

3.1 Handling Struct, Union, and Enum

C allows for user-defined types in the forms of struct, union, and enum (SUE) constructs. The preprocessor usage inside and outside of SUE definitions cause complex interactions which have not been properly addressed by past works, as they either ignore SUE constructs or transform them with the naive combinatorial approach. But using the strategy of taking a cross-product of all possible struct definitions creates a large explosion of desugared variations, since real-world structs commonly use configurable data types as well as have structs as field members that may themselves have multiple definitions. As an optimization, we represent the fields from all variations of an input struct using a single combined SUE specifier when possible. In Figure 2a, the preprocessor usage in lines 1-5 defines two possible types of `thisSize`. Using `thisSize` to declare the field `f` in struct `X`, the type of `f` depends on the value of the preprocessor condition `WORDSIZE`. Figure 2b shows the result of applying SugarC on the code. Because each struct stores its fields

```
1  #if WORDSIZE == 32
2  typedef long int thisSize;
3  #else
4  typedef int thisSize;
5  #endif
6  struct X {
7       thisSize f;
8  };
9  void foo () {
10      struct X data;
11      data.f = 3;
12 }
```

(a) Struct with a configurable field.

```
1  typedef long int thisSize_1;
2  typedef int thisSize_2;
3  struct X_3 {
4       thisSize_1 f_4;
5       thisSize_2 f_5;
6  };
7  void foo_4 () {
8       struct X_3 data_5;
9       if (WORDSIZE == 32)
10          data_5.f_4 = 3;
11      if (!(WORDSIZE == 32))
12          data_5.f_5 = 3;
13 }
```

(b) Desugared code.

Figure 2: Desugaring struct with a configurable field.

in the symbol table presented in Section 2.2, getting a member for `f` would only return mutually exclusive conditions for each possible definition. This can be seen in Figure 2b where `data.f` expands into an if statement representing the presence conditions associated with `f_4` and `f_5` respectively.

Multiple conditional SUE definitions and forward references. However, it is not always possible to use one object to represent every SUE definition. For instance, a flexible array is an array with no defined size value, but upon memory allocation through malloc, the array can use any number of elements as any extra memory is attributed to the size of that array. Due to this unique property, a flexible array must be the *last* element of a struct, and each struct *may only have one* flexible array. This specific instance can be seen in Toybox as structs used for status messages, as shown in Figure 3a. If we were to desugar this in the same manner that we handled the code in Figure 2a, then the result would violate the one flexible array per struct rule. So as Figure 3b demonstrates, we are forced to split *struct optval_status* into two separate structs.

This makes forward references to SUE objects challenging since now the declaration could refer to any of the definitions. In addition, they are also forcibly pointers, since otherwise their size cannot be determined. As we continue parsing the code, we post-hoc add the SUE definitions to the union, and print it at the top of the scope, but below the moved up SUE definitions as displayed in Figure 3b.

Anonymous Objects. It is common for real-world SUE objects to contain unnamed SUE objects as fields, or for variables to be defined with a struct or union without naming the structure. We give every anonymous object a name, which is "anonymous_#" and additionally a renaming "anonymous_#_#". This is so that later symbol table accesses can find the specific renamings that may be linked to an anonymous object without running into renamings from a different anonymous object. Since anonymous nested SUE objects can have their fields referenced by their parent, we print an additional field access when encountered, which is necessary since the inner object may have to be split into multiple objects. The types associated with the variables themselves become important as we get into sizeof and typeof operations.

```
1  #if WORDSIZE == 32
2  typedef long int thisSize;
3  #else
4  typedef int thisSize;
5  #endif
6  struct optval_status *status;
7  struct optval_status {
8       unsigned int status;
9       thisSize message[];
10 };
```

(a) Struct with a flexible array

```
1  typedef long int thisSize_1;
2  typedef int thisSize_2;
3  struct optval_status _5{
4       unsigned int status_3;
5       thisSize_1 message_4[];
6  };
7  struct optval_status _7{
8       unsigned int status_3;
9       thisSize_2 message_6[];
10 };
11 struct
        __forward_tag_reference_8
        {
12      union {
13      struct optval_status_5
             optval_status_5;
14      struct optval_status_7
             optval_status_7;
15      };
16 };
17 struct
        __forward_tag_reference_8
        *status_9;
```

(b) Desugared code.

Figure 3: A simplified excerpt from Toybox where flexible arrays requires the transformation to separate the structs.

Sizeof. Sizeof becomes problematic with our method of combining SUE object fields, as the size of the transformed SUE object no longer represents the size of what any given configuration may have. In Figure 4a, fields of struct `Y` are conditionally defined. Specifically, if `A` is defined, then `Y` contains an **`int`** field `b`; if `A` is not defined, the field `b` is a **`short`**. As discussed above, this struct is transformed into lines 6-11 in Figure 4b by representing all fields in one object. If no special treatment is given to the **`sizeof`**, **`sizeof(struct X_4)-sizeof(struct Y_9)`** will result in a negative value (the size of three ints in `X_4` minus the size of three ints and a short in `Y_9`) and therefore cause a type error.

Our solution, shown in lines 12-24 in Figure 4b, is to create a *standin struct* for what the definition would be under each presence condition. Then in lines 25-30, we make two versions of the array; `array_20` is associated with `A` being defined, and `array_21` is associated with `A` being undefined. Therefore, the standin structs used in expressions at lines 27 and 30 correlate with `__sizeofStandin_16` and `__sizeofStandin_19`, respectively. This approach allows us to account for whatever padding that might be incurred by the compiler to make the size as accurate as possible. While this introduces overhead, it is practical because the use of **`sizeof`** on conditional defined structs does not appear frequently in real code. We handle a union object similarly when **`sizeof`** is used.

Initialization. Struct objects are initialized with an initializer list consisting of constant values (line 17 in Figure 4a). This brings two challenges to SugarC. First, if there are different presence conditions within the initializer list, multiple definitions need to be created. Second, depending on which preprecessor condition is used, different fields could be assigned to, as shown in Figure 4b. To handle these challenges, we define the struct as a standalone definition, and then assign each value individually, as shown in lines 32-40 in

```c
struct X {
     int a;
     int b;
     int c;
};
struct Y {
#ifdef A
     int a;
     int b;
#else
     int a;
     short b;
#endif
};
int array[sizeof(struct X)-
      sizeof(struct Y)];
void foo() {
     struct Y y = {4,2};
}
```

(a) Example of initialization and sizeof usage on structs.

```c
struct X_4 {
     int a_1;
     int b_2;
     int c_3;
};
struct Y_9 {
     int a_5;
     int b_6;
     int a_7;
     short b_8;
};
struct {
  typeof (int) __tmp_10;
  typeof (int) __tmp_11;
  typeof(int) __tmp_12;
  } __sizeofStandin_13;
struct {
  typeof (int) __tmp_14;
  typeof (int) __tmp_15;
  } __sizeofStandin_16;
struct {
  typeof (int) __tmp_17;
  typeof (short) __tmp_18;
  } __sizeofStandin_19;
int array_20[
     sizeof(typeof(
           __sizeofStandin_13
           )) -
     sizeof(typeof(
           __sizeofStandin_16
           ))];
int array_21[
     sizeof(typeof(
           __sizeofStandin_13
           )) -
     sizeof(typeof(
           __sizeofStandin_19
           ))];
void foo_23() {
     struct Y_9 y_22;
     if (A)
          y_22.a_5 = 4
     if (!A) {
          y_22.a_7 = 4;
     if (A)
          y_22.b_6 = 2;
     if (!A)
          y_22.b_8 = 2;
}
```

(b) Desugared code.

Figure 4: Desugaring sizeof when used on structs, and initializing struct variables.

Figure 4b. If the struct or list is constant, or in the global scope, we have no choice but to duplicate the definitions and correct which values are assigned to what fields.

3.2 Emitting Errors and Line Numbers

As shown in Figure 1b, we represent type errors at the statement level alongside type-safe variations of the code, which allows static analyses to identify and terminate these control flows. If one or more of the expressions in a statement contain type errors, then the error is lifted to the statement level for representation as an error

```c
IP_UDP_DHCP_SIZE = sizeof(struct ip_udp_dhcp_packet) -
      CONFIG_UDHCPC_SLACK_FOR_BUGGY_SERVERS,
char c[IP_UDP_DHCP_SIZE == 576 ? 1 : -1];
```

Figure 5: Excerpt from BusyBox for illustrating SugarC's expression evaluation limitation.

function call; the entire configuration is invalidated by a type error, which would halt compilation if selected as a single variation of the unpreprocessed C. Since our desugarer performs type-checking during transformation, we can express type error handling in our translation rules. SugarC represents presence conditions in the output by creating a variable for each Boolean predicate in the presence condition, declared as `const bool`. It records presence conditions as symbolic formulas, using z3 [10] and BDDs [45]. When emitting any presence condition, we replace the Boolean predicate with its corresponding variable name.

Syntactic errors in a configuration, however, represent violations of the grammar on which we specify our translation rules. Since there is no way to identify what illegal syntax was supposed to mean, and branches may fork across different scopes, there is no consistent way to correctly display the error alongside valid code; instead we collect the presence conditions of all branches with syntactic errors into a single unioned presence condition. We then emit the union as a run-time conditional check that results in a runtime error if triggered. By guaranteeing this check is performed at runtime, the remainder of the desugared output can elide run-time checks for configurations that have syntactic errors. This significantly reduces the number of static conditionals that need to be emitted during desugaring; we have observed desugared outputs reduced to less than 10% of their previous sizes without this optimization in syntactic error handling due to the reduction in the size of output presence conditions.

In addition, the underlying parser generation was modified to emit line numbers in the output that correspond to the unpreprocessed source line numbers. This enables us to automatically map the desugared output back to the unpreprocessed input, which we use to automatically map alarms in desugared output to their unpreprocessed counterparts.

3.3 Limitations

There are several limitations in the current implementation of SugarC. First, because SugarC does not take any system configuration as input, it cannot properly evaluate the value of the size of types. This results in improper sizes or unexpected calculations in some configurations. In Figure 5, for SugarC to determine if `IP_UDP_DHCP_SIZE == 576` requires evaluating the size of each field of the struct `ip_udp_dhcp_packet`. This means that SugarC needs to make judgments about the size of types such as int or long, which it cannot do without using additional system configuration as inputs. Similarly, SugarC cannot validate the names of registers for assembly statements. In terms of nonequivalent output, there may be different values associated with enum values than expected. Since each subsequent value increases, combining the lists of enums together could make expressions evaluate differently.

Table 1: DesugarBench.

Category	Abbr.	# of Programs
(Declarations) basic	BASIC	18
(Declarations) struct, union and enum	SUE	16
(Declarations) typedef	TYPEDEF	5
(Declarations) external	EXTERN	4
(Function definitions) ANSI	ANSI	7
(Function definitions) K&R	K&R	6
(Function definitions) variadic	VARG	4
(Expressions) function call and identifier	FCI	10
(Expressions) unary, binary and ternay	UBT	7
(Statements) control-flow	CTRL	6
Semantic error	SEM	21
Syntactic error	SYN	4

Second, variadic arguments are only partially supported in SugarC. While ellipsis can be parsed, and is supported when used in functions and type checking, usage of `va_args` is not identified as a type and therefore will result parsing errors as attempting to parse as a new identifier.

Third, our handling of **`sizeof`** in structs may cause memory issues when executing the desugared code. This could happen when **`sizeof`** is used in the context of `malloc`, which requires the actual size of a desugared struct.

Lastly, the implementation of the language is not complete. Features introduced in later versions of C, such as declarations leading switch cases, are not supported by our parser. Extensions provided by compilers may also not be supported such as `_builtin_offset_`. There are also a few obsolete representations, such as K&R style function definitions, that we do not support.

4 DESUGARBENCH

We present DesugarBench, a benchmark that allows for comparisons between desugaring tools in terms of their support for C features. DesugarBench consists of 108 configurable C files under 12 categories to represent language features that are important for desugaring tools to handle. Table 1 shows the categories, abbreviations and number of programs of DesugarBench. We will refer to the categories by their abbreviations in the rest of the paper.

We define the following principles for generating DesugarBench programs. First, the benchmark programs should cover many C language and program features. We first inspect the grammar discussed in Section 2, and identify four high-level constructs: declarations, expressions, function definitions, and statements. These constructs are further divided into the first 10 categories in Table 1 that cover different language features. For example, standard C compilers still support the original K&R C functions declaration as well as the now-common ANSI C functions, which have very different syntax [22]. We therefore include K&R and ANSI as two categories of function definitions. As discussed above, syntactic and/or semantic errors may exist in some configurations in real programs. Thus, we include cases of these in two categories, SEM and SYN of DesugarBench. Each benchmark program is manually created and is meant to test a specific feature. Some programs are created to test common features. For example, the program *function_pointer* in the ANSI category is designed to test if a function pointer defined in a static branch can be correctly transformed. Others simulate corner cases. For example, *function_no_param_dec* in the K&R category describes an unusual situation where a function can be legally declared without parameters and defined with parameters in ANSI style, as the declaration can be interpreted as K&R style.

Second, the benchmark programs are created with small sizes and not biased toward certain features. The small sizes make it easier to reason about the expected results of the desugaring tools (i.e., the program semantics should be preserved), since we manually check for semantic equivalence. The sizes of programs in DesugarBench range from 3 to 37 lines of code. In addition, we make sure that there are not redundant benchmark programs for the same feature by checking the syntax. Category BASIC, SUE and SEM cover more C syntax than other categories. Thus, they contain more programs, as shown in Table 1.

Third, each benchmark program should include at least one static conditional. This ensures that each program is the appropriate target of the desugaring tools, which transform the unpreprocessed C program into an equivalent program represented in C. In DesugarBench, all programs in the declarations, expressions, function definitions, and statements categories do not contain errors in any static branch, while each program in the semantic and parsing error categories contains an error in one of its static branches.

5 EVALUATION

We conducted experiments to answer two research questions on SugarC's practicality of desugaring real-world preprocessor usage.

- **RQ1:** Does SugarC support more language features than existing desugarers?
- **RQ2:** Can SugarC desugar real-world C programs?

To answer **RQ1**, we compare SugarC with C Reconfigurator and Hercules on DesugarBench. We check if these tools can transform the benchmark programs into semantically equivalent programs. Using this result, we discuss how well the features in each benchmark category are supported by each tool. We also compare the execution results of the original files and the SugarC-desugared files under same static conditionals to check implementation correctness. To answer **RQ2**, we run SugarC to desugar three real-world programs, and report its performance. In addition, we compile and apply an off-the-shelf static analysis tool on the transformed programs.

5.1 Experimental Setup

5.1.1 Running desugaring tools on DesugarBench. We implemented a script to run SugarC, C Reconfigurator, and Hercules on all DesugarBench programs and semi-automatically diagnose the outputs by compiling each desugared file using GCC to check if the tools produce valid C programs. If a tool fails to generate a desugared file (e.g., tool crashes)[1] or if the desugared file does not compile with GCC, we report that the tool fails on this benchmark program. Otherwise, one author performed manual review to decide

[1]Recall that Hercules relies on TypeChef to catch type errors and does not transform any programs with such errors. C Reconfigurator uses a similar strategy for syntactic errors. For the SEM and SYN categories in DesugarBench, we treat C Reconfigurator and Hercules as passing these programs if the correct errors are caught.

Table 2: SugarC, C Reconfigurator, and Hercules results on DesugarBench.

Tool	Declarations				Function Defintions			Expressions		Statements	SEM	SYN	Total
	BASIC	SUE	TYPEDEF	EXTERN	ANSI	K&R	VARG	FCI	UBT	CTRL			
SugarC	18 | 0	16 | 0	5 | 0	4 | 0	7 | 0	0 | 6	1 | 3	9 | 1	7 | 0	6 | 0	20 | 1	4 | 0	97 | 11
C Reconfigurator	5 | 13	5 | 11	3 | 2	3 | 1	3 | 4	0 | 6	1 | 3	1 | 9	5 | 2	3 | 3	0 | 21	3 | 1	32 | 76
Hercules	12 | 6	12 | 4	4 | 1	2 | 2	7 | 0	2 | 4	0 | 4	6 | 4	6 | 1	5 | 1	17 | 4	0 | 4	73 | 35

if each desugared file is semantically equivalent to its original program.

To check if SugarC-desugared files can produce equivalent results as the original files under the same configuration, we update each benchmark program in all categories except SEM and SYN to return different values from the `main` function under different static conditionals. We then use a file with external declarations of the static conditionals in each original file as Booleans. These Booleans are evaluated to true or false to execute the desugared file. This allow us to compare the results obtained from executing the desugared files with the results obtained from executing the original files under the same configurations.

In our experiments, we used the latest versions of Hercules [19] and C Reconfigurator [28] as of August 2021. We observed out-of-memory when running C Reconfigurator (implemented in Java) with its default configuration. Therefore, we assigned JVM options *-Xms64g -Xmx64g* to allocate enough memory. Default configurations of SugarC and Hercules were used. We also had to make all macro names in the benchmark to start with `ENABLED_` for C Reconfigurator to work properly. All three tools took less than 5 minutes to run on all DesugarBench programs.

5.1.2 Running SugarC on real-world programs. We evaluated SugarC on three real-world programs, axTLS 2.1.4 [4], Toybox 0.7.5 [39], and BusyBox 1.28.0 [6]. We chose these programs because they are common targets in the past evaluations of variability-aware analyses [1, 21, 26, 33, 35, 37], and contain large numbers of configuration options (94, 316, and 998 in axTLS, Toybox, and BusyBox, respectively). We ran SugarC on all the `.c` files in each program. Any header inclusions, included directories, or macro definitions given by the build system were automatically extracted from the output of running `make`. Since hand-verifying semantic equivalence for the entire programs was cost prohibitive, we say that the desugarer produced a compilable output if the output can be compiled by GCC without error. We also applied an off-the-shelf static analysis tool, Clang Static Analyzer, on all the compilable desugared outputs to further validate if this tool can produced bug reports using the desguared results as inputs.

Setting system macros. Most real-world C programs make use of Standard Library headers, which contain large number of system macros. These headers need to be desugared along with each `.c` file that includes them, but much of the code does not apply to our system. To address this, we identify a set of macros that models the behavior of our system, assuming only GCC 9.4.0 [18] and Ubuntu 16.04 [40] (the environment in which we ran SugarC). This set of macros has all the default macros of GCC 9.4.0 turned on, and all macros that are associated with other system environments turned off, which were identified over six weeks by 4 computer science student workers with C development experience. This resulted in 129 macros either being defined or undefined. When desugaring a `.c` file, we run SugarC with the `-nostdinc` option to prevent desugaring any files outside our set. Additionally, we replace GCC-specific code with an equivalent macro. For example, `builtin_offset()` function is replaced by the macro definition `#define __builtin_offsetof(st,m)((size_t)&(((st *)0)->m))`.

Setting guard macros. We also predefine the marcros that are not intended for use as configuration macros and instead are undefined by default; we call those *guard macros*. Such macros are commonly used as header inclusion guards to prevent files from being included multiple times and to mark typedefs as being defined to avoid redefinition. In total, there were 257 guard macros in the standard libraries, 1 in axTLS, 1 in BusyBox, and 454 in Toybox as it has flags for each file to turn on cleanup statements.

Experimental environment. All experiments were conducted on a server with 192GB of RAM and 48 CPUs running Ubuntu 16.04. To measure SugarC performance, it was run three times on each file in the real-world programs, and we report the median. We set a 1-hour timeout for each file.

5.2 RQ1: Does SugarC support more language features than existing desugarers?

5.2.1 Semantic Equivalence. Table 2 shows the results of SugarC, C Reconfigurator, and Hercules, organized by the categories of DesugarBench. Each cell of the table reports the number of benchmark programs passed and failed, respectively, with green and red bars that represent the proportion of passed and failed benchmarks for each category and tool. We observe that *SugarC supports many more features than C Reconfigurator and Hercules.* Specifically, SugarC passed *all* benchmark programs in 8 out of 12 categories, while C Reconfigurator and Hercules only passed all programs in 0 and 1 categories, respectively.

For SugarC, all but one program in the K&R and VARG categories failed. As discussed in Section 3.3, SugarC currently does not support K&R style function definitions and `va_list` used in variadic functions. The one program in the VARG category passed because this program only defines a variadic function using ellipsis, which SugarC supports, but does not use `va_list`. SugarC failed on 1 out of 10 programs in the FCI category. The failing program involves the operator `offset`, a compiler extension provided by GCC. It is also a limitation of SugarC as discussed in Section 3.3. Finally, SugarC failed on one program in the SEM category. This program has a type error in one configuration by declaring an array with negative size. SugarC does not evaluate expressions and check array

bounds, and so produces a program that retains the negative array size type error. For the 2 programs that SugarC failed on FCI and SEM category, C Reconfigurator and Hercules also did not pass.

C Reconfigurator and Hercules failed on 76 and 35 out of 108 benchmark programs, respectively. We observed several common reasons for these failures. First, C Reconfigurator and Hercules failed on 12 and 5 programs, respectively, due to mistakes in renaming multiply-declared symbols. For example, in one program in the BASIC category, Hercules renames the multiple declarations of variable `x` inside of preprocessor conditionals but does not rename a subsequent use of `x`, causing a type error in its desugared code. Second, Hercules failed on 5 programs due to incomplete desugaring, i.e., missing variations from the original program. C Reconfigurator did not fail on any program for this reason. One program in the SYN category has two static branches, one with valid syntax (`int *x`) and the other not (`int *`). Hercules preserved the valid declaration but omitted the invalid one in its resulting transformation, representing an invalid configuration as valid. Third, the error handling mechanisms in both tools are incomplete. C Reconfigurator in particular failed on all programs with type error (SEM), because it does not perform any type checking. While C Reconfigurator reports some syntactic errors, it still missed one from the SYN category. Hercules, which relies on a separate tool—TypeChef—for type checking, failed to identify type errors in four programs in the SEM category. Hercules also missed all programs in the SYN category.

Other programs reveal additional reasons of failure. For C Reconfigurator, it incorrectly handles C's "most closely nested" scoping rule to transform a reference to a local variable into a reference to a global variable in two BASIC programs. Forward references, structs with a flexible array, `sizeof`, and struct initialization are not supported by C Reconfigurator, since by design it supports a restricted subset of C. Additionally, the benchmark evaluation shows that it misses most function calls. Hercules had partial support for structs, unions, and enums, but it missed `sizeof` and struct initialization.

When executing SugarC-desugared files, we found that for all but 2 files, they produced the equivalent outputs as the original programs under the same configurations. One was the result of a bug in SugarC in handling a static conditional that exists between the function declaration and body, and another was due to mistyping of an enum initialized with a large value, making it `long long`.

In summary, SugarC's formal approach to specifying desugaring rules enables it to support a wide-range of real-world C features correctly, in contrast to the informal approaches of prior work.

5.3 RQ2: Can SugarC Desugar Real Programs?

Table 3 shows the results of running SugarC on axTLS, Toybox, and BusyBox. The first row shows the number of C files in each program that SugarC produced a compilable output and those that it did not. The second row is the median running time in seconds SugarC took to desugar the files with the semi-interquartile range (SIQR) in a smaller font. Overall, *for 774 out of 813 files SugarC produced an output that when compiled by GCC had no errors, and for 95% of these files, SugarC took less than 2 minutes per file.*

Table 3: SugarC results on axTLS, Toybox, and BusyBox.

	axTLS	Toybox	BusyBox
# of files "compilable | uncompilable"	28 | 0	230 | 0	516 | 39
Median runtime (sec) and SIQR	77 31	47 14	51 2

In Table 3, the median desugaring times across all three programs are between 47 and 77 seconds, showing that SugarC is efficient in desugaring many programs. The two files that took the longest time to complete were both from Toybox (`posix/ps.c` and `pending/traceroute.c`), which took 586 and 614 seconds, respectively. The longest running times for a file on axTLS and BusyBox were 302 seconds and 484 seconds, respectively. After manual investigation, we conjecture that the overhead in `ps.c` was likely due to a constant array of structs defined in the global scope named `typos` , which is referenced frequently throughout the remainder of the code. The array uses `va_args`, which causes a series of type errors that have a multitude of type errors in many variations. The overhead in `traceroute.c` was likely due to a 60 line switch statement where every case is a macro definition.

Among the 39 files that SugarC could not desugar or had errors in its output, SugarC did not produce desugared results in 20 files, and the desugared files could not be compiled by GCC in 19 BusyBox files. For the 19 files not desugared in BusyBox, 15 files used statements unsupported by SugarC, while the remaining 4 either took longer than an hour to execute or ran out of memory. Manual investigation of the 4 files that ran into timeout or memory issues revealed two issues. Either the file had a large series of consecutive `#ifdef` statements which led to a large multitude of parser branches, or the file had a long list of conditional values inside of an enum or list, which caused exponential explosion.

For the 20 desugared files that could not be compiled, the recurring errors were (1) an inability to evaluate expressions, which led to negative-sized arrays, and (2) assembly statements looking for registers that did not exist; both issues were discussed in Section 3.3. The other encountered issues are: (1) the inability to differentiate between constant variables and constant numbers when getting addresses; (2) external declarations showing up later as static declarations; (3) an unidentified instance where field names in a struct initialization were not renamed; (4) incomplete struct types not being marked as errors. Future updates to our data structures for representing types would allow us to correct the first issues, and further work is needed to reconcile the cause of the third issue with the implementation.

The median size of the original files was 3KB, while the median size of the desugared files was 1.2MB. However, the sizes between the original file and the desugared file do not have a linear relationship. When fitting the linear regression with original file size as the independent variable and the desugared file size as the dependent variable, the R^2 value was 0.155. On the other hand, the R^2 of the number of static conditionals (as independent variable) to the desugared file size (as dependent variable) was 0.511. We also find the largest desugared files were not the files with the most conditions, but files which featured a complex struct or global list with multiple static conditionals inside.

We ran Clang Static Analyzer on the 774 desugared files across three programs. Clang Static Analyzer reported 336, 1122, and 3255 alarms running on the desugared code of axTLS, Toybox, and BusyBox, respectively. When performing its static analysis, Clang Static Analyzer checks for certain function names (e.g., `malloc`) to find all errors besides those of the type `Dead store` and `Logic error`. Since SugarC renames all functions and variables during the desugaring process, Clang Static Analyzer was only able to produce bug reports of these two types in the desugared code. Nonetheless, this provides further evidence that SugarC produced meaningful translations.

6 THREATS TO VALIDITY

There are two potential threats to the validity of our evaluation. First, DesugarBench may not represent all language features and could introduce bias toward SugarC. To mitigate this, we constructed the benchmark programs to cover the grammar, including cases currently not supported by SugarC (e.g., K&R). In addition, the author who constructed DesugarBench did not have knowledge of SugarC's implementation. Second, the three real-world programs used in the evaluation may not represent all C features and preprocessor usage. Indeed, we improved the SugarC implementation while desugaring these programs; we expect that other programs also have features SugarC currently does not support. Still, these programs are common targets of variability analysis [1, 21, 26, 33, 35, 37], suitable for evaluating SugarC's scalability.

7 RELATED WORK

Hercules and C Reconfigurator. C Reconfigurator [21] uses SuperC [16] to parse the code, and then uses Xtend [46] to perform the transformation. The transformation rules of C Reconfigurator were only specified and proven on the idealized imperative language IMP. The prototype implementation was evaluated on simplified excerpts from BusyBox and the Linux kernel, as well as Libssh files [29]. As shown in Section 5.2, C Reconfigurator has limited supported for many C features. SugarC, in constrast, specifies its transformation rules on a formal C grammar, and implements optimizations to support real-world C usage.

Hercules [12, 41, 43] presents a tree transformation on TypeChef's variational AST [26]. Hercules also relies on TypeChef's type checker to find type errors, which halts if it finds a type error in any configurations, preventing Hercules from being able to transform such cases. Hercules transformed SQLite and BusyBox, albeit when provided with a feature model for the software. Unlike SugarC, Hercules' transformation only has an informal description, and thus only partial support for many C language features as shown in Section 5.2. Moreover, SugarC's simultaneous type checking and desugaring approach allows for transforming programs that are not type-safe in all configurations while preserving the compile-time errors as run-time errors.

Variability-aware analysis and parsing. Different variability-aware static analyses have been developed in the past [3, 5, 8, 23, 27, 30, 32, 38, 42]. These approaches often perform special-purpose analyses on variational data structures that represent both the preprocessor and C. For example, Rhein et al. [42] built data- and control-flow analyses on top of the variational ASTs and control flow graphs to detect bugs. SugarC takes a direction complementary to these variability-aware analyses, desugaring preprocessor usage into C to allow the desugared results to be used as a common intermediate language for variability-oblivious and variability-aware analyses. It is significant that our evaluation demonstrates that the desugaring approach is scalable because this may significantly reduce the engineering efforts to develop new variability-aware analyses.

SugarC, as a syntax-directed translation, is closely related to past works on variability-aware parsing. TypeChef [25] performs both variability-aware parsing and type checking on a specified configuration space for a given file. The tool creates an AST that holds variability-aware information inside of it, which has been used as the basis for several variability-aware works, including variability-aware static analysis [37] and refactoring [31]. SuperC [16], which SugarC uses as the underlying parser, takes a different approach to also create a variational AST which uses several optimizations to improve the process.

8 CONCLUSIONS

This paper presented SugarC, a novel desugaring tool that uses a syntax-directed translation to transform preprocessor usage into pure C. SugarC performs type checking and desugaring simultaneously to allow programs with syntactic and type errors in some configurations to be desugared, and these compile-time errors to be preserved as run-time errors. SugarC specifies its translation rules on real C grammar and includes novel optimizations to handle the challenging, real-world user-defined types. We create DesugarBench to compare existing desugaring tools, Hercules and C Reconfigurator, with SugarC. The results show that SugarC supports many more C language features than past tools. Our evaluation also shows that SugarC is scalable to desugar three real-world programs and demonstrates the applicability of these results as inputs to the Clang Static Analyzer.

In the future, we plan to build an analysis framework that uses SugarC-desugared results as the intermediate language and develop new variability-aware analyses for bug detection. We will continue improving and maintaining SugarC to efficiently desguar many more real-world code and support complete language features.

ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-1840934 and CCF- 1816951.

REFERENCES

[1] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 3, Article 10 (Jan. 2018), 34 pages. https://doi.org/10.1145/3149119
[2] A.V. Aho, A.V. Aho, R. Sethi, J.D. Ullman, and J.D. Ullman. 1986. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company.
[3] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. 2010. Type safety for feature-oriented product lines. *Automated Software Engineering* 17, 3 (2010), 251–300.
[4] axTLS. 2016. *axTLS Embedded SSL.* http://axtls.sourceforge.net
[5] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years *(PLDI '13).* Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/2491956.2491976
[6] Busybox. 2021. *BUSYBOX.* https://busybox.net
[7] CBMC. 2021. *C Bounded Model Checker.* https://github.com/diffblue/cbmc

[8] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 335–344.

[9] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 38–48. http://dl.acm.org/citation.cfm?id=776816.776822

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[11] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (Dec. 2011), 27 pages. https://doi.org/10.1145/2063239.2063245

[12] Florian Garbe. 2017. *Performance Measurement of C Software Product Lines*. Master's thesis.

[13] Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. In *ICSM*. 379–388.

[14] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. 2009. An improved meta-heuristic search for constrained interaction testing. In *2009 1st International Symposium on Search Based Software Engineering*. IEEE, 13–22.

[15] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/3106237.3106283

[16] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 323–334. https://doi.org/10.1145/2254064.2254103

[17] gnu. 2014. *GNU Bison*. https://www.gnu.org/software/bison/

[18] gnu. 2019. *Using the GNU Compiler Collection (GCC)*. https://gcc.gnu.org/onlinedocs/gcc-9.4.0/gcc/

[19] Hercules. 2017. *Hercules*. https://github.com/joliebig/Hercules

[20] Infer. 2021. *Infer static analyzer*. https://github.com/facebook/infer

[21] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2017. Effective Analysis of C Programs by Rewriting Variability. *CoRR* (2017).

[22] ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland. 683 (est.) pages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[23] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21 (2012).

[24] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 805–824.

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. *SIGPLAN Not.* 46, 10 (Oct. 2011), 805–824. https://doi.org/10.1145/2076021.2048128

[26] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-aware Module System. In *OOPSLA*. ACM, 773–792.

[27] Kim Lauenroth, Klaus Pohl, and Simon Toehning. 2009. Model checking of domain artifacts in product line engineering. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 269–280.

[28] Alex Lazar and Jean Melo. 2017. C Reconfigurator. https://github.com/itu-square/c-reconfigurator

[29] libssh. 2021. The SSH library! https://www.libssh.org/

[30] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 105–114.

[31] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. 2015. Morpheus: Variability-Aware Refactoring in the Wild. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 380–391. https://doi.org/10.1109/ICSE.2015.57

[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/2491411.2491437 event-place: Saint Petersburg, Russia.

[33] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 643–654.

[34] Jean Melo, Claus Brabrand, and Andrzej Wasowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 679–690. https://doi.org/10.1145/2884781.2884831

[35] Austin Mordahl. 2019. Toward Detection and Characterization of Variability Bugs in Configurable C Software: An Empirical Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 153–155. https://doi.org/10.1109/ICSE-Companion.2019.00064

[36] Zachary Patterson, Zenong Zhang, Brent Pappas, Shiyi Wei, and Paul Gazzillo. 2021. SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C. https://doi.org/10.5281/zenodo.5915048.

[37] Alexander Von Rhein, JörG Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 18 (Nov. 2018), 33 pages. https://doi.org/10.1145/3280986

[38] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. 2012. Analysis strategies for software product lines. *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012* (2012).

[39] Toybox. 2021. *Toybox*. https://github.com/landley/toybox

[40] ubuntu. 2018. Ubuntu 16.04.7 LTS (Xenial Xerus). https://releases.ubuntu.com/16.04/

[41] Alexander von Rhein. 2016. *Analysis strategies for configurable systems*. Ph.D. Dissertation. Universität Passau.

[42] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 27, 4 (2018), Article No. 18. https://doi.org/10.1145/3280986

[43] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *J. Log. Algebraic Methods Program.* 85, 1 (2016), 125–145. https://doi.org/10.1016/j.jlamp.2015.06.007

[44] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 213–226. https://doi.org/10.1145/2661136.2661143

[45] John Whaley. 2007. JavaBDD. http://javabdd.sourceforge.net/.

[46] xtend. 2021. Java with spice! http://www.eclipse.org/xtend/

[47] C. Yilmaz, M. B. Cohen, and A. A. Porter. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering* 32, 1 (Jan. 2006), 20–34. https://doi.org/10.1109/TSE.2006.8