Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations

ZHENYANG XU, University of Waterloo, Canada
YONGQIANG TIAN, Monash University, Australia
MENGXIAO ZHANG, University of Waterloo, Canada
CHENGNIAN SUN, University of Waterloo, Canada

Program reduction is a widely used technique in testing and debugging language processors. Given a program that triggers a bug in a language processor, program reduction searches for a canonicalized and minimized program that triggers the same bug, thereby facilitating bug deduplication and simplifying the debugging process. To improve reduction performance without sacrificing generality, prior research has leveraged the formal syntax of the programming language as guidance. Two key syntax-guided transformations—Compatible Substructure Hoisting and Quantified Node Reduction—were introduced to enhance this process. While these transformations have proven effective to some extent, their application excessively prunes the search space, preventing the discovery of many smaller results. Consequently, there remains significant potential for further improvement in overall reduction performance.

To this end, we propose a novel syntax-guided transformation named Structure Form Conversion (SFC) to complement the aforementioned two transformations. Building on SFC, we introduce three reduction methods: Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization, designed to effectively and efficiently leverage SFC for program reduction. By integrating these reduction methods to previous language-agnostic program reducers, Perses and Vulcan, we implement two prototypes named $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$. Extensive evaluations show that $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ significantly outperforms Perses and Vulcan in both minimization and canonicalization. Specifically, compared to Perses, $\text{SFC}_{\text{Perses}}$ produces programs that are 36.82%, 18.71%, and 41.05% smaller on average on the C, Rust, and SMT-LIBv2 benchmarks at the cost of 3.65×, 16.99×, and 1.42× the time of Perses, respectively. Similarly, $\text{SFC}_{\text{Vulcan}}$ generates programs that are 14.51%, 7.65%, and 7.66% smaller than those produced by Vulcan at the cost of 1.56×, 2.35×, and 1.42× the execution time of Vulcan. Furthermore, in an experiment with a benchmark suite containing 3,796 C programs that trigger 46 unique bugs, $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ reduce 442 and 435 more duplicates (programs that trigger the same bug) to identical programs than Perses and Vulcan, respectively.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

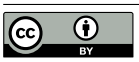
Additional Key Words and Phrases: Program Reduction, Test Input Minimization, Automated Debugging

ACM Reference Format:
Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, and Chengnian Sun. 2025. Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 275 (October 2025), 27 pages. https://doi.org/10.1145/3763053

1 Introduction

Program reduction is a widely utilized technique in the testing and debugging of language processors such as compilers, interpreters, and debuggers [20–22, 29, 32, 36, 37, 41]. Given a program

Authors' Contact Information: Zhenyang Xu, University of Waterloo, Canada, zhenyang.xu@uwaterloo.ca; Yongqiang Tian, Monash University, Australia, yongqiang.tian@monash.edu; Mengxiao Zhang, University of Waterloo, Canada, m492zhan@uwaterloo.ca; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, Canada.



ACM 2475-1421/2025/10-ART275
https://doi.org/10.1145/3763053

P that triggers a bug in a language processor, along with an oracle ψ that verifies whether a program triggers the bug, program reduction aims to produce a minimized, canonicalized version of P that still triggers the same bug. By minimizing bug-irrelevant code from P, this technique streamlines the debugging process. The significance of such minimization is underscored by the fact that many language processor communities mandate that users and developers perform reduction on bug-triggering programs before reporting bugs [4, 6, 14, 23]. Furthermore, program reduction also facilitates the canonicalization[1] of bug-triggering programs, reducing differences among duplicates (two programs are duplicate if they trigger the same bug). This capability is particularly advantageous for deduplication in large-scale automated testing campaigns of language processors [2].

Program reduction presents significant challenges for two primary reasons. First, the original bug-triggering programs can be considerably large since many of these programs are generated through fuzz testing [20, 29, 41], and empirical evidence suggests that configuring a fuzzer to generate larger programs can enhance bug-finding performance [41]. Meanwhile, a language processor is typically extremely complex, and thus reducing the program by directly analyzing which part is essential for triggering the bug is almost impossible. Second, programs are highly structured and adhere to complex, strict grammatical rules. Naively removing parts of a program using algorithms like `ddmin` [42] often produces syntactically invalid code, which is unlikely to reproduce the original bug, leading to inefficient and ineffective reduction.

To address these challenges, many trial-and-error, *syntax-aware* program reduction techniques have been proposed over the past decades [8, 19, 25–27, 30, 39]. One notable example is Hierarchical Delta Debugging (HDD), proposed by Misherghi and Su [25]. HDD builds on the `ddmin` reduction algorithm but introduces a hierarchical approach: instead of directly applying `ddmin` to the program, it first transforms the program into a tree representation (*e.g.*, an abstract syntax tree) and then applies `ddmin` at each level of the tree in a top-down manner. This hierarchical strategy significantly improves both the effectiveness and efficiency of reduction in cases involving highly structured inputs. However, HDD still tends to generate a substantial number of syntactically invalid programs during the reduction process, which hampers its overall performance. This issue was addressed by Perses, a *syntax-guided* program reducer [30]. Perses additionally takes as input the formal syntax of the programming language and applies only *syntax-guided* transformations (*i.e.*, transformations that preserves the syntactical validity) during reduction. By doing so, Perses avoids generating syntactically invalid programs, leading to improved performance compared to HDD.[2]

Prior Syntax-Guided Transformations. Perses introduces two syntax-guided transformations: Compatible Substructure Hoisting and Quantified Node Reduction. Compatible Substructure Hoisting replaces a given syntactical structure in P with one of its sub-structures (*i.e.*, a descendant of the given syntactical structure in the parse tree). For instance, it can transform the expression `a+b` into one of its sub-expressions (*e.g.*, `a` or `b`). Quantified Node Reduction uses `ddmin` to reduce syntactically parallel or repeated structures, such as a sequence of statements or a list of function parameters, These syntax-guided transformations preserve syntactical validity, have

[1] Canonicalization aims to reduce different programs that trigger the same bug to an identical program or a set of programs with minimum differences. This concept is further discussed in § 2.1.

[2] Besides the language-agnostic reducers mentioned in the paragraph, there are many language-specific reducers such as C-Reduce and ddSMT [19, 26, 27]. These reducers are not only syntax-awared but also specifically optimized for the target languages. However, these tools require considerable language-specific knowledge and cannot be well generalized to reducing programs in other languages. In this paper, the focus is improving language-agnostic program reduction approaches, and thus these language-specific approaches are not discussed in detail.

been shown to be highly effective, and have since been adopted by various program reduction techniques [7, 8, 34, 39, 45].

The Missing Piece. Despite their success, the two prior syntax-guided program transformations are inherently limited because, given a structure in P, many smaller yet syntactically valid structures cannot be generated. Relying solely on these two transformations for program reduction excessively prunes the search space $\mathbb{P}$ and can ultimately result in inadequate reductions [39]. For instance, consider a function invocation expression `f(e1,e2,e3)` in a C program P where the arguments `e1`, `e2` and `e3` generally represent three expressions rather than just variables, and assume that evaluating these three expressions altogether is necessary to trigger a bug, while calling `f` is irrelevant to the bug. Neither Compatible Substructure Hoisting nor Quantified Node Reduction can reduce this expression. Compatible Substructure Hoisting attempts to reduce the entire expression by replacing it with one of its sub-expressions, *i.e.*, `e1`, `e2`, or `e3`. Note that according to the C language, the syntactical structure of `e1,e2,e3` in the parse tree of calling `f` is an argument list but not an expression. Thus, Compatible Substructure Hoisting cannot replace the entire expression with `e1,e2,e3`. Quantified Node Reduction attempts to reduce the syntactically parallel structure (*i.e.*, the argument list) with the `ddmin` algorithm (*e.g.*, Quantified Node Reduction can possibly reduce the expression to `f(e1)`). Nevertheless, there exist other transformation strategies that still preserve syntactic validity. For example, deleting the function name `f` and the parentheses leaves `e1,e2,e3`, which is also a valid expression in C. Although this transformation, missed by both Compatible Substructure Hoisting and Quantified Node Reduction, only involves deleting three tokens, it can open up further reduction opportunities across the entire program P by eliminating a use of the function `f`. Namely, if `f` is unused elsewhere, its definition can also be entirely removed.

Structure Form Conversion. To mitigate the limitations of Compatible Substructure Hoisting and Quantified Node Reduction, in this paper, we propose a novel syntax-guided transformation named Structure Form Conversion (SFC). SFC is designed to transform a given syntactic structure into others forms that preserve the syntactical validity of whole programs. For instance, given the expression `f(e1,e2,e3)`, SFC attempts to transform it to other forms of expressions defined by the formal syntax, such as the previously mentioned expression `e1,e2,e3`. Unlike Compatible Substructure Hoisting and Quantified Node Reduction, converting a structure to another form does not ensure the resulted structure is smaller than the original one, and the number of structures that can be possibly generated is vast or even infinite. To tackle this challenge, we establish five guiding principles for SFC based on the chracteristics of program reduction, in order to constrain the space of generated programs. Additionally, we design three reduction methods—Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization—to effectively apply SFC in program reduction. Specifically, Smaller Structure Replacement reduces a program by replacing structures with smaller, compatible ones generated by SFC. Identifier Elimination strives to create reduction opportunities by utilizing SFC to eliminate the uses of an identifier, in the hope of successfully removing the corresponding definition or initialization of the identifier. While these two methods focus on enhancing minimization, Structure Canonicalization emphasizes canonicalization. It transforms structures into the most canonical forms with SFC, making duplicate programs (*i.e.*, those that trigger the same bug) more likely to be reduced to identical or similar forms. By integrating these three reduction methods into existing language-agnostic program reducers, Perses and Vulcan [39], we develop two prototype reducers: $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$.

To evaluate the performance of SFC in minimization and canonicalization, we utilize two benchmark suites, `Benchmark-Reduce` and `Benchmark-Cano`, respectively. `Benchmark-Reduce` contains 20 C benchmarks from Perses [30], 20 Rust benchmarks partially from Vulcan [39], and 205 SMT-LIBv2 benchmarks from ddSMT2.0 [19]. Each benchmark consists of a program triggering

a unique real-world crash or miscompilation bug. In contrast, `Benchmark-Cano` contains many duplicated C programs that triggers the same bug, including 2,501 programs triggering 11 unique crash bugs in GCC 4.3.0, and 1,295 programs triggering 35 miscompilation bugs also in GCC 4.3.0 [2]. Experiments with `Benchmark-Reduce` show that the results of $\text{SFC}_{\text{Perses}}$ is 36.82%, 18.71%, and 41.05% smaller than those of Perses on the C, Rust, and SMT-LIBv2 benchmarks, respectively. In terms of execution time, $\text{SFC}_{\text{Perses}}$ takes 3.65×, 16.99×, and 3.97× the time of Perses on average. As for $\text{SFC}_{\text{Vulcan}}$, it can produce 14.51%, 7.65%, and 7.66% smaller results than Vulcan at the cost of 1.56×, 2.35×, and 1.42× the execution time of Vulcan, respectively. Moreover, experiments with `Benchmark-Cano` show that $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ can reduce 442 and 435 more duplicates (programs that trigger the same bug) to identical programs than Perses and Vulcan, respectively.

Contributions. This work makes the following contributions.

- We propose a novel syntax-guided transformation, Structure Form Conversion, complementing to prior transformations Compatible Substructure Hoisting and Quantified Node Reduction.
- We propose three novel reduction methods to effectively and efficiently leverage SFC in program reduction. We further implement two language-agnostic reducer prototypes, $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ by integrating these methods into existing reducers.
- By conducting extensive evaluation, we demonstrate the $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ significantly outperforms Perses and Vulcan in terms of both minimization and canonicalization capability.
- For reproducibility and replicability, we have released the implementation and data at https://github.com/sfc-reducer/sfc-reducer.

2 Preliminary

This section provides the essential background knowledge relevant to this work.

2.1 Program Reduction

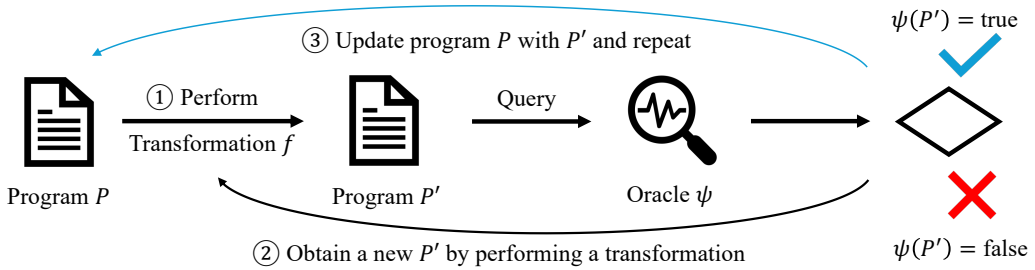


Fig. 1. A typical workflow of a program reducer.

Let $\mathbb{P}$ denote the search space of program reduction, namely, the set of all possible programs during reduction. Given an oracle $\psi : \mathbb{P} \to \{\text{true}, \text{false}\}$ that can verify whether a program triggers a specific bug, and a bug-triggering program $P \in \mathbb{P}$ *s.t.* $\psi(P)$, program reduction aims to output a minimized bug-triggering program P_{min} by keeping searching for another program $P' \in \mathbb{P}$ *s.t.* $\psi(P') \wedge |P'| < |P|$, ($|P|$ is the size of program P). In this paper, we use the number of tokens as the metric for measuring program sizes. This metric can abstract away details like identifier length and whitespace, which do not fundamentally affect the complexity of a program [25].

Additionally, using number of tokens as the metric is also widely adopted in prior closely related studies [25, 30, 39].

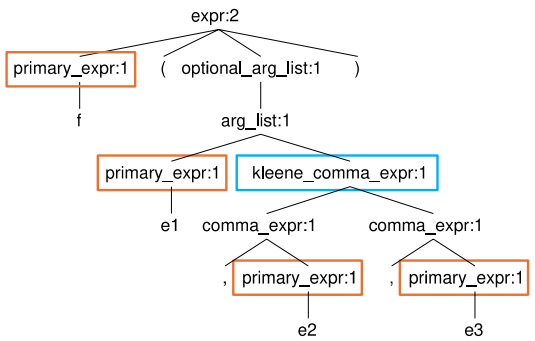
Canonicalization. In addition to minimizing bug-triggering programs, another important and highly desired feature of program reduction is canonicalization [10, 24]. Ideally, a perfectly canonicalizing reducer can reduce all duplicates (different programs that trigger the same bug) to an identical program. However, achieving perfect canonicalization is impractical [2, 10]. Therefore, a more attainable goal is often pursued: given a set of duplicates, the program reducer should minimize the differences among them as much as possible and reduce them to a minimal set of unique programs. This canonicalization capability is highly valuable in practice. For example, during a large-scale automated testing campaign for a language processor, a significant amount of bug-triggering programs may be generated. In such a case, a highly canonicalizing reducer could eliminate most duplicates and help address the fuzzer taming problem [2], *i.e.*, rank bug-triggering programs so that those triggering distinct bugs appear early in the list. From a debugging perspective, canonicalization can reduce the need for developers to examine numerous duplicates that vary only in unimportant ways, thereby simplifying and accelerating the debugging process [10].

Typical Workflow. Program reduction is typically a trial-and-error process. As shown in Figure 1, a program reducer first performs a transformation $f : \mathbb{P} \to \mathbb{P}$ to the original program P and obtains a different program P' (step ①). If P' does not trigger the same bug as P does (*i.e.*, $\psi(P') = \text{false}$), P' is discarded and the reducer will attempt to obtain another different program by performing another transformation and repeat the process (step ②). Otherwise, if $\psi(P') = \text{true}$, which means a smaller program is successfully found, the reduction continues by treating P' as the original program P and searching for the next smaller program (step ③).

2.2 Syntax of Programming Languages

```
1  expr := primary_expr kleene_comma_expr
2  expr := primary_expr
       LPAREN optional_arg_list RPAREN
3  kleene_comma_expr := comma_expr*
4  comma_expr := COMMA primary_expr
5  primary_expr := ID
6  primary_expr := LPAREN expr RPAREN
7  optional_arg_list := arg_list?
8  arg_list := primary_expr kleene_comma_expr
9  ID := [a-zA-Z_][a-zA-Z0-9_]*
10 LPAREN := '('
11 RPAREN := ')'
12 COMMA := ','
```

(a) An example CFG that defines a language of simple expressions. The bolded **expr** is the start symbol.



(b) An example parse tree that represents the expression `f(e1,e2,e3)` based on the grammar shown in Figure 2a. For simplicity, in the example, `e1`, `e2`, and `e3` are simply three variables.

Fig. 2. An example CFG and an example parse tree.

The syntax of a programming language is a set of rules that define the valid textual structure of programs written in this language. Typically, the syntax of a programming language can be formally expressed with a context-free grammar (CFG). A CFG G consists of four components: a finite set of nonterminal symbol N, a finite set of terminal symbol Σ, a finite set of production (or rewrite) rules R, and a start symbol $S \in N$, *i.e.*, $G = (N, \Sigma, R, S)$. The language defined by a

CFG G is called a context-free language (CFL), denoted as $L(G)$. Figure 2a shows an example CFG that formally defines a language of simple expressions, and Figure 2b shows the parse tree of an example expression, `f(e1,e2,e3)`, which conforms to the example CFG.

Nonterminal Symbol. A nonterminal symbol $v \in N$ represents a syntactic structure in the language defined by the corresponding CFG. Each non-leaf sub-tree st (a sub-tree whose height is not 0, also referred to as a structure in this paper) in the parse tree of a program corresponds to a nonterminal symbol v in the grammar of the language, denoted as $st \mapsto v$ in this paper. For example, in Figure 2b, sub-trees rooted at nodes marked with orange box correspond to the nonterminal symbol `primary_expr`. In Figure 2a, nonterminal symbols are all named with lowercase words while terminal symbols are uppercased.

Terminal Symbol. While nonterminal symbols can be rewritten to other symbol(s), a terminal symbol $\sigma \in \Sigma$ directly maps to character(s) (a token) that appears in the actual textual content and cannot be expanded to other symbols. Terminal symbol are either defined with constant character(s) or regular expressions. They form the alphabet of the language defined by the grammar.

Production Rule. A production rule $r \in R$ defines how a nonterminal symbol can be rewritten. Its left-hand side is the nonterminal symbol to be defined and the right-hand side is a sequence of symbols (either terminal or nonterminal symbols). Note that a nonterminal can have multiple different production rules. In the example grammar, there are eight production rules (line 1-8), and the nonterminal `expr` has two different production rules (*i.e.*, $|R_{\texttt{expr}}| = 2$), which means it can be rewritten in two different ways. Given a non-leaf sub-tree st and the nonterminal v that st corresponds to, we can further map st to a production rule $r \in R_v$ (denoted as $st \mapsto r$), indicating that the st is derived by following the production rule r. In Figure 2b, it can be observed that each non-leaf node is written in the format of `nonterminal:index`. The `index` after the colon is the index of the production rule to which the sub-tree corresponds. For example, the root of the entire parse tree is `expr:2`, indicating the parse tree corresponds to the second production rule of `expr`.

Start Symbol. The start symbol of the CFG is a special nonterminal symbol. All the strings that can be represented by the start symbol (*i.e.*, can be possibly derived by applying a sequence of production rules starting from the start symbol) form the language defined by the grammar. In the example grammar, `expr` is defined to be the start symbol.

Normal Forms of CFGs. The CFG can be written in various ways to represent a certain CFL. To impose specific restrictions thus facilitating certain parsing algorithms or theoretical analysis on top of the grammar, many normal forms of CFGs are proposed, such as Chomsky Normal Form (CNF) [3] and Greibach Normal Form (GNF) [9]. In this work, we make the CFG conform to Perses Normal Form (PNF), which is proposed and utilized by the previously proposed syntax-guided program reducer, Perses [30]. PNF requires that each production rule is in one of the following forms: (1) $A ::= B_1B_2...B_n$ (2) $A ::= B_1*$ (3) $A ::= B_1+$ (4) $A ::= B_1?$, and (5) $S ::= \epsilon$. Symbol A represents any nonterminal symbol, B_i ($\forall i \in [1, n]$) represents any nonterminal symbol or terminal symbol, and S represents the start symbol. The example grammar in Figure 2a conforms to PNF.

2.3 Syntax-Guided Transformation

Syntax-guided transformations are widely used in the process of program reduction to improve its effectiveness and efficiency [27, 30]. The insight behind this is that invalid programs are unlikely to trigger the bug that the original program reveals, and avoiding generating invalid programs can significantly shrink the search space of program reduction, thus saving considerable reduction time. The following definition formally defines syntax-guided transformations.

Definition 2.1 (Syntax-Guided Transformation)**.** *Assuming $\mathbb{P}_{valid}$ is the set of all syntactically valid programs, a transformation f is a syntax-guided transformation if and only if $\forall P \in \mathbb{P}_{valid}, f(P) \in \mathbb{P}_{valid}$.*

Two syntax-guided transformations are adopted by prior language-agnostic program reducer [30].
Compatible Substructure Hoisting: Given a tree, this transformation attempts to simplify the tree by hoisting one of its compatible descendants. A descendant is considered to be compatible if the nonterminal that the root corresponds to can be directly or indirectly rewritten by the nonterminal of the descendant. In Figure 2b, the entire parse tree represents `f(e1,e2,e3)` and corresponds to the nonterminal `expr`. Among all its descendants, four orange nodes corresponding to `primary_expr` are compatible, since according to line 1 in Figure 2a, `expr` can be rewritten by `primary_expr` (`kleene_comma_expr` can be empty). Thus, eventually, Compatible Substructure Hoisting attempts to simplify the original parse tree by reducing it to each four compatible descendant (`f`, `e1`, `e2`, and `e3`).
Quantified Node Reduction: This transformation aims to simplify trees that correspond to a production rule in the form of $A ::= B_1*$ or $A ::= B_1+$, or $A ::= B_1?$. For example, the sub-tree rooted at the blue node in Figure 2b corresponds to the production rule `kleene_comma_expr := comma_expr*` which is in the form of $A ::= B_1*$. For such a sub-tree, all its children can be deleted without breaking the syntax; thus Quantified Node Reduction attempts to simplify the tree by applying `ddmin` to its child list. For trees corresponding to production rules like $A ::= B_1+$, Quantified Node Reduction additionally ensures that at least one child remains, and for trees corresponding to $A ::= B_1?$, Quantified Node Reduction simply attempts to remove the single child.

3 Methodology

This section introduces Structure Form Conversion (SFC) (§ 3.1), how to effectively and efficiently apply SFC to the program reduction task (§ 3.2), and the implementation (§ 3.3)

3.1 Structure Form Conversion (SFC)

SFC converts a structure to another form. For example, given a function invocation expression `f(e1,e2,e3)` in a C program, SFC may convert it to `e1,e2,e3`. Such a conversion complements with Compatible Substructure Hoisting and Quantified Node Reduction and may create extra reduction opportunities.

Algorithm 1 describes the general idea of SFC. Given the formal syntax of the programming language G[3] and a sub-tree st in the parse tree of the program written in the specific language, SFC can produce a list of transformed sub-trees T that correspond to the same nonterminal as the original sub-tree st. Specifically, SFC first obtains the nonterminal v that the subtree st corresponds to, *i.e.*, $st \mapsto v$ (line 1). Next, SFC retrieves all the production rules of the nonterminal v in the grammar G (line 2). Then, for each production rule, SFC builds a list of new sub-trees (line 3-5), and eventually all the sub-trees built based on the production rules are returned (line 6).

Algorithm 1: The SFC Algorithm – `SFC(`st, G, `pred)`

Input : st, the sub-tree to be transformed. G, the formal syntax of the programming language. `pred`, an additional predicate for matching compatible sub-trees
Output : A list of transformed sub-trees T

1 $T \leftarrow [\,]$; $v \leftarrow$ the nonterminal symbol that st maps to, *i.e.*, $st \mapsto v$
2 $R_v \leftarrow$ the production rules for v
3 **for** $i \in [0, R_v$`.size() - 1]` **do**
4 **if** $st \mapsto R_v[i]$ **then continue** // skip the production rule that the original sub-tree corresponds to
5 $T_{R_v[i]} \leftarrow$ `buildSubTreesForTheRule(`$R_v[i]$, st, `pred);` T`.addAll(`$T_{R_v[i]}$`)`
6 **return** T

[3]In the algorithm, it is assumed that the grammar is in Perses Normal Form (PNF).

3.1.1 Design Principles. The most essential and challenging part of SFC is the algorithm for building sub-trees based on a given production rule. Making this algorithm effective and efficient for program reduction faces two main challenges. First, given a production rule, there is often an infinite (or at least extremely large) number of valid sub-trees that can be generated. Therefore, the algorithm needs to be properly guided and restricted. Otherwise, it will be too expensive to be feasible. Second, randomly generated sub-trees may render the semantics of the program invalid even though they preserve the syntactic validity. Although previous approaches like Perses faces the same problem as they cannot ensure semantic validity during the reduction, randomly generating sub-trees can amplify this problem and thus significantly impair the reduction performance. For example, a randomly generated expression is very likely to contain a random identifier which is not defined in the program. In such a case, the program after the replacement is not likely to trigger the same bug as the language processor probably terminates when it detect this semantic error in an early stage.

To overcome the aforementioned two challenges, we propose five principles to restrict the sub-tree building algorithm. The insight behind all the principles is that the fewer changes are made, the more likely the bug-triggering property can be preserved. In other words, instead of aggressively simplifying the problem in a single step, gradually reducing it through small steps is more likely to succeed, especially since SFC is designed to operate on programs that have already been reduced to a relatively small size. Therefore, the essence of these principles is to avoid making excessive and unnecessary changes to the original sub-trees.

Principle 1: *Do not randomly generate any structure. Reuse the structures in the original sub-tree instead.* This principle is to avoid generating numerous sub-trees that are unlikely to replace the original sub-tree without making the program no longer trigger the same bug. For example, given a sub-tree that represents `f(e1,e2,e3)`, generating a new expression with some random identifiers is usually meaningless as such an expression would probably lead to an undefined identifier error.

Principle 2: *A structure in the original sub-tree can only be reused once.* This principle further reduces the number of sub-trees that can be possibly generated. Meanwhile, it can also reduce the difference between the generated sub-tree and the original sub-tree, thus increasing the possibility that replacing the original sub-tree with the new one makes the program still trigger the same bug.

Principle 3: *The reused structures should remain the same order as they appear in the original sub-tree.* Similar to principle 2, this principle narrows down the search space of the algorithm and makes the generated sub-trees be more similar to the original sub-tree.

Principle 4: *The reused structure should be as high-level as possible.* This principle aims to avoid generating sub-trees that are aggressively simplified. Meanwhile, adhering to this principle makes SFC more orthogonal to Compatible Substructure Hoisting. It makes SFC focus on converting the structure to another form and delegates the simplification task to the subsequent Compatible Substructure Hoisting.

Principle 5: *Reuse as many structures as possible.* When the input production rule is in the form of $A ::= B_1*$ or $A ::= B_1+$, the algorithm needs to decide the number of structures that correspond to B_1 the generated sub-trees should contain. This principle dictates the algorithm to maximize this number while not breaking other principles. Similar to principle 4, this principle can avoid generating aggressively simplified sub-trees. Meanwhile, it makes SFC more orthogonal to Quantified Node Reduction.

3.1.2 Sub-Tree Building. Algorithm 2 describes the sub-tree building algorithm of SFC. This algorithm takes three inputs, *i.e.*, a sub-tree to be transformed st, a production rule r, and an additional predicate for matching sub-trees (this input is for identifier elimination and will be discussed in § 3.2.2). The output of the algorithm is a list of sub-trees that are generated based on the input production rule r. Recall that in PNF, the right hand side of a production rule can be a sequence

Algorithm 2: The Sub-Tree Building Algorithm – `buildSubTreesForTheRule(r, st, pred)`

```
   Input  : st, the sub-tree to be transformed.  r, the production rule based on which the sub-tree is built.
            pred, an additional predicate for matching compatible sub-trees
   Output : A list of built sub-trees T
1  T ← []
2  if The form of the production rule r is A ::= B1* or A ::= B1+ or A ::= B1? then
3      s ← the symbol in the right hand side of the production rule
4      compatibleTrees ← findCompatibleSubTrees(s, st, pred)
5      if compatibleTrees.size() == 0 then return []
6      if s is followed by an optional quantifier (?) then
7          foreach ct ∈ compatibleTrees do
8              if ∃ct' ∈ compatibleTrees, s.t. ct is a descendant of ct' then continue // Principle 4
9              t' ← a subtree built based on r by resuing ct; T.append(t')
10     else // s is followed by a Kleene Star (*) or a Kleene Plus (+)
11         t' ← a subtree built based on r by reusing all highest-level compatible sub-trees in compatibleTrees
               T.append(t')
12 else // The form of the production rule r is A ::= B1B2...Bn
13     symbolList ← all the symbols in the right hand side of the production rule r
14     uniqueSymbols ← symbolList.toSet()
15     symbolToCompatibleMap ← an empty map
16     foreach s in uniqueSymbols do
17         if s is a terminal and is defined with fixed text then continue // No need to match for constants
18         else
19             compatibleTrees ← findCompatibleSubTrees(s, st, pred)
20             n ← the number of symbol s in symbolList
21             if n > compatibleTrees.size() then return []
22             symbolToCompatibleMap[s] ← all n-combinations of compatibleTrees
       // symbolToCompatibleMap now records all combinations of compatible subtrees for each unique symbol
23     reusings ← the cartesian product of symbolToCompatibleMap[s] of each unique symbol s
24     adoptedReusings ← all elements of reusings in which all subtrees are independent and as high-level as
           possible // Adhere to principle 2 and 4
25     foreach reusing in adoptedReusings do
26         t' ← a subtree built based on r by resuing subtrees in reusing; T.append(t')
27 return T
```

of symbols or a single symbol followed by a kleene star (*), or a kleene plus (+), or an optional quantifier (?). [4] For different forms of production rules, the way to build sub-trees are different.

Building for Production Rules with a Quantified Symbol. When the right hand side of the production rule r is a single quantified symbol, the algorithm first extracts the single symbol s (line 3), and then searches for all the compatible sub-trees in st, *i.e.*, the tree to be transformed (line 4). If no compatible st is found, meaning no sub-tree can be reused for the building, the algorithm simply returns an empty list, indicating no sub-tree can be built (line 5). Otherwise, the algorithm builds sub-trees according to the type of quantifier that follows the symbol s. If the symbol s is followed by an optional quantifier (?), the algorithm traverses the found compatible sub-trees. For each compatible sub-tree, if it is at the highest level [5] (line 8), a transformed sub-tree

[4] The special form $S ::= \epsilon$ for including empty string in the defined language is omitted here.

[5] A compatible node is at the highest level if none of its ancestors is compatible. According to the fourth aforementioned design principle, the algorithm is designed to prefer reusing node as high-level as possible.

is built by simply reusing the compatible sub-tree (line 6-9). If the symbol s is followed by an Kleene star (*) or Kleene plus (+), the algorithm only builds one transformed sub-tree by reusing all the highest-level compatible sub-trees [5] (line 10-11).

Builidng for Production Rules with a Sequence of Symbols. If the production rule r is in the form of $A ::= B_1B_2...B_n$, the algorithm needs to find a compatible sub-tree for each symbol in the right hand side of r (except terminal symbols with constant text such as `COMMA` and `LPAREN`) to reuse. In other words, a sequence of compatible sub-trees with which all the symbols in the right hand side of r can have a compatible sub-trees to reuse (such a sequence is referred to as a matching for brevity) is required to build one transformed sub-tree. To illustrate the algorithm in this scenario, we describe the steps with a running example. We use the parse tree shown in Figure 2b as the input of the running example, let the input production rule be `expr := primary_expr kleene_comma_expr` (line 1 in Figure 2a), and suppose there is no additional predicate.

Initialization: First, the algorithm initializes symbolList with all the symbols on the right hand side of the production rule r (line 13), and extracts all the unique symbols to a set uniqueSymbols (line 14). In the running example, both symbolList and uniqueSymbols contain `primary_expr` and `kleene_comma_expr`. It should be noted that in practice, a symbol can appear multiple times in the right hand side of a production rule. Next, symbolToCompatibleMap, a map that maps each unique symbol to all the possible matchings, is initialized with an empty map (line 15).

Searching for Compatible Sub-Trees: The algorithm traverses uniqueSymbols. For each unique symbol s, if it is a terminal symbol and is defined with constant text, then no compatible sub-tree is needed and the algorithm moves to the next unique symbol (line 17). Otherwise, the algorithm searches for all the compatible sub-trees for the symbol s (line 19). In the running example, the symbol `primary_expr` has four compatible sub-trees, whose roots are marked with orange and the symbol `kleene_comma_expr` has one compatible sub-tree, whose root is marked with blue.

Obtaining Matchings for Each Unique Symbol: Next, the number of appearances n of this unique symbol is retrieved (line 20). Recall that to build a transformed sub-tree, each symbol needs a compatible sub-tree to reuse. Therefore, the number n is also the required number of sub-trees to form a matching for the unique symbol. If the total number of found compatible sub-trees is smaller than n, the algorithm returns an empty list, meaning no transformed sub-tree can be generated (line 21). Otherwise, all n-combinations of the found compatible trees are store as matchings in symbolToCompatibleMap with the corresponding unique symbol s as the key. In the running example, both unique symbols only appear once in the production rule. Thus, the number n for both unique symbols equals one, and the matchings for these two unique symbols are the 1-combinations of their compatible sub-trees. Specifically, there are four matchings for the unique symbol `primary_expr` and one matching for `kleene_comma_expr`.

Obtaining Matchings for the Entire Production Rule: Once the matchings of each unique symbol is found, the algorithm derives all the matchings for the entire right hand side of r by calculating the Cartesian product of the matchings of each unique symbol (line 23). In the running example, `primary_expr` has four matchings. Each of them contains one sub-tree that represents `'f'`, `'e1'`, `'e2'`, `'e3'`, respectively. `kleeene_comma_expr` has one matching containing one sub-tree representing `',e2,e3'`. Therefore the resulted Cartesian product contains four matchings representing `'f,e2,e3'`, `'e1,e2,e3'`, `'e2,e2,e3'`, `'e3,e2,e3'`. However, the last two matchings violate the second design principle (they reuse `'e2'` and `'e3'` twice, respectively), the algorithm filters such matchings by excluding all matchings that contain dependant compatible sub-trees (line 24). Finally, all the remaining matchings are utilized to build transformed sub-trees (line 25-26).

3.2 Program Reduction with SFC

Unlike Compatible Substructure Hoisting and Quantified Node Reduction, SFC does not ensure that the resulted structure is smaller than the original structure. Meanwhile, compared to the previous two syntax-guided transformation, the number of structures that can be possibly generated by SFC is usually much larger. Therefore, to effectively and efficiently adapt SFC to the program reduction task, we further propose three reduction methods, *i.e.*, Smaller Structure Replacement, Identifier Elimination via SFC and Structure Canonicalization. This section details how these reduction methods utilize SFC to boost program reduction.

3.2.1 Smaller Structure Replacement. This reduction method aims to reduce a program by replacing structures of the program to their smaller forms. It traverses the structures at different levels in the parse tree of the program, invokes the SFC algorithm to generate structures in different forms that are smaller, and attempts to replace the original structure with the newly generated structure. If the program after the replacement still triggers the same bug, the program is successfully reduced.

Algorithm 3: Smaller Structure Replacement– `reduceBySimplifyingStructure(`t, G`)`

```
   Input  : P, the program to be reduced.  G, the formal syntax of the programming language.
            ψ, the oracle that verifies whether a program triggers the specific bug
   Output : The reduced program P'
1  t ← the parse tree of P;  truePred ← a predicate that always returns true
2  queue ← a queue that only contains the root node of t
3  while queue is not empty do
4      root ← queue.poll();  st ← the sub-tree rooted at root
5      T ← the result of SFC(st, G, truePred) sorted in ascending order
6      for st' ∈ T do
7          if the number of leaves in st' is not smaller than that of st then break
8          t' ← a new parse tree obtained by replacing st with st'
9          P' ← the program derived from t'
10         if ψ(P') then t ← t'; root ← the root of st'; break
11     queue.addAll(root.children)
12 return the program derived by t
```

Algorithm 3 describes this reduction method. Given a bug-triggering program to be reduced P, the formal syntax of the language in which P is written, and a oracle that can check whether a program triggers the specific bug, the algorithm can produce a reduced program P' by simplifying structures in the program in a breadth-first manner. Specifically, at the beginning, the algorithm obtains the parse tree of the input program P, initializes a predicate that always returns true (required by SFC, meaning no additional criterion for compatible sub-trees), and adds the root of the parse tree to a queue (line 1-2). Next, the algorithm starts simplifying structures in a while loop (line 3-11). First, the node at the front of the queue and the sub-tree rooted at this node are retrieved (line 4). Next, the algorithm invokes SFC to obtain a list of transformed sub-trees and sorts the sub-tree list based on the number of leaves in ascending order (line 5). Then, in a for loop (line 6-10), the algorithm attempts to replace the original sub-tree st with each transformed sub-tree st' until the transformed sub-tree st' is not smaller than the original sub-tree st (line 7) or the program after replacement still triggers the bug (line 10), which means a structure is successfully simplified. After the simplification performed in the for loop, the children of the original sub-tree st (or the transformed sub-tree st' if the simplification succeeded) are added to the queue (line 11), and the algorithm repeats the process to keep simplifying structures in the program.

Algorithm 4: Identifier Elimination– `reduceByEliminatingIdentifers(`t, G`)`

```
Input  : P: the program to be reduced.  G: the formal syntax of the programming language.
         ψ: the oracle that verifies whether a program triggers the specific bug
Output : The transformed program P'
1  allIdentifiers ← all identifier tokens in program P
2  nameToIdTokensMap ← a map that maps each unique identifier name to the corresponding identifier tokens
3  foreach name ∈ nameToIdTokensMap.keys() do
4      useIds ← all the uses of name (i.e., identifier tokens that are not for ID definition or initialization)
5      foreach anotherName ∈ nameToIdTokensMap.keys()\name do
6          P' ← a new program obtained by renaming all useIds with anotherName
7          if ψ(P') then return P'
8      foreach use ∈ useIds do // Eliminate each use by either SFC or replacing
9          foreach anotherName ∈ nameToIdTokensMap.keys()\name do
10             P' ← a new program obtained by renaming use with anotherName
11             if ψ(P') then update P with P' and go to line 8 to eliminate the next use
12         success ← eliminateBySFC(use, P, ψ)
13         if ¬success then go to line 3 and move to next unique identifier name // Fail to eliminate
14 return P
15 Function eliminateBySFC(use, P, ψ):
16     parent ← use; pred ← a predicate that takes a sub-tree as input and returns true only if the sub-tree does not
         contain use
17     while parent has a parent node do
18         parent ← the parent node of parent
19         T ← the result of SFC(st, G, pred) sorted in ascending order
20         for st' ∈ T do
21             if the number of leaves in st' is not smaller than that of st then break
22             P' ← a program obtained by replacing the sub-tree rooted at parent with st'
23             if ψ(P') then update P with P'; return true
```

3.2.2 Identifier Elimination. In previous research work, eliminating the uses of an identifier has been demonstrated to be effective for further simplifying a bug-triggering program, since it can potentially make some definitions or initializations of identifiers redundant, thus being deletable [39]. However, previous approaches of identifier elimination only relies on replacing (*i.e.*, replacing all the uses of an identifier with another identifier). The reduction method proposed in this section aims at additionally utilizing SFC to eliminate uses of an identifier for better effectiveness. Specifically, given a use of an identifier, this approach replaces a structure that contains the use with another structure generated by SFC. During the generation, any sub-tree that contains the given use is forbidden to be reused; thus, after the replacement, the use of the identifier is eliminated.

Algorithm 4 describes the complete reduction method in detail. First, given a program to be reduced P, all the identifier tokens in P are collected and clustered by their names (line 1-2). To achieve this, the terminal symbol (*e.g.*, `ID` in Figure 2a) that represents identifier tokens in the corresponding grammar G must be specified. Although this requirement introduces some degree of language specificity, we believe it does not compromise the generality of our approach, as it can typically be addressed in a straightforward manner. Next, the algorithm traverses each unique identifier and attempts to eliminate all the uses for the specific identifier (line 20-23). To eliminate all the uses, the algorithm first attempts to rename all of them at once with another unique identifier (line 5-7). If the elimination is successful, the program after elimination is returned for further reduction. Otherwise, the algorithm attempts to eliminate the uses one by one (line 8-13). To do

so, the algorithm traverses all the uses, and for each use, it first attempts to rename it in a similar way as renaming all the uses at once (line 9-11). If the attempt fails, the function `eliminateBySFC` is invoked (line 19). This function takes as input a use (*i.e.*, an identifier token), the program P, and the oracle ψ, and then it attempts to eliminate the use by replacing the sub-tree that contains the use with a new sub-tree without the use generated by SFC. Specifically, the function first initializes a predicate `pred` which takes as input a sub-tree and only return `true` when the sub-tree does not contains the input use (line 16). Next, in a while loop, starting from the parent of the leaf node representing the use the function traverses the ancestors of the use from bottom to the top (line 17-23). For the sub-tree rooted at each ancestor node, a list of sub-trees are generated by invoking SFC (line 19). It should be noted that the predicate `pred` ensures that the generated sub-trees do not contains the use. Finally, for each generated sub-tree st' that is smaller than the original sub-tree st, the algorithm attempts to replace st with st' (line 20-22). If the replacing is successful (*i.e.*, the new program P' still triggers the same bug), P is updated by P' and the function immediately returns `true`, indicating the elimination is successful (line 23).

3.2.3 Structure Canonicalization. In addition to minimizing bug-triggering inputs, another desired feature of program reduction is canonicalizing inputs that trigger the same bug. This canonicalization feature can significantly facilitate bug deduplication and help solve the fuzzer taming problem [2], thus being highly demanded in practice. To boost the canonicalization aspect of language-agnostic program reducers, we propose a method named Structure Canonicalization. Its workflow is similar to Smaller Structure Replacement in Algorithm 3. While Smaller Structure Replacement attempts to replace the original structure with a smaller one, Structure Canonicalization attempts to replace the original structure with a structure having the same size but being more canonical. To decide which structure is more canonical, we follow the definition below.

Definition 3.1. *Given a nonterminal v and two sub-trees $st \mapsto v$ and $st' \mapsto v$. The subtree st is more canonical than the subtree st' if and only if $st \mapsto r$, $st' \mapsto r'$, and the index of r is smaller than that of r' in R_v.*

For example, suppose we have two different production rules for the nonterminal `expr`, ordered as follows: ① `expr:=ID+ID`; ② `expr:=ID.ID`. In this case, we consider `a+b` to be more canonical than `a.b`. If the order were reversed, then a.b would be considered more canonical. The purpose here is that if both a+b and a.b can trigger the same bug, we want to canonicalize them to a single, canonical form. In our implementation, the production rules are ordered based on their appearance in the Antlr4 grammar file.

The algorithm of Structure Canonicalization can be obtained by adjusting the smaller structure replacement algorithm (Algorithm 3) and the SFC algorithm (Algorithm 1) it invokes. First, `continue` on line 4 in SFC needs to be revised to `break`, so that SFC only generate more canonical sub-trees. Next, line 5 in Algorithm 3 shall invoke the revised SFC instead and filter out sub-trees having different size from the original sub-tree st. Finally, line 7 in Algorithm 3 needs to be removed.

3.3 Implementation

We implemented two prototypes, $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ on top of two previous language-agnostic program reducers, Perses and Vulcan, respectively. Figure 3 shows the workflow of $\text{SFC}_{\text{Perses}}$, which contains a main reducer Perses and three auxiliary reducers. The main reducer, Perses, is responsible for reducing the input efficiently. The three auxiliary are implemented based on the proposed three SFC-based reduction method and they are supposed to either aggressively search for reduction opportunities and/or create reduction opportunities for the main reducer. The architecture of $\text{SFC}_{\text{Vulcan}}$ is similar to $\text{SFC}_{\text{Perses}}$ but uses Vulcan as main reducer. Compared to $\text{SFC}_{\text{Perses}}$, $\text{SFC}_{\text{Vulcan}}$ additionally incorporates the auxiliary reducers proposed by Vulcan.

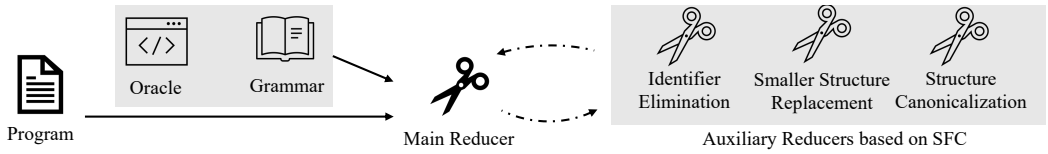


Fig. 3. The overall workflow of $\text{SFC}_{\text{Perses}}$

Both $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ take three inputs, *i.e.*, a bug-triggering program, the grammar in which the program is written, and an oracle that verifies whether a program triggers the specific bug. The input program is first reduced by the main reducer. Once the main reducer cannot make any progress, the first auxiliary reducer is invoked to further reduce the program and/or create reduction opportunities by performing certain transformations. If the auxiliary makes progress, the program is send back to the main reducer, otherwise, the next auxiliary reducer is invoked. For example, if Identifier Elimination successfully eliminates an identifier, the program will be directly sent back to the main reducer. Otherwise, if Identifier Elimination cannot eliminate any identifier, the program will be sent to the next auxiliary reducer, *i.e.*, Smaller Structure Replacement. This alternating process is repeated until the last auxiliary reducer cannot make any progress, and it retains the same theoretical guarantee (e.g., 1-minimality) as the main reducer.

4 Evaluation

To evaluate the proposed SFC-based reduction methods, we conducted experiments to investigate the following research questions.
RQ1: Can SFC benefit previous language-agnostic program reducers in minimization?
RQ2: Can SFC benefit previous language-agnostic program reducers in canonicalization?
RQ3: To what extent does each SFC-based method contribute to the performance improvement?

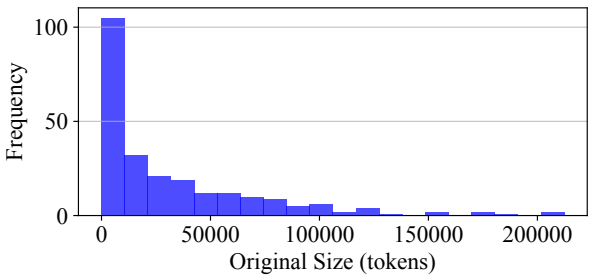


Fig. 4. Distribution of the original program sizes in `Benchmark-Reduce`.

Benchmark Suites. We use two benchmark suites, `Benchmark-Reduce` and `Benchmark-Cano`, to assess the performance in minimization and canonicalization for each reducer. Specifically, `Benchmark-Reduce` contains 20 C benchmarks, 20 Rust benchmarks, and 205 SMT-LIBv2 benchmarks. Each benchmark consists of a program that triggers a unique real-world crash or miscompilation bug and a shell script that checks whether a program triggers the specific bug. The original sizes of these bug-triggering programs range from 15 to 212,259 tokens, and the distribution of the original sizes is shown in Figure 4. We utilize this benchmark suite for assessing the minimization performance of reducers. These benchmarks are also widely used for evaluation in previous studies

in this direction [8, 19, 30, 33, 38, 39, 43, 45]. `Benchmark-Cano` contains 3,796 bug-triggering C programs. 2,501 of them trigger 11 unique crash bugs in GCC 4.3.0, and the remaining 1,295 programs trigger 35 miscompilation bugs also in GCC 4.3.0. This benchmark suite is used for evaluating bug deduplication performance in previous studies [2, 40]. In our evaluation, we use this suite mainly for evaluating the canonicalization performance of reducers.

Baselines. To evaluate $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$, we compare them against the corresponding previous language-agnostic program reduction tools, Perses [30] and Vulcan [39], and two program reducers that are specifically optimized for C/C++ and SMT-LIBv2, *i.e.*, C-Reduce [28] and ddSMT [19].

- *Perses* is the first syntax-guided language-agnostic program reducer. It is implemented with two syntax-guided transformations, Compatible Substructure Hoisting and Quantified Node Reduction, to achieve efficient program reduction.
- *Vulcan* is a language-agnostic program reduction framework that aims to trade off the efficiency for a smaller reduction result. It deploys Perses as the main reducer and includes three auxiliary reducers to search for reduction opportunities when Perses cannot make any progress.
- *T-Rec* is a fine-grained language-agnostic program reduction technique that mainly aims to improve canonicalization by searching for reduction opportunities within tokens [38]. We included it as a baseline when evaluating the canonicalization performance. Unlike Perses, Vulcan, and SFC—which treat tokens as atomic elements—T-Rec is capable of reducing and canonicalizing token content. For example, it can reduce a random string literal to an empty string or a numeric constant to 0. We chose to include T-Rec in our evaluation of canonicalization for two reasons. First, T-Rec also aims to improve canonicalization. By integrating it, we demonstrate that SFC is complementary to T-Rec and that combining them can further enhance canonicalization performance. Second, integrating T-Rec simplifies the evaluation. Although SFC improves canonicalization by aligning program structure, few duplicates are reduced to exactly identical programs because Perses, Vulcan, and SFC do not modify characters within token contents. As a result, structurally similar programs may still differ in unimportant literals. T-Rec addresses this by canonicalizing tokens, thereby enabling more duplicates to be textually identical, which simplifies downstream deduplication tasks. T-Rec is implemented as an additional auxiliary reducer, integrated by appending it to the list of auxiliary reducers.
- *C-Reduce* and *ddSMT* are program reducers specifically optimized for reducing C/C++ and SMT-LIBv2 programs, respectively. By comparing our approach with C-Reduce and ddSMT, we aim to assess the extent to which SFC narrows the gap between language-agnostic and language-specific reducers. Since C-Reduce also includes certain language-agnostic reduction components, it can be applied to programs in other languages as well. Therefore, we include C-Reduce as a baseline for the Rust benchmarks. However, we exclude C-Reduce when evaluating on SMT-LIBv2 benchmarks, as we observed that it often fails to reduce these programs effectively within a reasonable time.

Metrics. We use two metrics in our evaluation. To assess the minimization capability of a reducer, we define the **Size** of a program by the number of tokens it contains. A smaller output indicates a stronger minimization capability. We use **Number of Eliminated Duplicates** to measure the performance of canonicalization. Given a list of duplicates, assuming the number of distinct programs that the list contains is n. Suppose that after reduction the number of distinct programs becomes m. Then the difference (*i.e.*, $n - m$) is the number of eliminated duplicates.

A larger difference indicates that more duplicates become identical after reduction (thus can be eliminated), demonstrating a stronger canonicalization capability.

Experiment Setup. The experiments are run on an Ubuntu 20.04 server with AMD 7950X CPU and 128 GB RAM. For fair comparisons, all the experiments are run with a single thread.

4.1 RQ1: Performance in Minimization

To investigate this research question, we compare $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ to Perses and Vulcan on `Benchmark-Reduce`, respectively. Figure 5, Figure 6, and Figure 7 show the results of the experiment. For the C and Rust benchmarks, detailed results are also listed in Table 1 and Table 2.

$\text{SFC}_{\text{Perses}}$ vs. Perses. As shown in Table 1 and Figure 5-7, the results produced by $\text{SFC}_{\text{Perses}}$ are generally and significantly smaller than those of Perses, indicating that integrating SFC-based reduction methods can effectively boost the minimization performance of Perses. Specifically, in terms of size, the results of $\text{SFC}_{\text{Perses}}$ are 36.82%, 18.71%, and 41.05% smaller than those of Perses on average on C, Rust, and SMT-LIBv2 benchmarks, respectively. In terms of execution time, incorporating SFC-based methods inevitably introduces overhead. On average, $\text{SFC}_{\text{Perses}}$ takes 3.65×, 16.99×, and 3.97× the time of Perses on C, Rust, and SMT-LIBv2 benchmarks. The unusually high time ratio on the Rust benchmarks is primarily due to the presence of many small benchmarks, which inflates the average. For instance, when considering only the 10 smallest benchmarks from `rust-77320` to `rust-78720`, the time ratios of these benchmarks range from 11 to 64, averaging 26.46. However, the absolute time differences for these benchmarks range from 82s to 628s, averaging 282.8s—less dramatic than the ratios suggest.

$\text{SFC}_{\text{Perses}}$ vs. Vulcan. From Table 1 and Figure 5-7, it can be observed that the overall performance of $\text{SFC}_{\text{Perses}}$ and Vulcan is comparable. Specifically, the results of $\text{SFC}_{\text{Perses}}$ are 1.86%, 3.34%, and 3.34% smaller than those of Vulcan. The p-values yielded by the Wilcoxon signed-rank test are 0.105, 0.530, and 0.598, respectively, which indicate that there is no statistically significant difference between the two reducers in terms of minimization performance. Time-wise, $\text{SFC}_{\text{Perses}}$ takes 1.19×, 1.33×, and 0.95× the time of Vulcan on C, Rust, and SMT-LIBv2 benchmarks, and the corresponding p-values are 0.04, 0.812, and 1.256×10^{-6}, respectively. These results indicate that statistically, $\text{SFC}_{\text{Perses}}$ is significantly slower than Vulcan on C benchmarks, significantly faster on SMT-LIBv2 benchmarks, and not significantly different on the Rust benchmarks. Although $\text{SFC}_{\text{Perses}}$ and Vulcan achieve comparable overall minimization performance, it is worth noting that they are complementary to some extent. Specifically, in 53 out of 245 benchmarks, $\text{SFC}_{\text{Perses}}$ produces results that are more than 10% smaller than Vulcan, while in 48 benchmarks, it produces results that are more than 10% larger. Furthermore, our comparison of $\text{SFC}_{\text{Vulcan}}$ with Vulcan demonstrates that integrating SFC-based reduction methods into Vulcan leads to further improvements in minimization.

$\text{SFC}_{\text{Vulcan}}$ vs. Vulcan vs. C-Reduce and ddSMT. As illustrated in Table 2 and Figure 5-7, $\text{SFC}_{\text{Vulcan}}$ significantly outperforms Vulcan in terms of minimization. On average, the results of $\text{SFC}_{\text{Vulcan}}$ are 14.51%, 7.65%, and 7.66% smaller than those of Vulcan on C, Rust, and SMT-LIBv2 benchmarks, respectively. However, on C benchmarks and SMT-LIBv2 benchmarks, $\text{SFC}_{\text{Vulcan}}$ does not outperform the corresponding language-specific reducers, *i.e.*, C-Reduce and ddSMT, which is expected. Specifically, $\text{SFC}_{\text{Vulcan}}$ produces 50.39% and 7.55% larger results than C-Reduce and ddSMT on C and SMT-LIBv2 benchmarks, respectively. In contrast, the results of Vulcan are 80.38% and 16.22% larger than those of C-Reduce and ddSMT, showing that $\text{SFC}_{\text{Vulcan}}$ closes a substantial portion of the gap. For the Rust benchmarks, although C-Reduce is also effective at reducing Rust programs, $\text{SFC}_{\text{Vulcan}}$ achieves better results—producing outputs that are, on average, 11.58% smaller than those of C-Reduce. In terms of execution time, $\text{SFC}_{\text{Vulcan}}$ takes 1.56×, 2.35×, and 1.42× the time of Vulcan on C, Rust, and SMT-LIBv2 benchmarks. Compared to language-specific reducers,

SFC_{Vulcan} takes 1.85× and 3.22× the time of C-Reduce on C and Rust benchmarks, and 6.04× the time of ddSMT on SMT-LIBv2 benchmarks.

Trade-off between Time Overhead and Reduction. As shown in the results discussed above, SFC introduces a notable time overhead. However, it should be noted that the reducer always maintains a minimal result during the reduction process, allowing users to stop at any point that they deem the result sufficiently simplified. The objective of SFC is not to replace Perses or Vulcan, but rather to offer an effective and efficient method for users to further trade execution time for a smaller result when needed. Given that minimizing bug-triggering programs is not always time-sensitive, and manual inspection and minimization can be labor-intensive—demanding specific language and compiler expertise—we believe providing the option to trade additional CPU time for a smaller program to initiate manual analysis is beneficial.

Example Simplifications Performed by SFC. To provide insights on how SFC-based reduction methods enable further reduction, we manually inspect some of the C benchmarks and summarize some example simplifications performed by the SFC-based reduction methods shown below (in the examples, `ei` represents an expression, `si` represents a statement, and `a` and `b` are identifiers):

```
(1) e1 && e2 ==> !e1
(2) e1 < e2 ==> !e1
(3) e1.e2 ==> &e1
(4) e1.e2 ==> sizeof e1
(5) e1[e2] ==> *e1
(6) e1[e2].[e3] ==> e1->e3
(7) e1.e2 = e3 ==> ++e1.e2
(8) f(e1, e2) ==> --e1
(9) s1 ==> ;
(10) if (a=b); ==> a=b;
(11) for(;a;a++) ==> while(a++)
(12) for(;;) { s1 s2 } ==> for(s1;;s2)
```

RQ1: While SFC-based reduction methods introduce some time overhead, they significantly boost the minimization capabilities of Perses and Vulcan. Specifically, the results of SFC_{Perses} are 36.82%, 18.71%, and 41.05% smaller than those of Perses on C, Rust, and SMT-LIBv2 benchmarks. As for SFC_{Vulcan}, the corresponding percentage decreases are 14.51%, 7.65%, and 7.66%, respectively.

4.2 RQ2: Performance in Canonicalization

To evaluate whether SFC-based reduction methods can boost performance in canonicalization, we run Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} to reduce all the bug-triggering programs in `Benchmark-Cano`. As mentioned previously, to demonstrate that SFC-based reduction methods are complementary to T-Rec, another technique for improving canonicalization performance, we integrate T-Rec to each reducer in the experiment.

The evaluation results are shown in Table 3 ($SFC_{Perses^{-sc}}$ is a variant for the ablation study covered in § 4.3, where its results are discussed). From the table, it can be observed that compared to Perses, SFC_{Perses} can eliminate 439 more crash duplicates and 3 more miscompilation duplicates. Meanwhile, compared to Vulcan, SFC_{Vulcan} eliminates 425 and 10 more crash and miscompilation duplicates, respectively. Such results indicate that SFC-based reduction methods significantly improve the canonicalization capabilities of both Perses and Vulcan. Time-wise, the average execution time of SFC_{Perses} is 1.69× and 3.08× the time of Perses on the crash and miscompilation benchmarks, respectively. As for SFC_{Vulcan}, its average time is 1.06× and 1.35× the time of Vulcan, respectively.

Table 1. Evaluation results of Perses, Vulcan, and SFC_{Perses} on C and Rust benchmarks. Column %Δ shows the average **percentage change in size** achieved compared to the baseline reducer.

Benchmarks	Perses		Vulcan		SFC_{Perses}					
	Time (s)	Size (#)	Time (s)	Size (#)	Time (s)	Size (#)	%Δ in size *w.r.t.* Perses	Time ratio *w.r.t.* Perses	%Δ in size *w.r.t.* Vulcan	Time ratio *w.r.t.* Vulcan
clang-22382	508	144	1,058	108	1,114	113	-21.53%	2.19	4.63%	1.05
clang-22704	1,376	78	1,574	62	1,650	61	-21.79%	1.20	-1.61%	1.05
clang-23309	1,619	464	6,802	303	7,748	227	-51.08%	4.79	-25.08%	1.14
clang-23353	750	98	1,025	91	1,392	82	-16.33%	1.86	-9.89%	1.36
clang-25900	915	239	1,676	104	3,136	193	-19.25%	3.43	85.58%	1.87
clang-26760	1,718	120	2,384	56	2,263	52	-56.67%	1.32	-7.14%	0.95
clang-27137	7,391	180	9,962	88	13,137	83	-53.89%	1.78	-5.68%	1.32
clang-27747	1,013	117	1,759	79	2,149	83	-29.06%	2.12	5.06%	1.22
clang-31259	1,719	406	11,695	282	13,375	249	-38.67%	7.78	-11.70%	1.14
gcc-59903	3,291	308	4,743	198	6,236	158	-48.70%	1.89	-20.20%	1.31
gcc-60116	2,200	443	6,945	247	9,373	211	-52.37%	4.26	-14.57%	1.35
gcc-61383	2,128	272	10,766	195	17,711	183	-32.72%	8.32	-6.15%	1.65
gcc-61917	1,064	150	1,650	103	1,865	70	-53.33%	1.75	-32.04%	1.13
gcc-64990	2,508	239	4,416	203	5,634	165	-30.96%	2.25	-18.72%	1.28
gcc-65383	827	153	1,531	84	5,523	119	-22.22%	6.68	41.67%	3.61
gcc-66186	2,158	327	16,992	226	10,035	221	-32.42%	4.65	-2.21%	0.59
gcc-66375	2,409	440	11,656	227	16,719	289	-34.32%	6.94	27.31%	1.43
gcc-70127	2,726	301	15,053	230	14,502	188	-37.54%	5.32	-18.26%	0.96
gcc-70586	5,496	157	17,335	94	7,717	75	-52.23%	1.40	-20.21%	0.45
gcc-71626	45	51	123	38	140	35	-31.37%	3.11	-7.89%	1.14
Mean	2,093.05	234.35	6,457.25	150.90	7,070.95	142.85	-36.82%	3.02	-1.86%	1.19
STDEV	1,724.73	130.24	5,746.84	82.89	5,587.87	74.75	0.1329	2.31	0.2673	0.63
rust-111502	71	166	930	157	1,155	160	-3.61%	16.27	1.91%	1.24
rust-112061	2,188	458	17,728	442	14,658	418	-8.73%	6.70	-5.43%	0.83
rust-112213	3,617	736	55,834	635	17,786	549	-25.41%	4.92	-13.54%	0.32
rust-112526	2,900	382	10,932	338	10,407	355	-7.07%	3.59	5.03%	0.95
rust-44800	1,048	467	7,766	284	6,117	307	-34.26%	5.84	8.10%	0.79
rust-66851	4,047	728	31,256	713	39,530	660	-9.34%	9.77	-7.43%	1.26
rust-69039	518	114	5,397	101	6,363	99	-13.16%	12.28	-1.98%	1.18
rust-77002	331	263	1,582	247	1,772	246	-6.46%	5.35	-0.40%	1.12
rust-77320	10	39	153	39	147	9	-76.92%	14.70	-76.92%	0.96
rust-77323	4	13	26	13	86	13	0.00%	21.50	0.00%	3.31
rust-77910	13	34	112	21	451	20	-41.18%	34.69	-4.76%	4.03
rust-77919	28	74	280	62	656	69	-6.76%	23.43	11.29%	2.34
rust-78005	18	102	212	102	282	82	-19.61%	15.67	-19.61%	1.33
rust-78325	5	28	61	26	320	26	-7.14%	64.00	0.00%	5.25
rust-78651	9	17	55	9	99	9	-47.06%	11.00	0.00%	1.80
rust-78652	14	56	182	49	195	56	0.00%	13.93	14.29%	1.07
rust-78655	5	26	107	26	230	26	0.00%	46.00	0.00%	2.15
rust-78720	25	72	351	56	493	56	-22.22%	19.72	0.00%	1.40
rust-91725	284	174	1,700	78	1,315	108	-37.93%	4.63	38.46%	0.77
rust-99830	544	303	3,921	281	3,212	281	-7.26%	5.90	0.00%	0.82
Mean	783.95	212.60	6,929.25	183.95	5,263.70	177.45	-18.71%	12.37	-2.55%	1.33
STDEV	1,302.36	229.65	13,915.68	208.93	9,579.61	193.12	0.1994	15.44	0.2099	1.24

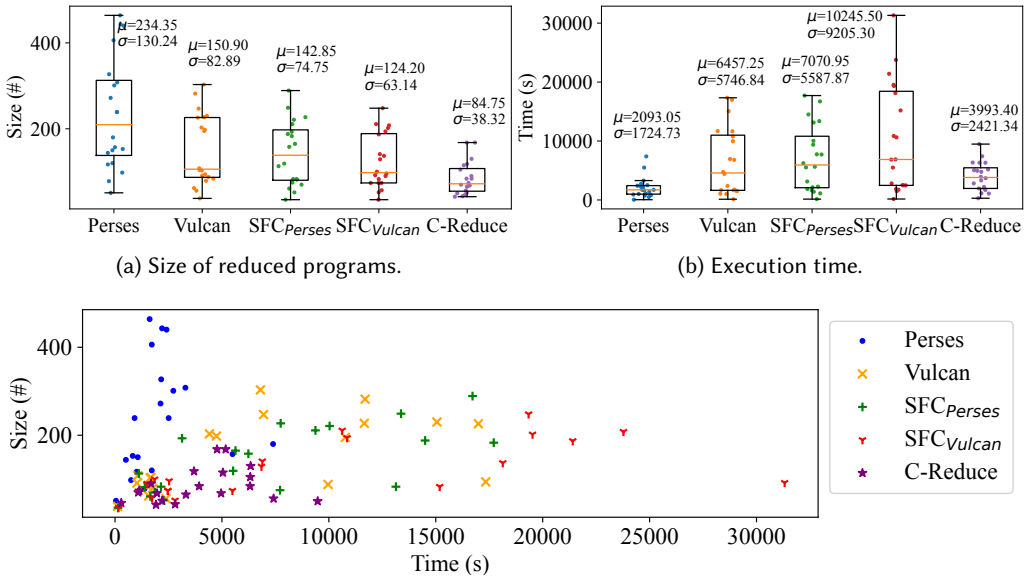
RQ2: Despite introducing some time overhead, SFC-based reduction methods can significantly enhance the canonicalization capabilities of Perses and Vulcan (with T-Rec integrated). Specifically, SFC_{Perses} and SFC_{Vulcan} elimiate 442 and 435 more duplicates than Perses and Vulcan, respectively.

Table 2. Evaluation results of C-Reduce, Vulcan, and SFC_{Vulcan} on C and Rust benchmarks. Column %Δ shows the average **percentage change in size** achieved compared to the baseline reducer.

Benchmarks	C-Reduce		Vulcan			SFC_{Vulcan}				
	Time (s)	Size (#)	Time (s)	Size (#)	%Δ in size *w.r.t.* C-Reduce	Time (s)	Size (#)	%Δ in size *w.r.t.* Vulcan	Time ratio *w.r.t.* Vulcan	%Δ in size *w.r.t.* C-Reduce
clang-22382	1,085	70	1,058	108	54.29%	1,852	100	-7.41%	1.75	42.86%
clang-22704	1,916	42	1,574	62	47.62%	1,731	57	-8.06%	1.10	35.71%
clang-23309	3,696	118	6,802	303	156.78%	10,624	211	-30.36%	1.56	78.81%
clang-23353	1,135	74	1,025	91	22.97%	1,496	90	-1.10%	1.46	21.62%
clang-25900	1,687	90	1,676	104	15.56%	2,528	96	-7.69%	1.51	6.67%
clang-26760	2,811	43	2,384	56	30.23%	2,807	52	-7.14%	1.18	20.93%
clang-27137	9,477	50	9,962	88	76.00%	15,185	83	-5.68%	1.52	66.00%
clang-27747	1,963	68	1,759	79	16.18%	2,471	74	-6.33%	1.40	8.82%
clang-31259	4,772	168	11,695	282	67.86%	19,341	248	-12.06%	1.65	47.62%
gcc-59903	6,326	105	4,743	198	88.57%	6,849	129	-34.85%	1.44	22.86%
gcc-60116	5,174	168	6,945	247	47.02%	10,853	194	-21.46%	1.56	15.48%
gcc-61383	3,934	84	10,766	195	132.14%	18,134	137	-29.74%	1.68	63.10%
gcc-61917	3,316	65	1,650	103	58.46%	2,471	74	-28.16%	1.50	13.85%
gcc-64990	4,961	68	4,416	203	198.53%	6,888	141	-30.54%	1.56	107.35%
gcc-65383	2,206	51	1,531	84	64.71%	5,492	74	-11.90%	3.59	45.10%
gcc-66186	5,045	115	16,992	226	96.52%	23,779	208	-7.96%	1.40	80.87%
gcc-66375	7,414	56	11,656	227	305.36%	19,525	202	-11.01%	1.68	260.71%
gcc-70127	6,316	84	15,053	230	173.81%	21,406	187	-18.70%	1.42	122.62%
gcc-70586	6,342	130	17,335	94	-27.69%	31,316	92	-2.13%	1.81	-29.23%
gcc-71626	292	46	123	38	-17.39%	162	35	-7.89%	1.32	-23.91%
Mean	3,993.40	84.75	6,457.25	150.90	80.38%	10,245.50	124.20	-14.51%	1.56	50.39%
STDEV	2,421.34	38.32	5,746.84	82.89	79.90%	9,205.30	63.14	0.1073	0.49	63.10%
rust-111502	822	161	930	157	-2.48%	2,370	156	-0.64%	2.55	-3.11%
rust-112061	4,892	447	17,728	442	-1.12%	25,980	413	-6.56%	1.47	-7.61%
rust-112213	4,413	726	55,834	635	-12.53%	70,048	536	-15.59%	1.25	-26.17%
rust-112526	5,921	537	10,932	338	-37.06%	18,530	334	-1.18%	1.70	-37.80%
rust-44800	6,436	471	7,766	284	-39.70%	10,731	284	0.00%	1.38	-39.70%
rust-66851	5,673	649	31,256	713	9.86%	68,837	658	-7.71%	2.20	1.39%
rust-69039	649	109	5,397	101	-7.34%	10,379	97	-3.96%	1.92	-11.01%
rust-77002	2,097	264	1,582	247	-6.44%	3,428	246	-0.40%	2.17	-6.82%
rust-77320	147	39	153	39	0.00%	367	9	-76.92%	2.40	-76.92%
rust-77323	25	13	26	13	0.00%	128	13	0.00%	4.92	0.00%
rust-77910	88	23	112	21	-8.70%	373	20	-4.76%	3.33	-13.04%
rust-77919	422	70	280	62	-11.43%	699	59	-4.84%	2.50	-15.71%
rust-78005	308	75	212	102	36.00%	603	82	-19.61%	2.84	9.33%
rust-78325	122	34	61	26	-23.53%	442	26	0.00%	7.25	-23.53%
rust-78651	76	12	55	9	-25.00%	150	9	0.00%	2.73	-25.00%
rust-78652	176	49	182	49	0.00%	384	49	0.00%	2.11	0.00%
rust-78655	93	26	107	26	0.00%	450	26	0.00%	4.21	0.00%
rust-78720	476	51	351	56	9.80%	861	53	-5.36%	2.45	3.92%
rust-91725	2,387	101	1,700	78	-22.77%	2,341	77	-1.28%	1.38	-23.76%
rust-99830	4,278	164	3,921	281	71.34%	6,350	269	-4.27%	1.62	64.02%
Mean	1,975.05	201.05	6,929.25	183.95	-3.55%	11,172.55	170.80	-7.65%	2.35	-11.58%
STDEV	2,341.12	231.04	13,915.68	208.93	24.54%	21,095.72	190.52	0.1715	1.44	26.68%

4.3 RQ3: Ablation Studies

To investigate whether each proposed SFC-based reduction methods are effective, we conducted ablation studies. Specifically, we implemented six variants of SFC_{Perses} and SFC_{Vulcan} as follows. $SFC_{Perses}^{\text{--ie}}$, $SFC_{Perses}^{\text{--ssr}}$, and $SFC_{Perses}^{\text{--sc}}$ are variants of SFC_{Perses} that do not have the auxiliary reducer corresponding to Identifier Elimination,Smaller Structure Replacement, and Structure Canonicalization integrated, respectively. $SFC_{Vulcan}^{\text{--ie}}$, $SFC_{Vulcan}^{\text{--ssr}}$, and $SFC_{Vulcan}^{\text{--sc}}$ are variants of SFC_{Vulcan} without these three auxiliary reducer, respectively.

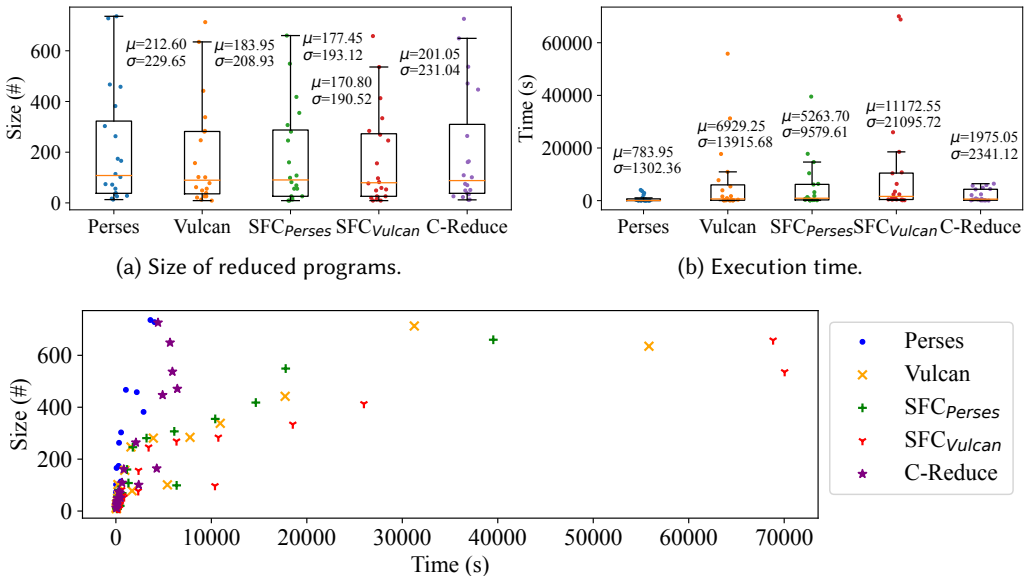


(a) Size of reduced programs.

(b) Execution time.

(c) Execution Time vs. Program Size.

Fig. 5. Evaluation results of Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} on C benchmarks.



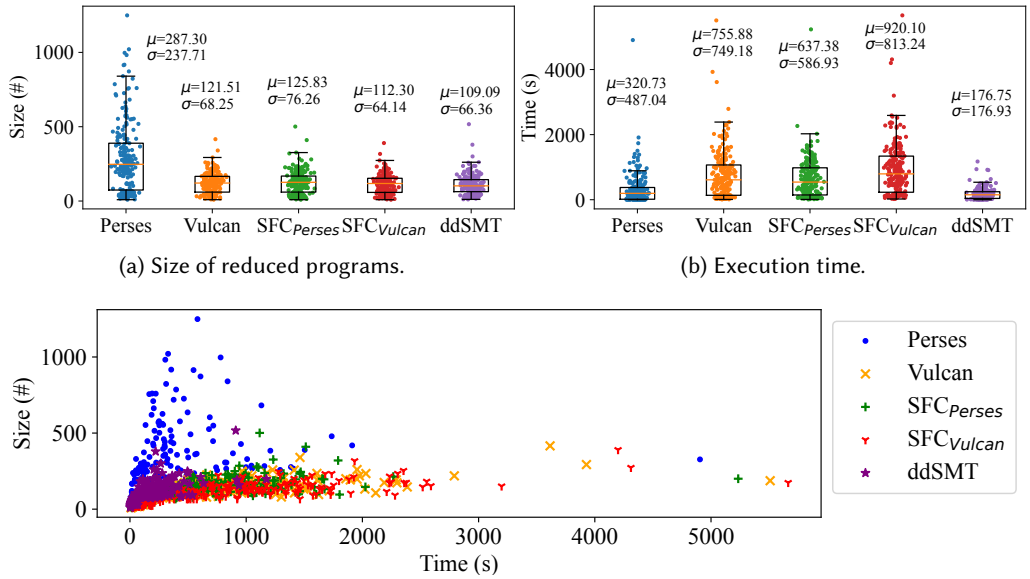
(a) Size of reduced programs.

(b) Execution time.

(c) Execution Time vs. Program Size.

Fig. 6. Evaluation results of Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} on Rust benchmarks.

Impacts of Removing Smaller Structure Replacement. Smaller Structure Replacement is proposed to mainly enhance the minimization capabilities of program reducer. Thus, to evaluate its contribution, we compare $SFC_{Perses}^{\text{--ssr}}$ and $SFC_{Vulcan}^{\text{--ssr}}$ with SFC_{Perses} and SFC_{Vulcan} on



(a) Size of reduced programs.

(b) Execution time.

(c) Execution Time vs. Program Size.

Fig. 7. Evaluation results of Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} on SMT-LIBv2 benchmarks.

Table 3. Evaluation results of Perses, SFC_{Perses}, $SFC_{Perses^{--sc}}$, Vulcan, SFC_{Vulcan}, and $SFC_{Vulcan^{--sc}}$ on `Benchmark-Cano`. The column **Eliminated** shows the total number of eliminated duplicates. The column **Size** and **Time** shows the average size of the outputs and the average execution time.

Reducer	Crash			Miscompilation		
	Eliminated (#)	Size (#)	Time (s)	Eliminated (#)	Size (#)	Time (s)
Perses	1,297/2,501	55.73	146.58	0/1,295	228.14	769.03
SFC_{Perses}	1,736/2,501	39.59	248.31	3/1,295	168.77	2367.54
$SFC_{Perses^{--sc}}$	1,382/2,501	41.32	217.75	1/1,295	169.99	2001.45
Vulcan	1,327/2,501	40.81	293.68	1/1,295	188.56	2896.75
SFC_{Vulcan}	1,752/2,501	36.16	312.42	11/1,295	160.88	3914.47
$SFC_{Vulcan^{--sc}}$	1,397/2,501	37.83	284.13	8/1,295	156.28	4245.39

`Benchmark-Reduce`, respectively. From Table 4, it can be observed that for both SFC_{Perses} and SFC_{Vulcan}, removing Smaller Structure Replacement significantly deteriorates the minimization performance. Specifically, removing Smaller Structure Replacement causes SFC_{Perses} to produce results that are 16.70%, 29.90%, and 16.31% larger on C, Rust, and SMT-LIBv2 benchmarks, respectively. Similarly, removing Smaller Structure Replacement from SFC_{Vulcan} results in outputs that are 13.56%%, 18.89%, and 6.03% larger, respectively.

Impacts of Removing Identifier Elimination. Similar to Smaller Structure Replacement, Identifier Elimination is also proposed to primary boost the minimization performance. Therefore, we evaluate its contribution in the same way we assess Smaller Structure Replacement. As shown in Table 4, removing Identifier Elimination also has a negative impact on the minimization capability of SFC_{Perses}, especially on the SMT-LIBv2 benchmarks. Specifically, the results of SFC_{Perses}

Table 4. Results of $SFC_{Perses^{--ie}}$, $SFC_{Perses^{--ssr}}$, $SFC_{Vulcan^{--ie}}$, and $SFC_{Vulcan^{--ssr}}$ on `Benchmark-Reduce`. Column **Worse** is the number of cases where the removal causes a deterioration in minimization capability. Column %Δ is the **average percentage increase in size** caused by the removal.

Reducer	C			Rust			SMT-LIBv2		
	Size	Worse	%Δ in Size	Size	Worse	%Δ in Size	Size	Worse	%Δ in Size
$SFC_{Perses^{--ie}}$	192.70	20/20	39.03%	194.05	10/20	7.21%	266.23	169/205	93.91%
$SFC_{Perses^{--ssr}}$	165.90	20/20	16.70%	190.65	17/20	29.90%	145.17	167/205	16.31%
$SFC_{Vulcan^{--ie}}$	132.35	12/20	5.22%	171.35	2/20	0.25%	113.83	29/205	1.11%
$SFC_{Vulcan^{--ssr}}$	141.10	20/20	13.56%	176.05	12/20	18.89%	117.38	137/205	6.03%

without Identifier Elimination are 39.03%, 7.21%, and 93.91% larger on C, Rust, and SMT-LIBv2 benchmarks, respectively. The negative impact of removing Identifier Elimination from SFC_{Vulcan} is not as significant as that of removing it from SFC_{Perses}, the corresponding percentage increases in size of the results are 5.22%, 0.25%, and 1.11%, respectively. The reason behind this may be that Vulcan also has an auxiliary reducer for eliminating identifiers, which is somewhat overlapped with Identifier Elimination. Nevertheless, the increases in size still demonstrates that Identifier Elimination proposed in this paper is more effective than the auxiliary reducer in Vulcan.

Impacts of Removing Structure Canonicalization. Since Structure Canonicalization is proposed primarily to enhance the canonicalization capability of program reducers, to evaluate the contribution of Structure Canonicalization to the improved canonicalization performance, we compare $SFC_{Perses^{--sc}}$ and $SFC_{Vulcan^{--sc}}$ to SFC_{Perses} and SFC_{Vulcan} on `Benchmark-Cano`, respectively. As shown in Table 3, removing Structure Canonicalization from SFC_{Perses} leads to 354 and 2 fewer eliminated duplicates. Removing Structure Canonicalization from SFC_{Vulcan} also leads to 355 and 3 fewer eliminated duplicates, respectively. Such results indicate that Structure Canonicalization significantly contributes to the superior canonicalization performance of both SFC_{Perses} and SFC_{Vulcan}.

> **RQ3**: All the proposed SFC-based reduction methods significantly contributes to the improved reduction performance of SFC_{Perses} and SFC_{Vulcan}. Removing Smaller Structure Replacement or Identifier Elimination can results in large outputs and the ratio is upto 93.91%. Without Structure Canonicalization, more than 300 duplicated crash bugs cannot be eliminated.

5 Related Work

This section discusses the studies related to this paper. *Delta Debugging* is the first systematic work in the direction of bug-triggering test input minimization [42]. This work proposed an algorithm named `ddmin` which is designed to reduce a list of elements. This algorithm still serves as a fundamental reduction algorithm in many advanced reduction techniques proposed by subsequent research studies [18, 19, 25, 26, 30, 39, 44, 46]. ProbDD later improves `ddmin` with a probabilistic model [35]. Zhang *et al.* conducted an in-depth theoretical and empirical analysis of ProbDD, and then proposed Counter-Based Delta Debugging (CDD), a simplified version of ProbDD that reduce the complexity from both theory and implementation perspectives [44]. Zhou *et al.* introduced Weighted Delta Debugging (WDD), a novel concept to leverage the size of elements in partitioning [46].

The algorithm `ddmin` and its variants cannot well handle highly structured inputs on themselves. Targeting to tackle this limitation, *Hierarchical Delta Debugging* (HDD) was proposed by Misherghi and Su [25]. Specifically, HDD performs `ddmin` on each level of the tree representation (*e.g.*,

the parse tree) of the input from top to bottom. After HDD was proposed, many researchers improved HDD from different aspects with different mechanisms [12, 13, 18, 34]. Hodován *et al.* proposed Picieny, which uses extended context-free grammars to modernized the original HDD implementation [12]. In the next year, the same authors proposed another technique called Coarse Hierarchical Delta Debugging (CHDD) to further improve the efficiency of HDD [13]. Kiss *et al.* proposed a recursive version of HDD named HDDr also for saving reduction time [18]. Vince *et al.* improve the performance of HDD by integrating the hoisting operation [34].

Although HDD and its variants can recognize the structure of the input to be reduced, the transformations (*i.e.*, deletions) that they perform do not preserve the syntactic validity, which eventually leads to limited overall reduction performance. To overcome this obstacle, Sun *et al.* propose Perses, a *syntax-guided program reducer* that always generates programs that are conform to the provided formal grammar during the reduction process. Gharachorlu *et al.* further improves the efficientcy of Perses by optimizing the order of reducing each portion of the program [7, 8], and Tian *et al.* proposed a better implementation that greatly reduces the engineering work required for applying Perses to different programming languages [31]. Tian *et al.* introduced a novel caching scheme to speed up program reduction [33].

To *further trade off efficiency for smaller reduction results*, Xu *et al.* proposed Vulcan, a reducer that can switch to more exhaustive reducers whenever the main reducer (*i.e.*, Perses) cannot make any progress [39]. In the study of Vulcan, three auxiliary reducers are proposed, which are Tree-Based Local Exhaustive Enumeration, Identifier Replacement, and Sub-Tree Replacement, respectively. Among these auxiliary reducers, the last two are similar to Identifier Elimination and Smaller Structure Replacement proposed in this paper. However, compared to Identifier Replacement, Identifier Elimination is more effective as it can eliminate uses of identifiers not only by replacing but also by utilizing SFC. Meanwhile, compared to Smaller Structure Replacement, Sub-Tree Replacement in Vulcan lacks a structure-reusing mechanism, which significantly limits its effectiveness. The extensive evaluation also demonstrates that compared to Vulcan, the methods proposed in this paper are either more effective or complementary. T-Rec is another syntax-guided program reducer proposed recently [38]. It improves the reduction performance (especially the canonicalization performance) by introducing a fine-grained reduction process, which leverages the lexical syntax to canonicalize tokens. As demonstrated in § 4.1, SFC-based reduction methods can significantly enhance the canonicalization capabilities of Perses and Vulcan with T-Rec integrated. Such a result indicates that T-Rec and our approach are complementary to some extent.

There are also many *language-specific program reducers* that are specifically optimized for reducing programs in certain programming languages. These reducers are designed with language-specific knowledge and often require heavy engineering work to implement. C-Reduce is a reducer that specifically optimized for reducing C programs [27]. ddSMT and ddSMT2.0 are designed for reducing SMT-LIBv2 programs [19, 26]. J-Reduce focuses on Java bytecode [16, 17], and JS Delta is for reducing JavaScript programs [15]. Zhang *et al.* proposed LPR, the first LLMs-aided technique leveraging LLMs to perform language-specific program reduction for multiple languages [43]. Compared to these language-specific ones, SFC-based reduction is much more general and can be applied to a wide range of programming languages.

There are also program reducers designed for software debloating. For example, Chisel is a reducer for debloating C/C++ programs [11]. It leverages a reinforcement learning-based approach to improve the efficiency and scalability. Additionally, Bruce *et al.* proposed JShrink, an end-to-end framework for debloating Java bytecode [1].

Instead of directly reducing the test case, some reducers reduce test cases in a different manner. Hypothesis reducer reduces test cases by minimizing the choice sequence with test case generators [24]. PPR, proposed by Zhang *et al.*, reduces pairs of programs, *i.e.*, a seed and a bug-triggering

mutant derived from the seed, in mutation-based fuzzing [45]. It reduces both the difference and the common parts of the two programs to facilitate debugging. While the reducer in spirv-fuzz also reduces difference between seed and mutant, it focuses on reducing the transformation sequences that mutate the seed to the mutant [5]. These works are all complementary to our work.

6 Conclusion

In this paper, we boost the performance of language-agnostic program reduction by proposing a novel syntax-guided transformation named Structure Form Conversion (SFC). SFC complements the previously proposed two transformations, Compatible Substructure Hoisting and Quantified Node Reduction. Building on SFC, we further propose three reduction methods: Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization, to effectively and efficiently leverage SFC for program reduction. Through extensive evaluation with two benchmark suites, we demonstrate the proposed three reduction methods can significantly improve both the minimization and canonicalization performance. Specifically, $\text{SFC}_{\text{Perses}}$ produces 36.82%, 18.71%, and 41.05% smaller results than Perses on average on the C, Rust, and SMT-LIBv2 benchmarks, respectively. Similarly, the average results of $\text{SFC}_{\text{Vulcan}}$ are 14.51%, 7.65%, and 7.66% smaller than Vulcan. Furthermore, in `Benchmark-Cano` SFC-based reduction methods help Perses and Vulcan further eliminate 442 and 435 more duplicates (programs that trigger the same bug), respectively.

7 Data Availability

For reproducibility and replicability, we have released the replication package at https://github.com/sfc-reducer/sfc-reducer.

Acknowledgments

We thank all the anonymous reviewers in OOPSLA'25 for their insightful feedback and comments, which significantly improved this paper. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under WHJIL, and CFI-JELF Project #40736.

References

[1] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: in-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 135–146.

[2] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 197–208.

[3] Noam Chomsky. 1959. On certain formal properties of grammars. *Information and Control* 2, 2 (1959), 137–167. doi:10.1016/S0019-9958(59)90362-6

[4] CPython. 2022. *Bug Report.* Retrieved 2022-09-20 from https://github.com/python/cpython/issues/new?assignees=&labels=type-bug&template=bug.md

[5] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1017–1032.

[6] GCC-Wiki. 2020. *A guide to Testcase reduction.* Retrieved 2022-09-20 from https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[7] Golnaz Gharachorlu and Nick Sumner. 2019. PARDIS: Priority Aware Test Case Reduction. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 409–426.

[8] Golnaz Gharachorlu and Nick Sumner. 2023. Type Batched Program Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 398–410. doi:10.1145/3597926.3598065

[9] Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52. doi:10.1145/321250.321254

[10] Alex Groce, Josie Holmes, and Kevin Kellar. 2017. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3092703.3092704

[11] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394. doi:10.1145/3243734.3243838

[12] Renáta Hodován and Ákos Kiss. 2016. Modernizing Hierarchical Delta Debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation* (Seattle, WA, USA) *(A-TEST 2016)*. Association for Computing Machinery, New York, NY, USA, 31–37. doi:10.1145/2994291.2994296

[13] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 194–203. doi:10.1109/ICSME.2017.26

[14] JerryScript. 2022. *Bug Report*. Retrieved 2022-09-20 from https://github.com/jerryscript-project/jerryscript/blob/master/.github/ISSUE_TEMPLATE/bug_report.md

[15] JS Delta. 2017. *JS Delta*. Retrieved 2022-10-28 from https://github.com/wala/jsdelta

[16] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 556–566. doi:10.1145/3338906.3338956

[17] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1003–1016. doi:10.1145/3453483.3454091

[18] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: A Recursive Variant of the Hierarchical Delta Debugging Algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 16–22. doi:10.1145/3278186.3278189

[19] Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 231–242. doi:10.1007/978-3-030-81688-9_11

[20] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. (2014), 216–226. doi:10.1145/2594291.2594334

[21] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.

[22] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[23] LLVM. 2022. *How to Submit an LLVM bug report*. Retrieved 2022-09-20 from https://llvm.org/docs/HowToSubmitABug.html

[24] David R MacIver and Alastair F Donaldson. 2020. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[25] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151. doi:10.1145/1134285.1134307

[26] Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*. 8–9.

[27] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. doi:10.1145/2254064.2254104

[28] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 335–346.

[29] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. doi:10.1145/2983990.2984038

[30] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. doi:10.1145/3180155.3180236

[31] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2137–2141. doi:10.1145/3611643.3613101

[32] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 4873–4882. doi:10.24963/IJCAI.2023/542

[33] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 17 (nov 2023), 30 pages. doi:10.1145/3617172

[34] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. 2021. Extending hierarchical delta debugging with hoisting. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 60–69.

[35] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 881–892. doi:10.1145/3468264.3468625

[36] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 146–158. doi:10.1145/3575693.3575740

[37] Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. 2025. Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 25-28, 2025*, Mike Papadakis, Myra B. Cohen, and Paolo Tonella (Eds.). ACM, 21–25. doi:10.1145/3713081.3731731

[38] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. 2024. T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax. *ACM Trans. Softw. Eng. Methodol.* (aug 2024). doi:10.1145/3690631 Just Accepted.

[39] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (apr 2023), 29 pages. doi:10.1145/3586049

[40] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. 2023. Silent Compiler Bug De-duplication via Three-Dimensional Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 677–689. doi:10.1145/3597926.3598087

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[42] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498

[43] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria)(ISSTA 2024). Association for Computing Machinery, New York, NY, USA. https://doi. org/10.1145/3650212.3652126.*

[44] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Xinru Cheng, and Chengnian Sun. 2025. Toward a Better Understanding of Probabilistic Delta Debugging. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2024–2035. doi:10.1109/ICSE55347.2025.00117

[45] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of*

Software Engineering. 338–349.

[46] Xintong Zhou, Zhenyang Xu, Mengxiao Zhang, Yongqiang Tian, and Chengnian Sun. 2025. WDD: Weighted Delta Debugging. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1592–1603. doi:10.1109/ICSE55347.2025.00071

Received 2025-03-25; accepted 2025-08-12