Perses: Syntax-Guided Program Reduction

Chengnian Sun
University of California
Davis, CA, USA
cnsun@ucdavis.edu

Yuanbo Li
University of California
Davis, CA, USA
yboli@ucdavis.edu

Qirun Zhang
University of California
Davis, CA, USA
qrzhang@ucdavis.edu

Tianxiao Gu
University of California
Davis, CA, USA
txgu@ucdavis.edu

Zhendong Su
University of California
Davis, CA, USA
su@ucdavis.edu

ABSTRACT

Given a program P that exhibits a certain property ψ (*e.g.*, a C program that crashes GCC when it is being compiled), the goal of *program reduction* is to minimize P to a smaller variant P' that still exhibits the same property, *i.e.*, $\psi(P')$. Program reduction is important and widely demanded for testing and debugging. For example, all compiler/interpreter development projects need effective program reduction to minimize failure-inducing test programs to ease debugging. However, state-of-the-art program reduction techniques — notably Delta Debugging (DD), Hierarchical Delta Debugging (HDD), and C-Reduce — do not perform well in terms of speed (reduction time) and quality (size of reduced programs), or are highly customized for certain languages and thus lack generality.

This paper presents `Perses`, a novel framework for *effective*, *efficient*, and *general* program reduction. The *key insight* is to exploit, in a general manner, the formal syntax of the programs under reduction and ensure that each reduction step considers only *smaller*, *syntactically valid* variants to avoid futile efforts on syntactically invalid variants. Our framework supports not only deletion (as for DD and HDD), but also general, effective program transformations.

We have designed and implemented `Perses`, and evaluated it for two language settings: C and Java. Our evaluation results on 20 C programs triggering bugs in GCC and Clang demonstrate `Perses`'s strong practicality compared to the state-of-the-art: (1) *smaller size* — `Perses`'s results are respectively *2%* and *45%* in size of those from DD and HDD; and (2) *shorter reduction time* — `Perses` takes *23%* and *47%* time taken by DD and HDD respectively. Even when compared to the highly customized and optimized C-Reduce for C/C++, `Perses` takes only *38-60%* reduction time.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

program reduction, delta debugging, debugging

ACM Reference Format:
Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180236

1 INTRODUCTION

Program reduction is important and widely used. Given a program P that exhibits a property, the objective of program reduction is to generate a smaller program P' from P that still exhibits the same property. The reduction process continuously generates smaller program variants, and checks them against the property. A minimal variant that perserves the property is returned at the end.

A common usage scenario for program reduction is reducing test programs that trigger compiler bugs. It is well-known that debugging typical application software is painstaking and daunting. Debugging compilers can be even more difficult as compilers are among the most complex software systems (*e.g.*, GCC's codebase is close to 20 million lines). It is worse when the bug-triggering program contains a large amount of irrelevant code to the bug as compiler developers need to manually distill useful information from the test program. Most of the time, the size difference between the original and the distilled version can be considerable. For example, a recent study on the characteristics of bugs in GCC and LLVM [21] has shown that, on average, minimized test programs contain only 30 lines of code to trigger compiler bugs. In contrast, the original bug-triggering test programs may have hundreds or thousands of lines of code [9, 11, 22, 25]. Therefore, it is important to automatically compute the distilled version. Indeed, compiler developers strongly encourage submitting small, reproducible test programs — both GCC and LLVM advocate test reduction in their bug reporting processes [4, 13].

Existing Reduction Techniques. The state-of-the-art program reduction techniques are Delta Debugging [24], Hierarchical Delta Debugging [16], and C-Reduce [17].

Delta Debugging (DD): Zeller and Hildebrandt [24] proposed DD to minimize failure-inducing test inputs. It is general, and works on not only programs, but also arbitrary inputs as DD can operate at the individual byte level. Initially, the test input is split into a list of elements at a certain granularity (*e.g.*, byte, lexeme or line). New

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05...$15.00
https://doi.org/10.1145/3180155.3180236

inputs are then systematically generated by deleting elements from the list, and checked whether they still trigger the same failure. The algorithm terminates with a minimal failure-inducing input.

DD is general, but not suited for reducing programs because, by design, it does not leverage nor respect syntatic structural constraints available in programs and enforced by a programming language's formal syntax. Thus, DD inevitably considers a large number of candidate inputs that are syntactically invalid, leading to poor reduction performance.

Hierarchical Delta Debugging (HDD): Misherghi and Su [16] proposed HDD to improve the effectiveness of DD for tree-structured inputs. Different from DD, HDD converts the input into a tree *w.r.t.* the input's format. In the particular case of program reduction, HDD converts a program into its parse tree or abstract syntax tree (AST), performs breadth-first search on the tree, and invokes the DD algorithm on each level to prune tree nodes from that level.

Although HDD's tree-based reduction considers fewer syntactically invalid programs than DD and thus outperforms it, it provides no guarantee and still generates many syntactically invalid inputs (*e.g.*, removing the variable name in a declaration). The reason is that HDD only uses the formal syntax of a language (*i.e.*, its grammar) to convert a test input into its tree representation, but does not exploit the grammar further to guide reduction. As HDD's core reduction is based on DD, it generates smaller programs only via deletion, thus may be incapable of generating smaller programs that require other transformations (see Section 2 for illustration).

C-Reduce: Regehr *et al.* proposed C-Reduce [17], a highly customized reducer for C/C++, that embodies a set of C/C++ program transformations implemented via the Clang LibTooling [14] library.

Although C-Reduce is powerful at minimizing C/C++ programs, the reduction process may take significant time. To mitigate its performance issue, Le *et al.* combined DD and C-Reduce to reduce C test programs in the EMI compiler testing work [9]. More importantly, C-Reduce is not general — to support another programming language, one needs to re-design and re-implement program transformations targetting the language, which is obviously nontrivial.

In short, DD is fast, but often cannot produce good quality (*i.e.*, small) reductions; C-Reduce offers high-quality reductions for C/C++ programs, but lacks generality and takes much reduction time; and HDD is in between — it produces better reductions than DD, and runs faster than C-Reduce.

Syntax-Guided Program Reduction. This paper addresses the aforementioned shortcomings of existing techniques by proposing a novel framework, `Perses`, to enable general, effective, and efficient program reduction. Our *conceptual insight* is to utilize the formal syntax (*i.e.*, grammar) of a programming language to guide reduction. Our *technical insight* is to leverage the grammar to (1) generate only *syntactically valid* inputs, and (2) support generic, effective program transformations.

In more detail, program reduction is about searching for suitable programs in a search space $\mathbb{P}$,[1] which can be partitioned into two disjoint sets: the set of *syntactically valid* programs $\mathbb{P}_{valid}$, and the set of *syntactically invalid* ones $\mathbb{P}_{invalid}$, *i.e.*,

$$\mathbb{P} = \mathbb{P}_{invalid} \uplus \mathbb{P}_{valid}$$

where $\uplus$ denotes the disjoint union operator. DD and HDD[2] have non-empty $\mathbb{P}_{invalid}$, because as aforementioned DD does not consider program syntax at the preprocessing step (*i.e.*, converting a test program into a list) or during reduction, and HDD does not either during reduction. Both algorithms generate a considerable number of syntactically invalid programs, only being a waste of reduction time. On the other hand, both reduction algorithms only delete elements from test programs, which limits the search space of syntactically valid programs, *i.e.*, $\mathbb{P}_{valid}$.

Our use of grammars is fundamentally different from that of HDD. HDD only uses grammars to parse programs into tree structures, whereas `Perses` further analyzes and leverages grammars during reduction. This brings us two advantages:

$\mathbb{P}_{invalid} = \varnothing$: From the definition of the grammar, during reduction `Perses` can determine whether a tree node is deletable. If no, we can avoid generating variants by deleting that node. This makes it very easy for `Perses` to ensure $\mathbb{P}_{invalid} = \varnothing$, which consequently reduces the number of unnecessary property tests.

Enlarging $\mathbb{P}_{valid}$: `Perses` leverages the grammar to support more advanced program transformations, which thus increases $\mathbb{P}_{valid}$. For example, according to the C grammar, a conditional statement `if (...) {print("");}` can be simplified to `if (...) print("");`, as the true branch of an if statement is a statement, and the compound statement `{print("");}` and the function call `print("");` are both statements. Though increased $\mathbb{P}_{valid}$ may increase the number of property tests, it enables `Perses` to generate more valid, diverse, possibly smaller variants than DD and HDD.

It is easy for `Perses` to support a new language by just providing its grammar in Backus-Naur form (BNF). However, it is very likely that the original grammar in arbitrary form may hinder the effectiveness of `Perses`. Thus we define a normal form of grammars to facilitate reduction, referred to as Perses Normal Form (PNF). We also design an algorithm to *automatically* convert any grammar into its *equivalent* PNF.

Our evaluation results of `Perses` on 20 large C programs that trigger bugs in GCC and Clang demonstrates that `Perses`'s strong practicality compared to the state-of-the-art: (1) *smaller size* — `Perses`'s results are respectively *2%* and *45%* in size of those from DD and HDD; and (2) *shorter reduction time* — `Perses` takes *23%* and *47%* time taken by DD and HDD respectively. Even when compared to the highly customized and optimized C-Reduce for C/C++, `Perses` takes only *38-60%* reduction time. To demonstrate the generality of `Perses`, we also evaluate `Perses` on 6 small Java programs triggering bugs in Javac and Eclipse Compiler for Java, and the results are consistent with those on C programs. `Perses` constantly outperforms HDD in terms of both time and reduction quality.

Contributions. This paper makes the following contributions.

- We propose `Perses`, a framework for effective and efficient program reduction. It is the first *general* program reducer that leverages formal syntax to guide program reduction. It guarantees no generation of syntactically invalid programs, and supports more program transformations to produce smaller reduction results.

[1] $\mathbb{P}$ is the universe of candidate programs that a program reducer can derive from P, where P is the initial program to reduce.

[2] We exclude C-Reduce from comparison and discussion in the remainder of the paper except Section 5, as C-Reduce is language-specific and this paper focuses on language-independent program reducers.

- We define Perses Normal Form of grammars to facilitate program reduction, and also propose an algorithm to *automatically* convert any context free grammar into an equivalent PNF.
- Our evaluation results on real-world benchmarks demonstrate significant improvement of `Perses` over the state of the art.
- `Perses` provides a new perspective for program reduction. With the knowledge of formal syntax, we can design more program transformations to produce better reduction results.

2 A MOTIVATING EXAMPLE

We illustrate `Perses` with a contrived C program. The full algorithm is detailed in Section 4. Figure 1a shows a program that prints three lines ('`1`', '`Hello world!`', and '`End`'). Assume that we are interested in the behavior of printing '`Hello world!`'. Then `Perses` outputs a minimized program that only prints '`Hello world!`'.

Different from HDD which performs reduction level by level, `Perses` does node by node. It maintains a priority queue Q to store the tree nodes for reduction. Each time, it retrieves the node with the most tokens from Q, and reduces that node with various *syntax-preserving* program transformations. After the node is reduced, the remaining children of the node are added to Q for future reduction.

Initially, `Perses` parses this program into the parse tree shown in Figure 1e. Then, it performs the following reduction steps:

Step 1 (`1.func_def`): `Perses` reduces the root of the parse tree, *i.e.*, '`1.func_def`'. However, removing it does not possess the property. So all its children are added to Q.

Step 2 (`2.compound_stmt`): The function body '`2.compound_stmt`' is dequeued for reduction, because it has the most tokens in Q. For this compound statement, we cannot delete it, as the function body is compulsory for a function definition. However, we can replace it with one of its descendants which is also a compound statement. In this case, we can replace it with '`6.compount_stmt`', the true branch of '`if(a)`' on line 3. After replacement, the new variant is listed below. Although it is syntactically correct, it does not compile as the identifier '`a`' is not defined. So this reduction step fails, and we add the three immediate children to Q.

```
int main() { printf("%d\n", a);
  printf("Hello␣"); printf("world!\n"); printf("End\n"); }
```

Step 3 (`3.stmt_star`): The suffix '_star' indicates that this node is a *Kleene-Star* node, and represents that it can have zero or more statements as immediate children. In other words, any child of this node can be deleted without violating the C program syntax. For this node, `Perses` uses DD to delete irrelevant children. Unfortunately, all three children cannot be deleted, and then are added to Q.

Step 4 (`4.if_stmt`): The largest node in Q is '`4.if_stmt`'. Its parent is '`3.stmt_star`' that expects zero or more statements. So we can find a statement to replace '`if_stmt`', and the program syntax will still be valid. We search all its children, and find its true branch to be a statement, *i.e.*, node '`6.compoudt_stmt`'. So we obtain the first successful reduced variant shown in Figure 1b.

Step 5 (`5.compound_stmt`): To reduce this node, we attempt to use one of its children to replace it. As it replaces '`4.if_stmt`', its parent becomes the node '`3.stmt_star`' which expects zero or more statements. Then we can use '`6.stmt_star`' (which is also a Kleene-Star node that expects zero or more statements) to replace this compound statement. In detail, '`6.stmt_star`' becomes a child of '`3.stmt_star`', and this transformation does not invalidate the grammar, because the tokens owned by '`3.stmt_star`' still constitute a list of statements. This transformation preserves the property, and the result is shown in Figure 1c.

Step 6 (`6.stmt_star`): As it is a Kleene star node, we use DD to reduce it. And nodes '`8.printf@4`' and '`11.printf@7`' are removed.

Following Steps: After '`8.printf@4`' (*i.e.*, the print statement on line 4) is removed, the dependency on variable `a` is also removed. When `Perses` picks '`12.int a = 1;`' from Q, this statement can be safely removed without introducing any compilation error.

Figure 1d shows the final result, and Figure 1f shows the final pruned parse tree. Only the two print statements and the return statement are kept in the result. The if statement is removed by replacing it with its child. This transformation is not possible in either HDD or DD except with case-by-case specialized solutions. In contrast, our approach is systematic, and involve no ad-hoc designs. All transformations in `Perses` are designed based on the semantics of language grammars, *e.g.*, deleting some children of Kleene Star nodes, replacing a statement node with another statement node.

3 PRELIMINARIES

This section formalizes program reduction, introduces DD, and defines `Perses` Normal Form.

3.1 Program Reduction

Let $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, P be a program that exhibits a property, $\mathbb{P}$ be the search space of programs defined by concrete program reduction algorithms over P. [3] We define a property test function $\psi : \mathbb{P} \to \mathbb{B}$, such that $\psi(P) = \mathsf{true}$ and for any program $p \in \mathbb{P}$, $\psi(p) = \mathsf{true}$ if p exhibits the property, otherwise $\psi(p) = \mathsf{false}$. The size of p is denoted as $|p|$, which is the number of tokens in p.

Given a program P and its property test ψ (*s.t.*, $\psi(P) = \mathsf{true}$), the goal of program reduction is to search for a minimized program $p \in \mathbb{P}$, such that $\psi(p) \wedge |p| < |P|$. *Ideally*, the goal of reducing P (denoted as $\mathsf{Reduce}(P)$) is defined as

$$\underset{p \in \mathbb{P} \wedge \psi(p)}{\arg\min} \ |p| \equiv \{p | p \in \mathbb{P} \wedge \psi(p) \wedge \forall x \in \mathbb{P}. |p| \leq |x|\}$$

1-Minimality and 1-Tree-Minimality. However, obtaining the global minimality is NP-complete [16, 24]. Therefore, in practice, the reduction problem is relaxed to compute the minimum result within a program reducer's capacity. For example, DD defines 1-minimality [24]. That is, $p \in \mathbb{P}$ is 1-minimal if any variant p' derived from p by removing a single element from p does not pass the property test, *i.e.*, $\psi(p') = \mathsf{false}$. HDD also defines a similar notion [16]: A program is 1-tree-minimality if any node of the tree representation of the program cannot be further simplified by the reducer.

3.2 Delta Debugging Algorithm

DD is integral to `Perses`, and we briefly introduce its reduction algorithm `ddmin`. Given an input and a property ψ, DD first converts the input into a list L of elements. Then `ddmin` determines a subset

[3] Note that the search space $\mathbb{P}$ is not the universe of all programs. As aforementioned in Section 1, it is defined by the concrete reduction algorithm over P, and $P \in \mathbb{P}$.

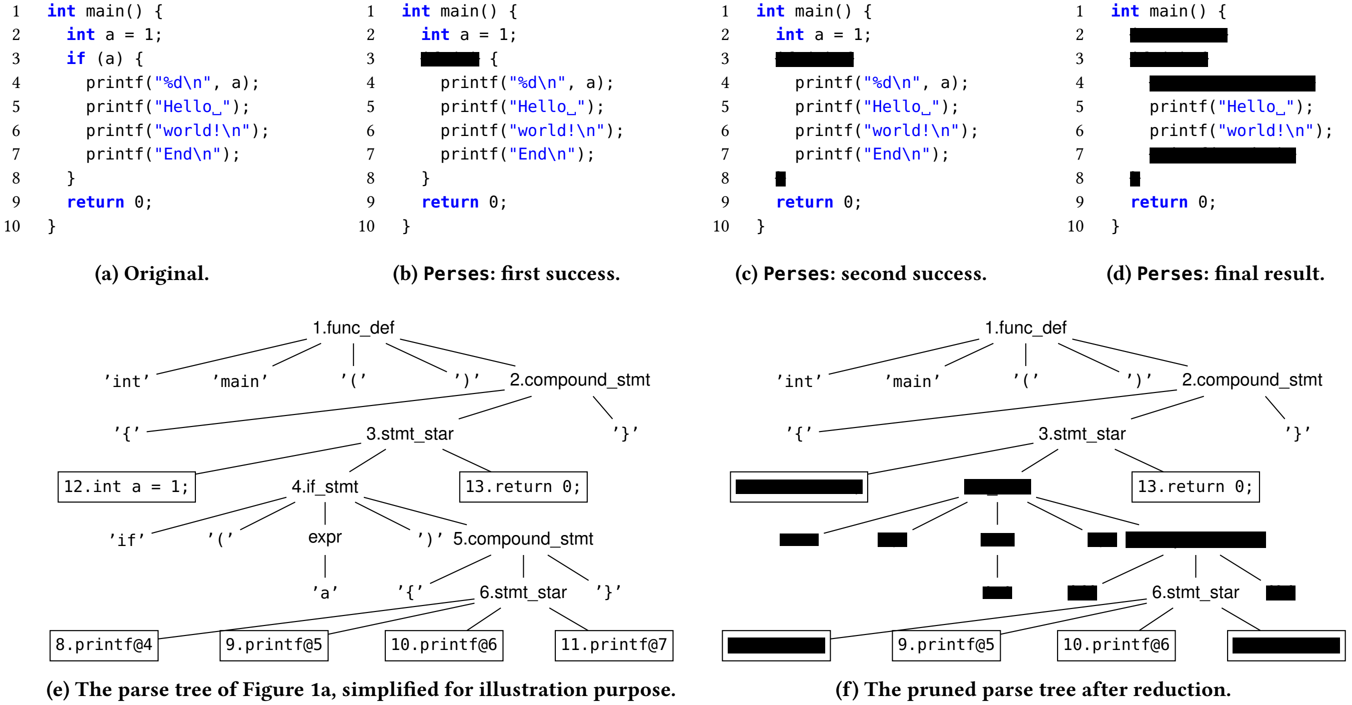

Figure 1: An example program.

of L such that no one element can be deleted from L while preserving ψ, *i.e.*, the reduced L is 1-minimal. It involves three steps.

Step 1: Split L into n partitions. For each partition u, test if u only without the other partitions can preserve the property. If yes, remove the complement of u from L, and resume at Step 1.

Step 2: Test if the complement of each partition u preserves the property. If yes, remove u from L, and resume at Step 1.

Step 3: Try to split each remaining partition into halves. That is, increase the number of partitions from n to $2n$. Then resume at Step 1 with the newly split smaller partitions. If each partitions cannot be further split, the remaining elements in L are the reduced result, and `ddmin` terminates.

3.3 Input Grammar Forms

`Perses` takes as input language grammars specified in Backus-Naur Form (BNF) notation. Unless otherwise stated, we always assume the grammars are expressed in BNF notation. `Perses` also supports the context-free grammar rules with three additional quantifier over terminals and nonterminals.

Kleene Star **($*$).** The quantified terminal or nonterminal should occur zero or multiple times. For example, A* can generate empty strings, 'A', 'AA', and so on.

Kleene Plus **($+$).** The quantified terminal or nonterminal should occur one or multiple times. For example, A+ is similar to A* except that A+ does not accept empty strings.

Optional **($?$).** The quantified terminal or nonterminal should occur either zero times or once. For example, A? accepts either empty strings or 'A'.

In particular, a grammar rule is defined as a *quantified rule* iff at least one of its right-hand side symbols is equipped with a quantifier. The three quantifiers simplify the grammar representation and have been widely used in many popular parser generators such as ANTLR [2] and JavaCC [7]. `Perses` leverages the three quantifiers to perform syntax guided program reduction. We say that a nonterminal A is *quantifiable* if A could be transitively described by a quantified rule. For instance, "$B ::= D^*$" is a quantified rule and the nonterminal B is quantifiable. Suppose that we have another rule "$A ::= BC$". The nonterminal A is also quantifiable since A could be transitively described by a quantified rule "$A ::= D^*C$" by replacing the B symbol with "D^*". Next, we formally introduce the grammar normal form used in `Perses`.

Definition 3.1 (Perses Normal Form). *A context-free grammar CFG is in Perses normal form (PNF) if all its production rules are of the following form:*

(i) $A ::= B_1 B_2 \dots B_n$, *or*
(ii) $A ::= B_1*$, *or*
(iii) $A ::= B_1+$, *or*
(iv) $A ::= B_1?$, *or*
(v) $S ::= \epsilon$,

where S denotes the start symbol, A denotes a nonterminal, B_i denotes either a terminal or a nonterminal for all $i \in [1, n]$, and $n > 1$. Moreover, all quantifiable nonterminals are transitively described by at least *one quantified rule.*

The PNF could be viewed as a restricted form of the extended Backus-Naur form. Every context-free grammar could be normalized to PNF. The normalization algorithm is detailed in Section 4.1.

Intuitively, rule (i) in Definition 3.1 refers to any non-epsilon rule in a context-free grammar. The epsilon rules could be safely removed during normalization [1]. Rules (ii)-(iii) correspond to any recursive rules with either left- or right-recursion. The normalization algorithm finds at least one such recursive rules for each quantifiable nonterminal. Rule (iv) could be distilled from the rules with optional terminal or nonterminals. Figure 2 shows the grammar in PNF for the program in Figure 1a. For illustration purpose it is simplified by covering a small subset of the C programming language.

⟨*func_def*⟩	::=	⟨*type*⟩ ⟨*identifier*⟩ '(' ')' ⟨*compound_stmt*⟩
⟨*stmt*⟩	::=	⟨*if_stmt*⟩
	|	⟨*decl_stmt*⟩
	|	⟨*expr_stmt*⟩
	|	⟨*compound_stmt*⟩
⟨*if_stmt*⟩	::=	⟨*cond_plus*⟩ ⟨*decl_stmt*⟩
	|	⟨*cond_plus*⟩ ⟨*expr_stmt*⟩
	|	⟨*cond_plus*⟩ ⟨*compound_stmt*⟩
⟨*cond_plus*⟩	::=	⟨*if_cond*⟩ +
⟨*if_cond*⟩	::=	'if' '(' ⟨*expr*⟩ ')'
⟨*decl_stmt*⟩	::=	⟨*type*⟩ ⟨*identifier*⟩ '=' ⟨*expr*⟩ ';'
⟨*expr_stmt*⟩	::=	⟨*expr*⟩ ';'
⟨*compound_stmt*⟩	::=	'{' ⟨*stmt_star*⟩ '}'
⟨*stmt_star*⟩	::=	⟨*stmt*⟩ *

Figure 2: A normalized grammar in PNF for the program in Figure 1a. We omit the rules of nonterminals ⟨*type*⟩, ⟨*identifier*⟩ and ⟨*expr*⟩ for brevity.

4 APPROACH

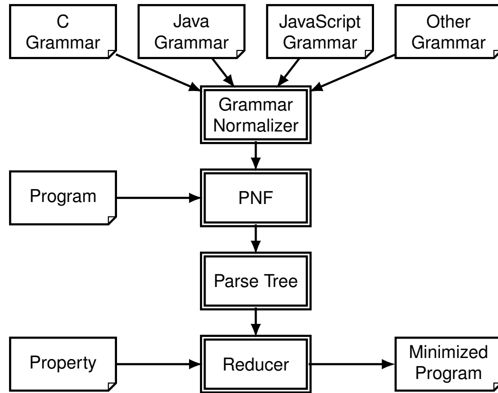

Figure 3: Overall Workflow of Perses.

Figure 3 shows the overview workflow of `Perses`. Conceptually, `Perses` supports any unambiguous context-free grammars. To support a specific language, `Perses` converts its grammar to the normal form *i.e.*, Definition 3.1. When `Perses` is used to reduce a program P *w.r.t.* a property. `Perses` first uses the normalized grammar to parse P into a parse tree T. Then T is fed to the core reduction algorithm consisting of a set of program transformations. Guided by the property test, the reducer gradually deletes irrelevant elements from the tree, and output the minimized program upon termination.

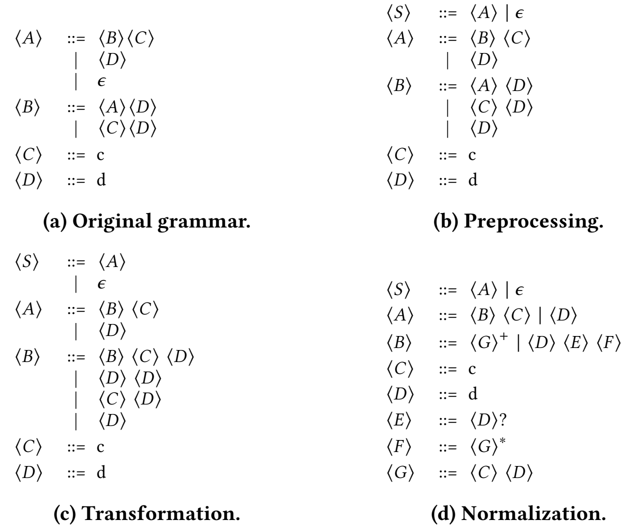

Figure 4: An illustrative example for PNF normalization.

4.1 PNF Normalization

This section describes our normalization algorithm for computing the `Perses` normal form (PNF). Our normalization algorithm takes as input any context-free grammar in BNF notation and outputs an equivalent grammar in PNF. The basic idea of our normalization algorithm is to compute quantified rules based on grammar transformations. We notice that the "∗" and "+" quantifiers correspond to the sequence repetitions in regular expressions. The repetition is typically specified using left- or right-linear context-free grammars. As a result, we could compute the "∗" and "+" quantified rules from those left- or right-recursion rules. The "?" quantified rules could be computed by comparing the differences of each pair of rules.

Our normalization algorithm involves three essential steps. We briefly describe these steps with an example in Figure 4. Figure 4a shows the original grammar where ⟨*A*⟩ is the start symbol. The original grammar is not in the PNF. In particular, the nonterminal ⟨*B*⟩ is quantifiable since it could potentially be described by a "+"-quantified rule obtainable from the left-recursion rule "⟨*B*⟩ ::= ⟨*B*⟩⟨*C*⟩⟨*D*⟩". However, the grammar in Figure 4a does not contain any quantified rule. We then discuss PNF normalization as follows.

- *Preprocessing.* Our preprocessing procedure eliminates the ϵ productions and unreachable rules. We also assign a random ordering to all nonterminals in the grammar. The preprocessed grammar is given in Figure 4b. In particular, ⟨*S*⟩ becomes the new start symbol.
- *Transformation.* To facilitate our next normalization step, we prefer left- or right-recursive productions. During the transformation procedure, for each nonterminal we try to describe it by a left- or right-recursive production rule according to the original grammar. Figure 4c shows the example after the transformation, nonterminal ⟨*B*⟩ is transformed into a left-recursion form.
- *Normalization.* In the normalization step, the corresponding left- or right-recursive production rules are transformed into quantified rules. Once all quantifiable nonterminals have been transitively described using quantified rules, We convert the result

Algorithm 1: PNF Normalization — Normalization(G)

Input: G: a context-free grammar in BNF notation.
Output: G': an equivalent grammar in PNF.
1 Preprocess (G)
2 $G_1 \leftarrow$ GrammarNormalizationLeft(G)
3 $G' \leftarrow$ GrammarNormalizationRight(G_1)
4 **return** G'
5 **Function** GrammarNormalizationLeft(G):
  Input: G: a context-free grammar in BNF notation.
  Output: G: an equivalent grammar without quantifiable nonterminals in form of left recursion.
6 let G_{cfg} be an empty digraph
7 **foreach** $N_i \rightarrow N_j \alpha$ **do** $G_{cfg} \leftarrow G_{cfg} \cup \{(N_i, N_j)\}$
8 SCC $\leftarrow$ COMPUTE_SCC(G_{cfg})
9 **foreach** $scc_i \in SCC$ **do**
10   $G_i \leftarrow \varnothing$
11   **foreach** $A \in scc_i$ **do** insert all rules of the form $A \rightarrow \alpha$ to G_i
12   $G'_i \leftarrow$ GrammarTransformationLeft(G_i)
13   $G'_i \leftarrow$ QuantifierIntroductionLeft(G'_i)
14   $G \leftarrow (G \backslash G_i) \cup G'_i$
15 **return** G
16 **Function** GrammarTransformationLeft(G):
  Input: G: a context-free grammar in BNF notation.
  Output: G: an equivalent grammar with direct left recursion
17 **foreach** *nonterminal* $N_i \in G$ **do**
18   **repeat**
19   **foreach** *rule* $N_i \rightarrow N_j \alpha_i \in G$ **do**
20   **if** $j < i$ **then**
21   remove $N_i \rightarrow N_j \alpha_i$ from G
22   **foreach** *rule* $N_j \rightarrow \alpha_j \in G$ **do**
23   add $N_i \rightarrow \alpha_j \alpha_i$ to G
24   **until** *Grammar G remains unchanged.*
25 **return** G
26 **Function** QuantifierIntroductionLeft(G):
  Input: G: a context-free grammar in BNF notation.
  Output: G: an equivalent grammar with quantifiers.
27 **foreach** *nonterminal* $N_i \in G$ **do**
28   StarIntroductionLeft (G, N_i)
29   **foreach** $N_i \rightarrow \alpha_1 \alpha' (\alpha')^* \alpha_2$ **do**
30   remove $N_i \rightarrow \alpha_1 \alpha' (\alpha')^* \alpha_2$ from G
31   let U_1, U_2 be new auxiliary nonterminals
32    $G \leftarrow G \cup \{N_i \rightarrow \alpha_1 U_1 \alpha_2,\ U_1 \rightarrow U_2^+,\ U_2 \rightarrow \alpha'\}$
33   **foreach** *pair of* $(N_i \rightarrow \alpha_i,\ N_i \rightarrow \alpha_j)$, *where* $|\alpha_i| \le |\alpha_j|$ **do**
34   **if** $\alpha_j = \alpha_1 \alpha' \alpha_2$ *and* $\alpha_i = \alpha_1 \alpha_2$ **then**
35   remove $N_i \rightarrow \alpha_i$ and $N_i \rightarrow \alpha_j$ from G
36   let U_3, U_4 be new auxiliary nonterminals
37     $G \leftarrow G \cup \{N_i \rightarrow \alpha_1 U_3 \alpha_2,\ U_3 \rightarrow U_4?,\ U_4 \rightarrow \alpha'\}$
38 **return** G
39 **Function** StarIntroductionLeft(G, N):
  Input: G: a set of grammar productions
  Input: N: a nonterminal
  Output: G: a set of grammar productions with $*$-quantified rules
40 $A \leftarrow \varnothing$ and $B \leftarrow \varnothing$
41 **foreach** *rule* $N \rightarrow \alpha \in G$ **do**
42   **if** $\alpha = N\alpha_1$ **then** $A \leftarrow A \cup \{\alpha_1\}$
43   **else** $B \leftarrow B \cup \{\alpha\}$
44   remove $N \rightarrow \alpha$ from G
45 **foreach** $b_i \in B$ **do**
46   denote set A as $\{a_1, a_2, \ldots, a_j\}$
47   let U_1, U_2 be new auxiliary nonterminals
48   $G \leftarrow G \cup \{N \leftarrow b_i U_1,\ U_1 \rightarrow U_2^*,\ U_2 \rightarrow a_1 \mid a_2 \mid \ldots \mid a_j\}$
49 **return** G

grammar to PNF by introducing necessary auxiliary nonterminals. Figure 4d gives the example grammar in PNF.

Algorithm Overview. Algorithm 1 describes our PNF normalization algorithm, where each N_i represents a nonterminal and each α_i represents a sequence of terminals and nonterminals.

Algorithm 2: The Main Algorithm — Reduce(P, ψ)

Input: P: the program to be reduced.
Input: $\psi : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved.
Output: A minimum program $p \in \mathbb{P}$ *s.t.* $\psi(p)$
1 best $\leftarrow$ ParseTree(P)
2 worklist $\leftarrow$ {RootNode(best)}
3 **while** |worklist| > 0 **do**
4 largest $\leftarrow$ GetAndRemoveLargestFrom(worklist)
5 **if** largest *is Kleene-Star Node* **then**
6   (best, pending) $\leftarrow$ ReduceStar(best, ψ, largest)
7 **else if** largest *is Kleene-Plus Node* **then**
8   (best, pending) $\leftarrow$ ReducePlus(best, ψ, largest)
9 **else if** largest *is Optional Node* **then**
10   (best, pending) $\leftarrow$ ReduceStar(best, ψ, largest)
11 **else if** largest *is Regular Rule Node* **then**
12   (best, pending) $\leftarrow$ ReduceRegular(best, ψ, largest)
13 **else continue** // Skip token nodes
14 worklist $\leftarrow$ worklist $\cup$ pending
15 **return** best

In particular, lines 1-4 describe the main normalization algorithm. After preprocessing, many quantifiable nonterminals may still not be transitively described using quantified rules. Our algorithm then processes the grammar in two passes. The first pass handles the left recursion rules, and the second pass handles the right recursion rules. In principle, the two passes are almost identical. Therefore, we only discuss the first pass.

On line 5, the function GrammarNormalizationLeft processes all left recursion rules. We first build a directed graph G_{cfg} by constructing edges (N_i, N_j) from a nonterminal N_i to the leftmost nonterminal N_j according to its productions (lines 6-7). Then, we find the strongly connected components (SCC) in G_{cfg} using Tarjan's algorithm (line 8). Each SCC corresponds to at least one recursive rule. Therefore, the algorithm performs grammar transformation and grammar normalization for each SCC (lines 12-13).

Grammar Transformation. The function on line 16 performs grammar transformation. In particular, it transforms any indirect recursion rule to an equivalent rule with left recursion. It utilizes the assigned ordering (line 20), and iteratively replaces the lower order nonterminals with its production rules (line 21-23). The procedure terminates when there is no indirect recursion rules *w.r.t.* the ordering (line 24).

Grammar Normalization. The grammar normalization procedure on line 26 introduces the quantifiers and outputs an equivalent grammar in the PNF form. Specifically, it converts the recursive rules into "$*$"-quantified rules (line 39), "+"-quantified rules (line 29-32), or "?"-quantified rules (line 33-37). Lines 32, 37 and 48 introduce the quantified rules to the output grammar. Finally, all rules are in the PNF according to Definition 3.1. The transformation procedure fully exploits all dependencies among the nonterminals, and the normalization procedure introduces the quantified rules *w.r.t.* the definition. It is straightforward to see that Algorithm 1 preserves the grammar equivalence.

4.2 Main Reduction Algorithm

Algorithm 2 lists the main procedure to reduce a program P *w.r.t.* ψ. The output is a minimum variant derived from P that still passes ψ. Initially, P is converted into a parse tree best with the PNF grammar. Then all the following reduction is performed on this parse tree.

The overall reduction process is a prioritized traversal of best. Algorithm 2 maintains a worklist of tree nodes pending reduction, and each time pops out the node with the most tokens to reduce on line 4. We treat quantified nodes and regular rule nodes differently. For example, if the node largest is a Kleene-Star node, we reduce it with the function `ReduceStar` on line 6 that returns a pair of which the first is the reduced result and the second is the pending nodes for future reduction. The other nodes are treated similarly only with different functions. When reducing largest finishes, we update the worklist by adding the pending nodes on line 14, and proceed to the next largest node in worklist.

4.3 Reducing Quantified Nodes

For a Kleene-Star, Kleene-Plus or Optional node, we use `ddmin` to delete its children, as each child is independent of the others in terms of syntax validity, which fits in the assumption of DD well.

Reducing Kleene-Star and Optional Nodes. We treat Kleene-Star and Optional nodes in the same way, as Optional nodes are a special case of Kleene-Star. Algorithm 3 shows the general reduction procedure. We use `ddmin` to perform Delta Debugging on all the children of *tree* to remove the children that are irrelevant to ψ. At last, we return the minimized tree by removing irrelevant nodes from *tree* and the remaining children of *node* as the result.

Algorithm 3: `ReduceStar`(*tree*, ψ, *node*)

Input: *tree*: the parse tree to be reduced.
Input: $\psi : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved.
Input: *node*: the parse tree node to be reduced.
Output: (best, pending): best is the minimum tree by reducing *tree*, and pending is a set of remaining descendants of *node*.

```
1 all ← Children(node)
2 remaining ← ddmin(all, ψ)
3 best ← tree.CopyAndRemove(all \ remaining)
4 return (best, remaining)
```

Reducing Kleene-Plus Node. Reducing Kleene-Plus nodes is similar to reducing Kleene-Star nodes, *i.e.*, Algorithm 3. It also uses `ddmin` as the underlying reduction algorithm. The main difference is that when reducing the children of *node*, we need to maintain one constraint induced by the semantics of Kleene-Plus — at least one child of *node* is not deleted. For example, when we are about to test a variant by deleting all the children of *node*, we need to keep one child, *e.g.*, the first child, in order to avoid syntax errors.

4.4 Reducing Regular Rule Nodes

Algorithm 4 shows how a regular rule node is reduced. The general idea is to replace the tree node *node* that is being reduced with one of its *compatible* descendants. The compatibility is determined based on the subsume relation defined as follows.

Definition 4.1 (Subsume Relation). *Given two symbols A and B (terminals or non-terminals) in a grammar, B is subsumed by A (denoted as B <: A) if one of the following conditions holds:*

- *A = B*
- *B can be derived from A*

For example, ⟨`stmt`⟩ <: ⟨`stmt`⟩, and ⟨`if_stmt`⟩ <: ⟨`stmt`⟩. Intuitively, ⟨`if_stmt`⟩ <: ⟨`stmt`⟩ means that any parse tree of ⟨`if_stmt`⟩ can be syntactically safe to be used in the context where a parse tree of ⟨`stmt`⟩ is expected. We define three auxiliary functions related to rule types for a node *n*.

`Rule`(*n*) returns the production rule that creates the node *n*. For example, the rule of the node '`4.if_stmt`' in Figure 1e is ⟨`if_stmt`⟩.

`ExpectedRule`(*n*) returns the expected production rule at the position of *n* in the context of its parent `Parent`(*n*). For example, the expected production rule for node '`4.if_stmt`' in Figure 1e is ⟨`stmt`⟩, because its parent '`3.stmt_star`' expects each of its children to be ⟨`stmt`⟩.

`QuantifiedRule`(*n*) only applies to Kleene nodes (*n* is either Kleene-Star or Kleene-Plus). It returns the quantified production rule of *n*. For example, `QuantifiedRule` returns ⟨`stmt`⟩ for the node '`3.stmt_star`', because it expects a list of ⟨`stmt`⟩ children.

With the subsume relation and auxiliary functions, we can identify two types of compatible descendants to replace *node*.

Regular Nodes (lines 2-5) A regular descendant node *n* of *node* is a replacement candidate if its production rule `Rule`(*n*) is subsumed by that of *node*, *i.e.*, `Rule`(*n*) <: `ExpectedRule`(*node*). Take Figure 1e as an example. Assume that we are reducing '`4.if_stmt`'. Its expected rule type is ⟨`stmt`⟩ as aforementioned. Then its descendant '`4.compound_stmt`' is a compatible node, as the rule type of this descendant is ⟨`compound_stmt`⟩ which is a kind of statements.

Kleene Node (lines 6-10) If the parent of *node* is either Kleene-Star or Kleene-Plus, then a descendant *n* is compatible if *n* is also a Kleene node and its quantified rule type is subsumed by the expected rule of *node*, namely `QuantifiedRule`(*n*) <: `ExpectedRule`(*node*).
For example, in Figure 1e, the node '`6.stmt_star`' can replace '`4.if_stmt`', because the parent '`3.stmt_star`' of '`4.if_stmt`' quantifies a list of ⟨`stmt`⟩ nodes, and using '`6.stmt_star`' to replace '`4.if_stmt`' still maintains the syntactical invariant, that is, '`3.stmt_star`' still quantifies a list of ⟨`stmt`⟩ nodes, though the newly added node '`6.stmt_star`' introduces a layer of indirection.

Note that the number of compatible nodes for *node* can be enormous. In order to limit the search space, we require the path L between *node* (exclusive) and its compatible node *n* (inclusive) satisfy the following two constraints: (1) The number of nodes in L is bounded; (2) There is no other compatible node on L before *n*. In other words, *n* is the first compatible node on L. The function `BoundedBFS` on line 18 implements these two constraints. In this work, we use 4 as the maximum length of L, *i.e.*, line 4 and line 9.

4.5 Fixpoint Reduction Mode

A single run of the reduction function `Reduce` does not guarantee that the reduced program is 1-tree-minimal (*cf.* Section 3.1), because the deletion of one node may enable the deletion of another node.

For example, assume two functions `foo` and *main*; `foo` has more tokens than `main`; `main` calls `foo`; `main` has the property of interest, and `foo` is irrelevant to this property. When `Reduce` is reducing this program, it first reduces `foo`. However `foo` is referenced by `main`, so it cannot be completely deleted. Later, when `main` is being

Algorithm 4: ReduceRegular(*tree*, ψ, *node*)

```
    Input: tree: the parse tree to be reduced.
    Input: ψ : ℙ → 𝔹: the property to be preserved.
    Input: node: the parse tree node to be reduced.
    Output: (best, pending): best is the minimum tree by reducing tree, and
            pending is a set of remaining descendants of node.
1   candidates ← ∅
2   begin searching for replacement candidates
3       subsume_pred ← λn.Rule(n) <: ExpectedRule(node)
4       replacement_candidates ← BoundedBFS(node, subsume_pred, 4)
5       candidates ← candidates ∪ replacement_candidates
6   if IsKleene(Parent(node)) then
7       kleene_pred ← λn.IsKleene(n)
8                   ∧QuantifiedRule(n) <: ExpectedRule(node)
9       quantified_candidates ← BoundedBFS(node, kleene_pred, 4)
10      candidates ← candidates ∪ quantified_candidates
11  best ← node
12  foreach c ∈ candidates do
13      t ← tree.CopyAndReplace(node, c)
14      if ψ(t) ∧ |t| < |tree.CopyAndReplace(node, best)| then
15          best ← {c}
16  if best = node then return (tree, Children(node))
17  else return (tree.CopyAndReplace(node, best), best)
18  Function BoundedBFS(node, pred, max_depth):
        Input: node: the starting node of breadth-first search
        Input: pred : TreeNodes → 𝔹: predicate to match tree nodes.
        Input: max_depth: depth bound.
        Output: result: the matched tree nodes.
19      Queue queue ← Children(node)
20      result ← ∅
21      while |queue| > 0 ∧ max_depth > 0 do
22          max_depth ← max_depth − 1
23          queue_size ← |queue|
24          for i ← 0 to queue_size do
25              n ← Dequeue(queue)
26              if pred (n) then
27                  result ← result ∪ {n}
28                  continue;
29              if max_depth > 0 then
30                  foreach c ∈ Children(n) do Enqueue(queue, c)
31      return result
```

reduced, the call to `foo` is deleted. Then `foo` can be removed. But it has been visited so `Reduce` will not delete it in the current run.

Therefore, similar to HDD* [16], we propose a fixpoint reduction mode, in which `Reduce` is repeatedly applied to the reduced result until no more tree nodes can be removed from the result. And the final result will be 1-tree-minimal.

5 EVALUATION

We evaluate `Perses` with 20 large C programs that trigger bugs in GCC and Clang, and compare it with DD, HDD, and C-Reduce. On average, `Perses` outperforms the other reducers nearly in every aspect, *e.g.*, reduced file size (55-98% smaller except C-Reduce), number of property tests (47-93% fewer), reduction time (34-77% shorter exception MultiDelta), and reduction speed (1.1-2.6x speedup), .

To demonstrate the generality of `Perses`, we instantiate `Perses` with an ANTLR Java grammar. The evaluation on six benchmarks confirms again that `Perses` outperforms HDD in various metrics. All experiments are conducted on a Ubuntu machine with an Intel Core i7-4770 CPU and 16 GB memory.

5.1 Evaluation on C Programs

Benchmark Collection. The benchmark C programs are collected from the official bug repositories of two mainstream C compilers (GCC and Clang). We have randomly selected 20 recent bug reports and requested the unreduced testcase from the original reporter. The selected bug reports include both crash and miscompilation bugs. Moreover, all selected bug reports are reproducible *w.r.t.* at least one stable release of the two compilers.

Tools for Comparison. We run `Perses` and $\texttt{Perses}^F$ (`Perses` in fixpoint mode) in comparison with the following tools: **Delta** [15] (a line-based Delta Debugging tool), **Delta**^{F} (Delta in fixpoint mode), **MultiDelta** [15] (A variant of Delta that is aware of blocks in programs), **C-Reduce** [17], **HDD** [16] and **HDD**^{F} (HDD in fixpoint mode).

5.1.1 Reduction Quality.

Table 1 shows the reduction quality results. It also gives the original token count of a test program and that of the minimized program by a reducer. In general, `Perses` produces much smaller results than the other reducers except C-Reduce (**55-98% smaller**). The row *size* in Table 3 shows the average improvement of `Perses` and $\texttt{Perses}^F$ over other reducers. For example, the size of the result by $\texttt{Perses}^F$ is only 2% of that of Delta, and 45% of HDD^F. Compared to C-Reduce, `Perses` produces 167 tokens more than C-Reduce on average. However, this is expected, because C-Reduce is a C/C++ language specialized reducer which performs aggressive *semantic* program transformations.

5.1.2 Reduction Efficiency.

Efficiency is another key criterion to evaluate a reducer. Table 2 and a part of Table 1 shows the details of efficiency-wise data.[4] This section further quantifies efficiency in the following three different metrics.

Number of Property Tests. This is the metric used in [16]. If the property test runs in constant time, then this metric can reliably reflect the runtime complexity of reducers. Table 1 lists this metric. $\texttt{Perses}^F$ runs property tests 5095 times on average, which is **2.26x** fewer than MultiDelta, **15.4x** fewer than Delta^F, **5.37x** fewer than C-Reduce and **3.31x** fewer than HDD^F. The row *#tests* in Table 3 lists the detailed improvement.

Reduction Time. This measures how long a reducer takes to terminate. It depends on the number of property tests and the time of each property test. The row *time* in Table 3 shows the time ratio between `Perses` and the other reducers. Except MultiDelta, `Perses` constantly takes shorter time to terminate (34-77% shorter). As far as MultiDelta, though taking similar time as MultiDelta. `Perses` produces much smaller results than MultiDelta.

Reduction Speed. Reduction speed is the number of tokens that a reducer can delete per second. The row *speed* in Table 3 lists the speed ratio between `Perses` and the other tools. Both `Perses` and $\texttt{Perses}^F$ run faster than the others. Especially `Perses` is the fastest (1.5-3.6x faster than the rest).

[4] In Table 1 Delta and Delta^F performs only two property tests, and in Table 2 they spend zero seconds on this benchmark. This is because all the tokens of the original test program are placed in a single line, and the line-oriented strategy of Delta just treats the program as a single line, which demonstrates the limitation of DD.

Table 1: Size information of original and reduced programs, and number of property tests.

Bug	O (#)	Delta		MultiDelta		Delta^F		C-Reduce		HDD		HDD^F		`Perses`		`Perses`^{F}	
		R (#)	Q (#)	R (#)	Q (#)	R (#)	Q (#)	R (#)	Q (#)	R (#)	Q (#)	R (#)	Q (#)	R (#)	Q (#)	R (#)	Q (#)
clang-22382	21,068	4,426	4,741	3,086	8,061	3,011	12,259	79	19,569	1,554	7,681	221	9,362	1,479	2,517	152	3,601
clang-22704	184,444	5,318	12,847	3,099	11,524	2,732	22,499	58	19,339	1,312	6,562	105	7,783	1,208	2,374	106	3,163
clang-23309	38,647	5,097	7,620	5,861	10,355	3,641	17,758	132	32,875	2,418	11,370	1,197	25,097	2,030	3,115	456	7,394
clang-23353	30,196	3,681	5,310	2,365	6,846	1,493	11,164	67	17,400	1,535	7,977	143	9,711	1,317	2,636	117	3,565
clang-25900	78,960	4,948	14,023	2,514	6,718	2,052	22,952	105	21,448	1,686	7,774	496	15,391	1,387	2,325	265	5,145
clang-26760	209,577	10,992	58,786	9,227	13,842	6,427	67,623	47	19,868	1,386	7,102	424	11,583	1,299	2,198	124	3,489
clang-27137	174,538	18,008	29,861	10,224	21,919	5,914	54,991	52	31,036	4,772	22,561	664	36,554	4,287	5,937	276	8,466
clang-27747	173,840	7,519	19,443	1,633	4,941	919	30,016	116	20,852	775	3,919	312	8,474	863	1,549	121	2,662
clang-31259	48,799	3,753	8,410	3,225	9,791	2,339	14,743	132	28,445	1,863	8,648	697	13,712	1,340	2,357	328	4,726
gcc-59903	57,581	6,592	8,242	7,152	12,276	5,699	19,759	103	54,443	2,086	9,551	754	17,575	2,228	3,241	206	6,509
gcc-60116	75,224	45,160	50,314	8,110	11,739	20,884	574,790	127	40,698	2,979	13,594	1,543	30,523	2,459	3,604	379	7,254
gcc-61383	32,449	14,728	24,857	5,322	8,860	2,817	139,024	92	37,402	2,117	10,354	586	19,969	1,855	2,859	293	6,671
gcc-61917	85,359	35,428	55,073	5,072	10,801	4,788	409,674	63	31,251	1,678	7,940	358	11,688	1,751	2,786	144	4,298
gcc-64990	148,931	5,504	14,056	9,787	25,203	3,317	26,118	82	23,535	1,924	9,195	519	14,972	1,854	3,179	478	6,112
gcc-65383	43,942	5,486	7,712	4,086	9,686	3,724	14,354	83	24,537	1,733	8,235	308	11,926	1,353	2,327	185	3,769
gcc-66186	47,481	5,701	8,575	3,949	11,228	3,680	21,287	133	30,932	1,898	8,928	620	15,933	1,656	2,542	330	5,629
gcc-66375	65,488	5,599	10,071	2,921	9,299	2,948	23,442	60	29,041	2,153	10,040	860	19,890	1,635	2,688	427	5,624
gcc-70127	154,816	7,783	20,412	6,154	14,714	2,671	38,771	84	27,268	2,505	11,895	658	19,324	1,534	2,779	317	5,024
gcc-70586	212,259	12,225	28,960	11,019	19,247	6,272	44,879	145	29,520	3,257	15,247	988	34,330	2,469	3,696	393	7,479
gcc-71626	6,133	6,133	2	235	4,078	6,133	2	46	7,730	538	3,504	53	3,931	596	1,193	53	1,335
mean	94,486	10,704	19,465	5,252	11,556	4,573	78,305	90	27,359	2,008	9,603	575	16,886	1,730	2,795	257	5,095
median	70,356	5,917	13,435	4,579	10,578	3,479	23,197	83	27,856	1,880	8,788	552	15,181	1,584	2,662	270	5,084

Columns O, R, Q list the number of original tokens, the number of tokens after reduction, and the number of queries, respectively.

Table 2: Reduction time and reduction speed.

Bug	Delta		MultiDelta		Delta^F		C-Reduce		HDD		HDD^F		`Perses`		`Perses`^{F}	
	T (s)	E (#/s)	T (s)	E (#/s)	T (s)	E (#/s)	T (s)	E (#/s)	T (s)	E (#/s)	T (s)	E (#/s)	T (s)	E (#/s)	T (s)	E (#/s)
clang-22382	471	35	462	39	710	25	904	23	532	37	588	35	378	52	442	47
clang-22704	2,556	70	1,991	91	2,915	62	2,182	85	2,816	65	2,873	64	1,143	160	1,205	153
clang-23309	1,375	24	1,096	30	1,751	20	5,586	7	1,998	18	3,733	10	831	44	1,394	27
clang-23353	644	41	570	49	804	36	1,010	30	1,046	27	1,112	27	476	61	543	55
clang-25900	2,707	27	726	105	3,111	25	1,575	50	890	87	1,253	63	522	149	776	101
clang-26760	14,248	14	2,613	77	13,700	15	2,868	73	2,459	85	2,665	78	1,200	174	1,365	153
clang-27137	16,382	10	6,706	25	18,501	9	7,787	22	16,122	11	18,430	9	7,150	24	8,296	21
clang-27747	7,107	23	635	271	7,640	23	1,456	119	638	271	912	190	601	288	797	218
clang-31259	1,446	31	1,149	40	1,674	28	2,887	17	2,078	23	3,504	14	748	63	2,155	22
gcc-59903	2,813	18	2,918	17	3,851	13	4,632	12	1,980	28	2,352	24	1,116	50	1,448	40
gcc-60116	8,955	3	1,330	50	57,722	1	4,888	15	2,704	27	4,441	17	1,409	52	2,475	30
gcc-61383	2,631	7	1,001	27	7,964	4	3,799	9	1,709	18	3,237	10	805	38	1,959	16
gcc-61917	10,732	5	1,576	51	33,502	2	2,847	30	1,565	53	1,685	50	1,043	80	1,154	74
gcc-64990	5,875	24	4,446	31	6,402	23	3,292	45	4,572	32	4,931	30	1,901	77	2,914	51
gcc-65383	904	43	703	57	1,162	35	2,295	19	1,069	39	1,365	32	474	90	660	66
gcc-66186	1,363	31	2,082	21	1,777	25	5,317	9	2,368	19	4,189	11	1,228	37	3,220	15
gcc-66375	3,553	17	2,686	23	4,407	14	8,442	8	4,325	15	8,998	7	1,297	49	3,136	21
gcc-70127	9,044	16	3,265	46	9,822	15	6,191	25	8,495	18	14,101	11	1,827	84	3,580	43
gcc-70586	16,733	12	5,955	34	16,791	12	5,312	40	9,416	22	12,612	17	3,398	62	6,376	33
gcc-71626	0	—	133	44	0	—	232	26	163	34	174	35	68	81	74	82
mean	5,477	24	2,102	56	9,710	20	3,675	33	3,347	46	4,658	37	1,381	86	2,198	63
median	2,760	23	1,453	42	4,129	20	3,090	24	2,038	28	3,055	26	1,080	63	1,421	45

Columns T, E list the time of reduction, and the efficiency in tokens per second, respectively.

Table 3: Ratio of various metrics between `Perses` and others.

		Delta	MultiDelta	Delta^F	C-Reduce	HDD	HDD^F
size	`Perses`	16%	33%	38%	1,922%	86%	300%
	`Perses`^{F}	2%	5%	6%	285%	13%	45%
#tests	`Perses`	14%	24%	4%	10%	29%	17%
	`Perses`^{F}	26%	44%	7%	19%	53%	30%
time	`Perses`	25%	66%	14%	38%	41%	30%
	`Perses`^{F}	40%	105%	23%	60%	66%	47%
speed	`Perses`	3.6x	1.5x	4.3x	2.6x	1.9x	2.3x
	`Perses`^{F}	2.6x	1.1x	3.2x	1.9x	1.4x	1.7x

5.2 Evaluation on Java Programs

We demonstrate the generality of `Perses` by instantiating it with an ANTLR Java grammar to reduce Java programs. To the best of our knowledge, `Perses` is by far the most effective program reducer for Java, as there is no C-Reduce-like tool for Java.

We manually checked the official bug repositories of OpenJDK and Eclipse Compiler for Java (ECJ), and collected 5 bugs that can be reproduced and have *unreduced* input programs attached in the bug reports. Table 4 shows the reduction results. `Perses` outperforms HDD. In general, `Perses` is **2.07x** faster, produces **1.13x** smaller results, and reduces **3.99x** fewer queries than HDD.

Table 4: Results of Java benchmarks.

Bug	O (#)	HDD^F			Perses^F		
		R (#)	Q (#)	T (s)	R (#)	Q (#)	T (s)
ecj-352665	1,142	180	2,921	364	157	834	166
ecj-361938	438	16	114	10	16	29	9
ecj-404146	348	170	1,787	147	155	403	78
jdk-8068399	447	81	998	127	69	207	60
jdk-8145466	462	53	368	42	44	77	20
Total	2,837	500	6,188	690	441	1,550	333

Columns O, R, Q and T list the number of original tokens, the number of tokens after reduction, the number of queries, and the reduction time, respectively.

6 DISCUSSION

Generality to Programming Languages. Perses is a general reduction framework, and it makes *no* assumption about specific programming languages. It takes as input language grammars, and thus is able to exploit any language-specific syntactical properties (*e.g.*, which nodes can be replaced with other nodes, or deleted) of the programs under reduction. All transformations presented in Section 4 are based on the PNF. Any context-free grammar could be converted to PNF. Therefore, Perses is applicable to any programming language. This is why we can easily, effectively support reducing Java programs in addition to C as presented in Section 5.2, which requires little effort on integrating the Java grammar.

Generality to Other Test Inputs. Although we only focus on program reduction in this paper, Perses is capable of reducing other structured test inputs (*e.g.*, XML and HTML documents). Perses is expected to perform at least as well as HDD. We focus on program reduction since programs have much more complex constraints (in terms of both quantity and complexity) over their syntactical structures than either XML or HTML. These complex constraints significantly increase the difficulty of reduction and increase the search space of invalid variants $\mathbb{P}_{invalid}$ as well.

Extensibility. The transformations described in Section 4 are the first attempt to instantiate the Perses reduction framework. More transformations can be designed. Perses currently supports reducing production rules either by deletion (*cf.*, Section 4.3) or replacement (*cf.*, Section 4.4). We can define new transformations to increase $\mathbb{P}_{valid}$. for example, replacing long string literals with an empty string, converting an expression by appending '`;`' to an expression statement to replace an existing large statement. Before Perses, due to the omission of the information in grammar definitions, it is impossible to design such an extensible reduction tool in a general way . Perses makes this feasible.

7 RELATED WORK

The most related work to Perses is Hierarchical Delta Debugging [16] that exploits the tree structure of the test inputs. Hodován *et al.* proposed an approach to speed up HDD in [5]. Delta Debugging [24] is the seminal work on testcase reduction. However, neither of them address the syntactical validity for program reduction. As a result, they waste a considerable amount of time in exploring the invalid search space $\mathbb{P}_{invalid}$. Many delta-debugging-based frameworks leverage the underlying language feature to achieve almost optimal program reduction for a specific language. For instance, JS Delta [8] relies on the WALA static analysis infrastructure [6] to reduce JavaScript programs. C-Reduce [17] employs a set of heuristics for efficient program reduction based on the C/C++ semantics obtained from Clang. Our evaluation shows that in terms of the reduction throughput (*i.e.*, the number of tokens deleted per second), Perses is 1.67× faster than C-Reduce.

For reducing C programs, it is possible to combine both Perses and C-Reduce. Specifically, we could run Perses to quickly remove irrelevant program elements, and then use C-Reduce to obtain the almost optimal reduction result. Le *et al.* introduce a meta-reducer by combining Berkeley Delta [15] and C-Reduce in their EMI testing project [9–11, 20]. Perses can improve the performance of the meta-reducer by replacing Berkeley Delta. Recently, Herfert *et al.* propose a Generalized Tree Reduction (GTR) algorithm which combines a generic set of transformations for a particular language by learning from a corpus of example data [18]. Perses is a general framework and does not require any prior knowledge on the input data. Perses could also be used for reducing other structured testcases such as minimizing structured text formats for security testing [12, 19]. Binkley *et al.* propose an Observational Program Slicing (OPS) technique to reduce a program based on the results of property checks [3, 23]. To reduce a program, the OPS technique uses line deletion whereas Perses manipulates trees. Perses is more general and supports arbitrary program properties.

8 CONCLUSION

In this paper, we propose a novel, effective program reduction framework Perses. Same as HDD, it is general to any programming languages and performs reduction on the parse trees of programs. But differently, it is aware of the syntactical constraints between nodes, and thus guarantees no generation of syntactically invalid programs during reduction and enables more effective program transformations. It significantly outperforms DD, HDD and even C-Reduce—a specialized reducer for C/C++ programs—in terms of size of reduced programs and efficiency of reduction.

In order to further demonstrate the generality of Perses, we instantiate with an Antlr grammar for Java. The evaluation on six bugs in Java compilers shows that Perses runs much faster than HDD and produced much smaller results, which is consistent with the evaluation on C programs.

We believe that Perses's integration of syntactical knowledge opens a new way towards general, effective, and efficient program reduction. Consequently, in the future more program transformations can be easily designed and implemented within Perses, either general or specific to a certain programming language.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by the United States National Science Foundation (NSF) Grants 1319187, 1528133, and 1618158, and by a Google Faculty Research Award. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[2] ANTLR. 2017. The ANTLR Parser Generator. (2017). http://www.antlr.org/, accessed: 2017-08-05.

[3] David Binkley, Nicolas Gold, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: language-independent program slicing. In *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.

[4] GCC. 2017. A Guide to Testcase Reduction. (2017). https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction, accessed: 2017-08-05.

[5] Renáta Hodován, Akos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 194–203.

[6] IBM. 2017. The T.J. Watson Libraries for Analysis. (2017). http://wala.sourceforge.net/, accessed: 2017-08-05.

[7] JavaCC. 2017. The Java Parser Generator. (2017). https://javacc.org/, accessed: 2017-08-05.

[8] JS Delta. 2017. JS Delta. (2017). https://github.com/wala/jsdelta, accessed: 2017-08-05.

[9] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[10] Vu Le, Chengnian Sun, and Zhendong Su. 2014. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*.

[11] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 386–399.

[12] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*. 1193–1204.

[13] LLVM. 2017. How to submit an LLVM bug report. (2017). https://llvm.org/docs/HowToSubmitABug.html, accessed: 2017-08-05.

[14] LLVM/Clang. [n. d.]. Clang documentation – LibTooling. ([n. d.]). https://clang.llvm.org/docs/LibTooling.html, accessed: 2017-08-06.

[15] Scott McPeak, Daniel S. Wilkerson, and Simon Goldsmith. [n. d.]. Berkeley Delta. ([n. d.]). http://delta.tigris.org/, accessed: 2017-08-20.

[16] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[17] 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.

[18] Jibesh Patra Satia Herfert and Michael Pradel. 2017. Automatically Reducing Tree-Structured Test Inputs. In *ASE*. To appear.

[19] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*.

[20] 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*. 849–863.

[21] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. 294–305.

[22] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294.

[23] Shin Yoo, David Binkley, and Roger D. Eastman. 2014. Seeing Is Slicing: Observation Based Slicing of Picture Description Languages. In *Proceedings of the 2014 IEEE International Working Conference on Source Code Analysis and Manipulation*. 175–184.

[24] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[25] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 347–361.