

Enabling Language-Specific Transformations in Language-Agnostic Program Reduction

by

Gaosen Zhao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

© Gaosen Zhao 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

As the lead author of this thesis, I was responsible for contributing to conceptualizing study design, building artifacts, conducting experiment data collecting and analysis, and drafting and submitting manuscripts. Mengxiao Zhang, Zhenyang Xu and Kai Ma provided feedback on draft manuscripts. Kai Ma and Kangcheng Xu helped build artifacts and evaluation benchmarks.

Abstract

When a program P triggers a bug in a language implementation, program reduction can reduce P by removing program elements that are irrelevant to the bug, to facilitate debugging. Program reduction has been widely used in communities of various language implementations. Generally, program reduction techniques can be classified into language-agnostic program reducers (ARs) category and language-specific program reducers (SRs) category. ARs work generally well in a wide range of languages, but usually produce less optimal results than SRs due to a lack of domain knowledge of specific languages. However, SRs require extensive engineering effort to leverage the domain knowledge, and can only function in their target language but not in other languages.

To combine the benefits of both ARs and SRs and minimize the gap between the two, a novel, general transformation framework, **Metis**,¹ is introduced. Specifically, **Metis** allows users to specify language-specific program transformations to further minimize the results by SRs and the users only need to know the syntax of the target language and a concise domain-specific language named MTL (**Metis** Transformation Language) provided by **Metis**; **Metis** automatically processes the transformation rules inscribed in MTL by performing pattern matching and subsequent rewriting operations on the parse tree of the program under reduction. **Metis** provides a general, unified framework for specifying program transformations for different languages.

We comprehensively evaluated **Metis** on two benchmark sets of C and SMT-LIB programs and the results demonstrate that **Metis** yields much smaller programs than the state-of-the-art language-agnostic program reducer by 35.8% on average. We also compared **Metis** with two SRs: ddSMT and C-Reduce. **Metis** produces results of comparable size to ddSMT, but with a noticeable 28.9% shorter reduction time; while falling short of matching the reduced program size by C-Reduce, **Metis** saves 82.4% of queries and achieves a speed improvement of 30.6% less runtime.

¹The selection of the name **Metis** finds its roots in Greek mythology’s figure of Metis, the goddess symbolizing wisdom and adept transformation which mirrors the essence of our tool. The designation **Metis** represents the evolution of our prior work, Perses, itself named after another titan god. Metis is solely intended for symbolic purposes in this context.

Acknowledgements

I would like to express my deepest gratitude and appreciation to the individuals who have played significant roles in the completion of this work.

First and foremost, I am profoundly grateful to my supervisor, Chengnian Sun, for his mentorship and expertise. His insightful comments, constructive criticism and constant availability have immensely contributed to the development and refinement of my research.

I would also like to extend my appreciation to my coworkers at SWAG Lab for their unwavering support and encouragement.

Furthermore, I am honored to collaborate with Kai Ma and Kangcheng Xu who helped me draft the proposal and first prototype. This work would not be finished without them.

Last but not least, I would also like to express my gratitude to Professor Michael W. Godfrey and Professor Shane McIntosh for being my thesis readers and providing constructive feedback.

Table of Contents

Author’s Declaration	ii
Abstract	iv
Acknowledgements	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Motivating Examples	4
3 Background	8
3.1 Program Reduction	8
3.2 Program Transformation	9
4 Methodology	10
4.1 Structure Overview	10
4.2 MTL Language Design	11
4.3 Schedule and Verification of Transformations	15
4.4 Metis and Perses Integration	16

5	Evaluation	17
5.1	Evaluation Setup	17
5.2	RQ1: Can Metis outperform the state-of-the-art program reduction algorithms?	17
5.3	RQ2: Can Metis effectively reduce programs in different programming languages?	20
5.4	RQ3: What is the effectiveness of different reduction settings?	21
6	Related Work	26
7	Discussion	28
7.1	Limitations	28
7.2	Threats to Validity	29
8	Conclusion and Future Work	30
	References	31

List of Figures

2.1	An example that shows how Metis transformations help reduction in C.	5
2.2	An example that shows how a Metis transformation helps reduction in SMT-LIB.	6
4.1	The structure overview of Metis	10
4.2	The syntax of Metis	12
4.3	The tree matching example	13
4.4	check-sat-assuming Elimination Rule	14
5.1	Evaluation results of program reduction algorithms on C.	19
5.2	Evaluation results of program reduction algorithms on SMT-LIB.	24

List of Tables

5.1	A list of transformations implemented in Metis	18
5.2	The first column is the name of the bug. The second column ($O(\#)$) is the original size of input program in the number of tokens. Each reducer has three columns that are the remaining size of the program in the number of tokens ($R(\#)$), the number of queries ($Q(\#)$) and the runtime in seconds ($T(s)$). The last two rows are the mean and median of each column.	23
5.3	This table shows the results of the benchmarks under different settings, i.e., window size of 1, 3, 5, and 10, respectively.	25

Chapter 1

Introduction

Program reduction is the process of transforming a program into a smaller-sized version while preserving its essential behavior. Formally, given a program P and a property ψ that P exhibits, program reduction produces a minimized program P' that still exhibits ψ . Program reduction is typically demanded in communities of various language implementations, where P represents a program used to test the correctness of a compiler and ψ means that P triggers a bug in the compiler when P is being compiled. Program reduction facilitates the process of debugging language implementations. For instance, bug-triggering programs contain 30 or so lines of code in LLVM and GCC bug reports [12], whereas the original bug-triggering programs have thousands of lines of code [21, 6, 24, 13].

The existing program reduction tools can be classified into two categories based on their generality: language-agnostic reducers (ARs) and language-specific reducers (SRs). ARs are designed to be applicable to a wide range of programs regardless of the specific programming language. ARs such as DD [22], HDD [9] and Perses [13] are primarily based on deleting code segments, guided by dedicated algorithms. In contrast, SRs, such as C-Reduce [10] for C/C++ and ddSMT [5] for SMT-LIB2, are crafted for specific programming languages. These reducers not only call ARs to quickly minimize P , but also incorporate special reducers specifically designed for the target programming language to further minimize the AR-reduced results. For example, in C-Reduce there are multiple source-to-source program transformations implemented on top of the Clang LibTooling [7] such as function inlining, copy propagation and type alias replacement, to further minimize the reduction results by performing transformations that are not possible in ARs.

Technical Challenges. Although ARs possess lower engineering costs and universal applicability, they exhibit weaker performance than SRs due to the lack of language-specific

information. Drawing inspiration from SRs, it is believed that applying transformation rules after the outputs of ARs might result in more compact programs. Experiments on SRs with and without language-specific transformations enabled demonstrate the effectiveness of language-specific transformations. However, implementing transformations on existing tools for each distinct language is labor-intensive and time-consuming. The unique syntax, semantics and idiomatic patterns of each language require developers to have deep domain knowledge. Adapting tools to support multiple languages involves complex modifications to the tool’s architecture, algorithms and interfaces. Moreover, ongoing maintenance and updates to accommodate language evolution demand substantial resources. Consequently, there is an inherent trade-off between generality and performance for these two categories. A key question emerges: Is it possible to overcome the disadvantage of SRs, *i.e.*, high engineering cost?

Our Solution. In this work, a transformation framework called **Metis** is introduced, which is designed for implementing transformations with rewriting rules written in a domain-specific language (DSL). **Metis** is a language-independent static parse tree rewriting system focused on adaptability and usability. To streamline the design and application of transformation rules across various languages, we take advantage of the generality of *Perses* and implement **Metis** on top of it. In contrast to SRs that employ language analysis tools explicitly crafted for each language, **Metis** can process input in any programming language that has an ANTLR [1] grammar without any additional information. It incorporates analysis and editing infrastructure upon a unified intermediate representation for different program languages. Additionally, different sets of transformation rules can be dynamically designed and enabled for various scenarios with **Metis**.

We evaluate the effectiveness and efficiency of **Metis** and compare it with state-of-the-art ARs and SRs. The results reveal that **Metis** advances the capabilities of ARs by 35.8% in the C benchmarks and 54.2% in SMT-LIB benchmarks, approaching the performance of language-specific reducers. Although **Metis** cannot outperform C-Reduce in the size of output, it has better efficiency (tokens reduced per second). In reducing SMT-LIB programs, **Metis** demonstrates comparable performance with *ddSMT* while exhibiting shorter runtime. **Metis**’s inability to outperform C-Reduce is possibly attributed to the complexity of the C language in comparison to SMT-LIB and the extensive optimization work done on C-Reduce.

Contributions. The following contributions are made:

- We propose **Metis**, a parse tree pattern matching and rewriting framework for describing transformation rules, in order to perform language-specific transformations in language-agnostic program reduction.

- We have collected and implemented a set of language-specific transformations in MTL for SMT-LIB and C.
- Our evaluations demonstrate that **Metis** outperforms the state-of-the-art ARs in output size. When compared to SRs, **Metis** is not able to generate smaller programs in C but achieves similar results (79% and 80% in reduction rate) in SMT-LIB with 28.9% less time.

Chapter 2

Motivating Examples

Two examples in C and SMT-LIB are introduced respectively in this section to elaborate on how **Metis** helps to achieve better reduction results.

The compiler crash bug LLVM-25900 [2] spans 25,546 lines. Figure 2.1a is the output of Perses. Perses ensures that the program cannot be reduced by merely deleting a single token or performing its deletion-based reductions on the parse tree. However, this does not guarantee the optimal solution is reached. We indicate the bug-triggering portion of code is in line 17 which means that ideally all code except the bug-triggering portion should be reduced since they are not related to the bug. It is worth noting that there are multiple unnecessary nested functions. Such nested definitions and calls complicate the code’s flow, ultimately impacting readability. For example, `func_1()` is defined and called only once (highlighted in purple). Replacing the function call site at line 20 with its function body at line 7 will reduce the program size. `func_1()` invokes `func_14()`, which in turn calls `func_44()`. Such a chain of function calls can finally be simplified into one function call by applying the inlining transformation multiple times. This transformation has been implemented by C-Reduce as a language-specific modular reducer. However, the source code of this reducer comprises approximately 660 lines of code.

In contrast to the complicated implementation found in C-Reduce, **Metis** offers a more straightforward approach, as demonstrated in Figure 2.1d. This example showcases a function inlining transformation written in the MTL as a rule for **Metis**. The rule represents a pattern where a function definition with a function body but without input parameters can be eliminated by replacing its function body at the occurrences of its function call. On the left side of the \Rightarrow symbol, we have the pattern that **Metis** seeks to identify. The right side of the \Rightarrow symbol represents the pattern that will replace the one on the left. In this

<pre> 1 typedef signed int8_t; 2 ... 3 typedef unsigned uint32_t; 4 5 uint16_t p_19; 6 int func_44(uint32_t); 7 int64_t func_1(){ 8 func_14(); 9 } 10 11 int8_t func_14(){ 12 func_44(p_19); 13 } 14 int func_44(uint32_t p){ 15 func_3(v1); 16 // bug triggering code 17 ... 18 } 19 int main(){ 20 func_1(); 21 } </pre>	<pre> 1 short p_19; 2 int func_44(unsigned); 3 4 int func_44(unsigned p){ 5 func_3(v1); 6 // bug triggering code 7 ... 8 } 9 int main(){ 10 func_44(p_19); 11 } </pre>
--	--

(a) The result by Perses.

(b) The result by Metis.

```

typedef <specifier:typeSpecifier> <id:Identifier> ;
⇒

```

```

do
    locations = findInSubtree(id)
    substituteInTree(locations, specifier)

```

(c) Type Alias Elimination Rule

```

<declarationSpecifier> <name:Identifier> () { <block:kleene_star__blockItem> }
⇒

```

```

do
    locations = findPattern(" $name () ;")
    substituteInTree(locations, block)

```

(d) Function Inlining Rule

Figure 2.1: An example that shows how Metis transformations help reduction in C.

example, the right side is blank, indicating that the corresponding pattern will be deleted. The operations following the `do` keyword are an iterative rewriting script section which is executed at run time during the transformation process.

Another program transformation that can effectively reduce program size is the removal of `typedef`. In Figure 2.1a, the code highlighted in yellow represents a `typedef` definition and its usage. The `typedef` statement provides an alias for existing type, and removing the definitions while restoring their usages will not alter the program’s semantics. Figure 2.1b is the final result of **Metis** after doing multiple rounds of two transformations illustrated above, which is more concise than Figure 2.1a

Figure 2.1c illustrates the Type Alias Elimination Rule in **Metis**, The Left-hand side of \Rightarrow describes the pattern of `typedef` statement and the right-hand side is blank means this pattern is to be removed. Functions followed by `do` will restore alias defined by corresponding `typedef`. The reducer in C-Reduce relies on the abstract syntax tree (AST) visitor and rewriter, necessitating in-depth knowledge of Clang/LLVM toolchains and substantial engineering effort. Conversely, **Metis** automates pattern matching and transformation writing with minimal coding required. By applying these two transformations, the original 53 lines of code will be reduced to 29 lines while preserving the same semantic meaning, resulting in improved readability and reviewing experience.

```

1 (set-logic QF_AUFBV)
2 (assert
3 (let ((x 1))
4 (not (= x 1)))
5 (check-sat))
6 (exit)

```

(a) The result by Perses

```

1
2 (set-logic QF_AUFBV)
3 (assert
4 (not (= 1 1)))
5 (check-sat)
6 (exit)

```

(b) The result by **Metis**

```

( let ( ( <sym:symbol> <tem1:term> ) ) <tem2:term> )
 $\Rightarrow$ 
  tem2
do
  locations = findInSubtree(tem2, sym)
  substituteInTree(locations, tem1)

```

(c) Let Binding Inlining Rule

Figure 2.2: An example that shows how a **Metis** transformation helps reduction in SMT-LIB.

Another example demonstrates a typical situation encountered during the reduction

process of SMT-LIB. Figure 2.2 presents a simplified code snippet that includes a let binding. This program cannot be further reduced by deleting any token. However, a transformation that substitutes the binding variable back into the body (highlighted in yellow) proves to be beneficial. In this example, the variable `x` is replaced by `1` within the body, allowing the elimination of the let binding clause. Although ddSMT incorporates this transformation, its implementation is heavily dependent on its own framework. In contrast, **Metis** can achieve this transformation through a find and rewrite action, as depicted in Figure 2.2c. It is worth noting that Figure 2.2c bears a strong resemblance to Figure 2.1d, despite the fact that they represent rewriting rules for entirely different languages.

Removing **typedef** and inlining function do not always yield positive results. However, they are proved to be effective in the specific scenarios presented in the examples above. These examples highlight the potential benefits of program transformation in code reduction. During the process of program reduction, program transformations not only reduce the program size directly but also play a gateway role for further reduction by other reducers. Regarding generality, SRs vary substantially between languages and depend on existing language analysis tools. On the other hand, **Metis** supports multiple programming languages with a single design.

Chapter 3

Background

3.1 Program Reduction

The formal definition of program reduction has been mentioned in §1. Recall that a program P has a certain property ψ . Program reduction is a process of reducing the size of program P while preserving the property ψ . Program reduction can be classified into two categories, language-agnostic reducers (ARs) and language-specific reducers (SRs), based on whether they support different programming languages.

ARs. The first systematic study of program reduction can be traced back to 2002 when Delta Debugging (DD) was proposed with the motivation of automating the process of narrowing down the search space of oversized bug-inducing programs [22]. DD proposed a method that treats the program as a sequence of elements and systematically feeds a subsequence of the original input into the test oracle and gradually reduces the size by selecting the subsequence that is still able to pass the property tests.

DD is so general that it is applicable to all kinds of programs because its `ddmin` algorithm treats the program as a general structure, i.e. a sequence of tokens. However, it also makes it hard to effectively reduce computer programs. Hierarchical Delta Debugging (HDD) is then proposed to solve this challenge [9]. Nodes at each level of a tree structure are fed to `ddmin` to determine which tree branches can be deleted. In other words, HDD runs `ddmin` in a top-down manner to trim the tree gradually. Perses was proposed with better performance on reducing the program by leveraging the syntax. It performs reduction over the parse tree of the program with a deeper understanding of the programming

language. Thus, it ensures the preservation of syntax validation in its output.

SRs. The performance of ARs is limited in terms of effectiveness due to the absence of domain-specific knowledge of the target language. In response to this limitation, language-specific program reduction has been proposed. By incorporating delta debugging with language-specific reducers, these approaches can capture and eliminate unrelated code that may be overlooked by language-agnostic approaches. SRs achieve enhanced outcomes by executing transformations that leverage domain-specific knowledge, thus overcoming a local minimum. These reducers work on ASTs and can perform more sophisticated and “meaningful” operations. C-Reduce, the state-of-the-art program reducer for C and C++ code, is designed with custom-written reducers for program transformation, which are specialized transformations tailored to reduce C and C++ programs. These reducers are implemented using the Clang compiler infrastructure. However, the high performance achieved comes at the cost of generality. C-Reduce is not able to support other programming languages because it is built on top of the C and C++ environments. The state-of-the-art language-specific reducer for SMT-LIB, ddSMT, faces a similar situation as C-Reduce. ddSMT parses programs into its intermediate representation and cannot reduce programs written in languages other than SMT-LIB.

3.2 Program Transformation

In general, program transformation is a process that converts a program to a different one, and it has wide-ranging applications across fields such as compiler construction, program verification, and software refactoring. It is important to note that the specific requirements for program transformation can vary greatly depending on the application scenario, and thus there are many tools for different purposes.

AST-based transformations are commonly used in program analysis, optimization, and refactoring. They can be used to identify performance bottlenecks, detect bugs, and improve the structure and readability of the code. However, it is important to note that generalizing AST transformation is not an easy task. Since AST is tightly coupled with language specifications; a deep understanding of specifications is required to build the tree and design transformation correctly for a specific language.

On the other hand, a parse tree is a structure that requires fewer language details and can be built easily with context-free grammar. It represents the syntactic structure of the program in a hierarchical manner, which abstracts from language intricacies and makes it applicable to a broader range of programming languages.

Chapter 4

Methodology

This section gives an overview of the **Metis** transformation framework followed by an explanation of each component in detail.

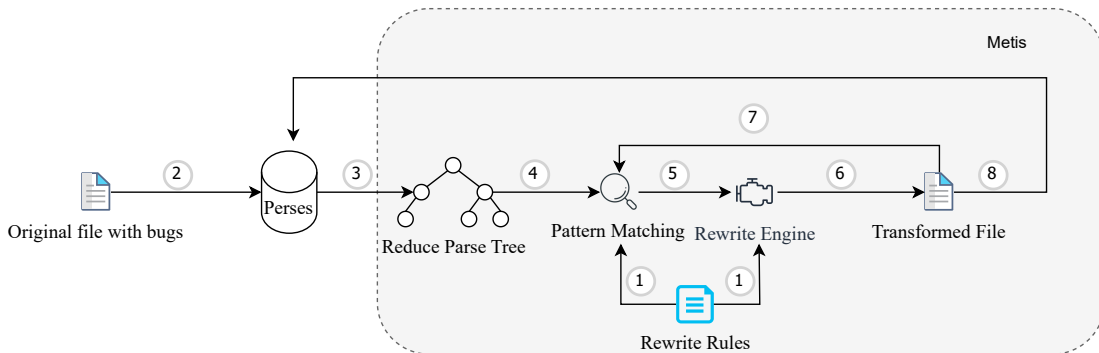


Figure 4.1: The structure overview of **Metis**.

4.1 Structure Overview

Figure 4.1 delineates the structure overview of **Metis**. In step ①, user-defined rewrite rules written in MTL are compiled by **Metis** during the bootstrapping phase. These rules consist of pattern descriptions and operations, which are compiled separately for future pattern matching and rewriting. The compilation process is depicted by arrows labelled with ①.

In step ②, the original program, which contains bugs and spans thousands of lines of code, is passed to Perses along with a property test for reduction. Perses converts the program into a parse tree and performs deletion and substitution operations on the tree. The process continues until a fixed point is reached, indicating that further reduction by Perses is not possible. At this point (step ③), **Metis** intervenes.

In step ④, the pattern-matching algorithm implemented in **Metis** compares the input program with target program patterns to identify occurrences of the desired pattern. Subsequently in step ⑤, **Metis** performs transformations by rewriting the corresponding pattern with user-defined operations and updates the program tree in step ⑥.

In step ⑦, the output generated by **Metis** is then returned to the pattern-matching stage for the next iteration. In each iteration, numerous potential applicable transformations are raised and verified, but only one transformation is applied to the parse tree. In step ⑧, The iterative process described above continues until either no further transformations can be applied or the execution time limit is reached. Once **Metis** reaches this fixed point, it relinquishes control back to Perses. Finally, the whole process terminates when both Perses and **Metis** reach fixed points.

4.2 MTL Language Design

Syntax. The syntax of MTL, as illustrated in Figure 4.2, defines the structure and composition of the language. Each rewrite rule in MTL follows the format of $p \Rightarrow q$ or $p \Rightarrow q$ **do operations**, where p and q represent syntax patterns, and **operations** refer to the rewriting script.

Syntax patterns consist of code fragments that adhere to the grammar of the target object language. New languages can be supported by providing an ANTLR grammar file. For example, `int x = a + b ;` is a syntax pattern that consists solely of concrete tokens. To improve expressiveness, syntax references can be incorporated. An instance of such a pattern is `int x = <expression> ;`, where **expression** serves as a syntax reference, while `x`, `=`, and `;` remain as concrete tokens. In this case, the former pattern represents itself, while the latter pattern represents a valid form of `int x = <expression> ;`, where the `<expression>` placeholder can be replaced by any string that conforms to the production rule of `<expression>`. Valid instances of this pattern may include `int x = c ;`, `int x = 1 ;` or `int x = functionA(a, b) ;`. Additionally, the references can be labelled with a name as demonstrated in Figure 4.2.

label: an identifier string
concreteToken: a token in the target language
tokenReference: a token reference in the target language grammar
ruleReference: parse rule in the target language grammar
operationStatement: predefined operation

$$\langle rule \rangle ::= \langle pattern \rangle ' \Rightarrow ' \langle pattern \rangle$$
$$| \langle pattern \rangle ' \Rightarrow ' \langle pattern \rangle ' do ' \langle operations \rangle$$
$$\langle pattern \rangle ::= \langle item \rangle ^*$$
$$\langle item \rangle ::= \text{concreteToken}$$
$$| \langle Reference \rangle$$
$$\langle Reference \rangle ::= \text{tokenReference}$$
$$| \text{ruleReference}$$
$$| \text{label ' : ' tokenReference}$$
$$| \text{label ' : ' ruleReference}$$
$$\langle operations \rangle ::= \langle operations \rangle \text{operationStatement}$$
$$| \epsilon$$

Figure 4.2: The syntax of **Metis**.

Patterns in MTL must conform to the syntax rules of the target language so that they can be parsed into a parse tree. The syntax of the rewriting script in MTL must align with the syntax of the hosting language, which is Kotlin in our implementation. The remaining components in **Metis** are delimiters that aid in the organization and structure of the code.

Pattern Matching. As mentioned in §3, **Metis** is a static parse tree parsing framework, so it operates on syntactic patterns and performs program transformations on a parse tree.

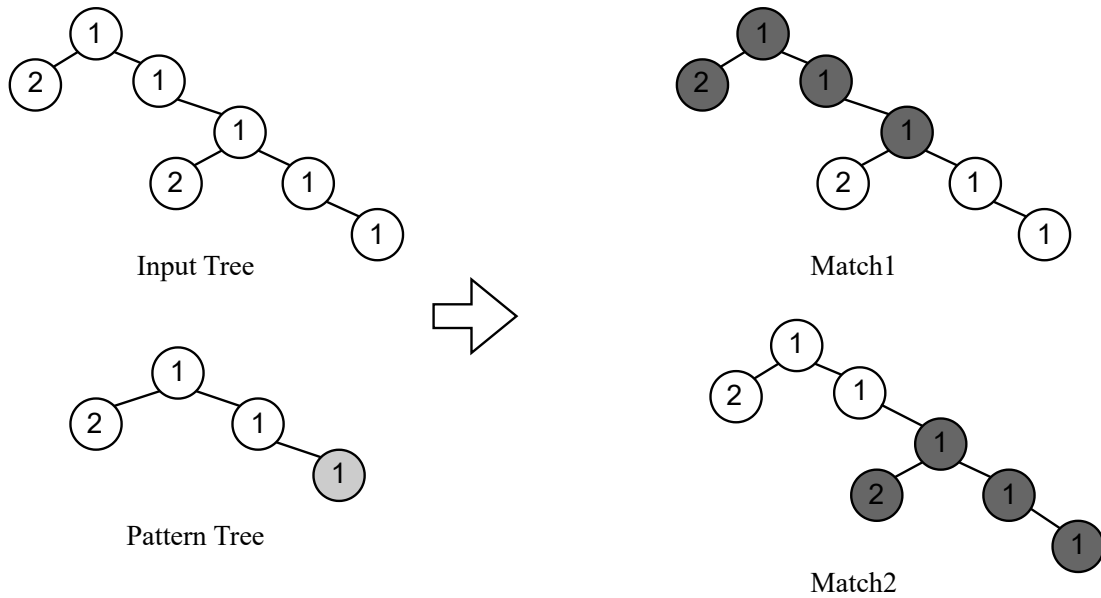


Figure 4.3: The tree matching example

Pattern matching can be converted to a tree-matching problem defined as follows: Given a pattern tree P and an input tree T , P matches at node α of T when there exists a one-to-one mapping between each node of P and subtree of T root at α . This mapping can involve either two internal nodes with the same generation rule or two leaf nodes with the same lexeme. To bind variables in the pattern script with nodes in the input tree, a special type of tree node called a “tag” node is designed for syntax rule name matching. These “tag” nodes contain types and can be loosely matched to input tree nodes with the same rule type.

In the example of Figure 4.3, two trees on the left of the arrow are passed as inputs. The pattern tree with a reference node (indicated in gray) can match two subtrees in the original tree, which are highlighted in black. Reference is bound to the corresponding respective

```

( check-sat-assuming ( <term> ) )
⇒
( check-sat )

```

Figure 4.4: check-sat-assuming Elimination Rule

position on the input tree. Two matching subtrees will eventually generate two different potential program transformations and the smaller one will be adopted. Fast tree matching algorithm by Dubiner *et al.* [3] can achieve the time complexity of $\mathcal{O}(n\sqrt{m}\text{polylog}(m))$ where m and n are the sizes of the pattern tree and target tree, respectively.

Rewriting. Rewriting is composed of two components in the transformation rule: Rewriting pattern template and rewriting script. The rewriting pattern template and rewriting script are connected to their corresponding matching pattern.

The underlying rationale is that once a pattern is identified, the new pattern (defined on the right-hand side of \Rightarrow) is constructed at run time and rewrites the original pattern at each matched occurrence in the program. This approach employs a declarative pattern template, offering greater expressiveness for simple program transformations. Similar to Parser Parser Combinators [16] which also adopts declarative pattern templates, it has been demonstrated to provide better usability compared to traditional language-specific tools. In the example presented in Figure 4.4, all commands conforming to the pattern (`check-sat-assuming (<term>)`) are substituted for (`check-sat`) at their respective locations.

In terms of designing more sophisticated transformations, rewriting scripts (those succeeding the “do” keyword in the syntax) are permitted to modify the input program more extensively. It allows the user to retrieve, iterate and modify the parse tree with an iterative script. This script has the same expressive power as a function on the hosting language. In the example exhibited in Figure 2.1, variables previously bound by pattern matching (`declarationSpecifier` and `name`) are employed in the rewriting operation to search for function calls.

Metis is language-independent as mentioned. To overcome the difficulty of the lack of languages-specific information like function tables and make designing transformations become more straightforward, we offer highly reusable pre-defined functions that encapsulate frequently used operations. They allow users to access and modify parse trees without needing in-depth knowledge of the underlying tree structure. For example, the `substituteInTree()` function replaces specified target locations with a provided tree root, accepting a list of locations and a tree root that represents program segments. It

is commonly used in conjunction with the `findInSubtree()` function, which efficiently identifies nodes within the subtree that have the same value as the target node. Together, these functions form a powerful tool for achieving complex transformations.

In Figure 2.1d, an example is showcased how a function definition is eliminated, and its function body is moved to the locations where the function is called. This transformation is achieved in three steps. First, it deletes the definition by leaving blank on the right-hand side of \Rightarrow . Second, the function call is located in the parse tree by calling `findPattern()` helper function, which is a direct call to the tree-matching algorithm. Substitution is performed by the `substituteInTree()` function in the last step.

4.3 Schedule and Verification of Transformations

Transformations should not directly modify the program itself, as not all of them lead to the optimal reduction result. For scheduling and verifying transformations, we introduced the following definitions. A program edit represents a single modification step applied to the current state of a specific program. An edit, denoted as ϵ , is a tuple consisting of three elements: p , l and α . p represents the program itself. l refers to a specific node in the parse tree. α indicates the action type which is either deletion or replacement. In the case of deletion, the node at location l and its children are removed from the parse tree. In the case of replacement, the node at location l and its children are substituted for another subtree. Therefore, a transformation is a sequence of concrete program edits. We have chosen to use the transformation as the minimal unit for verifying and applying changes. This ensures that the transformation process is fully guided by the user-defined rewriting rules, step by step. In the example of Figure 2.2c, A let binding transformation is applied. This transformation involves replacing the corresponding symbol in the let body and deleting the let statement in the outer layer.

Transformation verification has two steps. First, **Metis** ensures the consistency of edits within a transformation through dependency checking. For instance, a node under a deletion edit or a node to be replaced cannot be affected by other edits. Additionally, if the location of a node is changed, other edits containing this node need to be notified and updated accordingly. Second, a transformation is considered to be valid if the program can still pass property test after the transformation is applied.

The ordering of applying transformations has significant impact on the performance of the system, as highlighted by previous research (Whitfield *et al.*, 1997) [18]. When there are multiple potential transformations available simultaneously, applying one of them may

prevent the application of others, leading to the phase ordering problem. It is important to note that there is no universally optimal ordering of techniques that works well for all programs and architectures (Willsey *et al.*, 2021) [19]. The optimal order may vary based on factors such as the characteristics of the input program or the user-defined rewriting rule. To address this challenge, we employ a strategy where all program edits are generated simultaneously, and the transformation (sequence of program edits) that produces the smallest output is applied. While this strategy may miss the optimal path, it guarantees that **Metis** produces a smaller output step by step, preventing the generation of larger programs through recursive transformations (e.g., rewriting x as $x * 1$ and $x * 1 * 1 \dots$). However, the rewriting rules still need to be ordered carefully, as certain rules may block each other. By default, **Metis** performs extensive rewriting until no further patterns can be matched or until a preset maximum number of iterations (defaulting to one hundred) is reached.

4.4 **Metis** and Perses Integration

Metis and Perses are closely connected by design. The Perses paper acknowledges the potential of more general transformations for improved results in the future [13]. Building upon the Perses framework, we extend its functionality to create **Metis**, ensuring seamless integration with other Perses reducers. This integration saves runtime by eliminating repeatedly pretty-printing and parsing of programs between different representations. The programming languages supported by **Metis** are the same as Perses due to the integration and new languages can be easily adopted by adding the corresponding ANTLR grammar into **Metis**.

Compared to Perses reducers, **Metis** may have longer runtime but also offers smaller results. Therefore, **Metis** is activated only when all other reducers have reached their limits. In this way, **Metis** helps overcome local minimum points and returns the program to other reducers for further fast reductions.

In summary, **Metis** serves as a valuable addition to Perses reduction capabilities when language-agnostic reducers can no longer make progress. By integrating **Metis** within the reduction process, we can capitalize on its high effectiveness while minimizing the impact of its extended runtime. This approach ultimately leads to a more efficient and robust reduction process.

Chapter 5

Evaluation

5.1 Evaluation Setup

All experiments were conducted on an Ubuntu 20.04 server equipped with an Intel Xeon Gold 5217 CPU @ 3.00 GHz and 384 GB RAM. Each reducer operated on a single thread throughout all experiments. The performance evaluation focuses on two programming languages, C and SMT-LIB, as they have SRs capable of outperforming Perses.

Transformation rules for SMT-LIB were developed according to the “mutators” in ddSMT and rules for C-Reduce were from the “modular reducer” in C-Reduce. Given the distinctions between existing SRs and our generic platform, sixteen of the most effective and implementable transformations were selected and five transformations (listed in Table 5.1) are employed for the C program accordingly.

We evaluate the performance regarding the following research questions:

5.2 RQ1: Can **Metis** outperform the state-of-the-art program reduction algorithms?

To address this research question, we evaluate the effectiveness of **Metis** and Perses in reducing C programs by comparing their performance to the benchmarks from Perses. The Perses benchmarks comprise 22 real-world, reproducible bugs sourced from the public repositories of GCC and LLVM. We assess several aspects of the performance of Perses, **Metis**, and C-Reduce on this benchmark set, including:

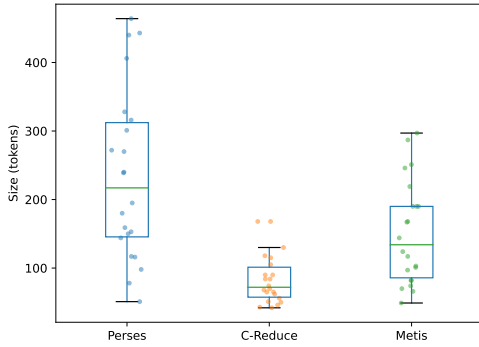
Table 5.1: A list of transformations implemented in **Metis**.

Language	Rule	Description
C	declaration-removal	Move a local declaration to the beginning of the program
	function-inlining (with-parameters)	Inline a function definition which has parameters
	function-inlining	Inline a function definition without a parameter
	void-return-function-definition	Change the return type of a function definition into void and remove the return statement
	typedef-replacement	Remove a <code>typedef</code> statement and restore the type it defines
SMT-LIB	let-substitution	Eliminate a let statement and inline its local variables
	annotation-removal	Remove an annotation
	equality-substitution	Propagate equalled variables
	plus-merge-with-children	Remove brackets in a <code>plus</code> expression i.e. $a + (b + c) = a + b + c$
	and-merge-with-children	Remove brackets in an <code>and</code> expression i.e. $a \text{ and } (b \text{ and } c) = a \text{ and } b \text{ and } c$
	forall-removal	Remove a <code>forall</code> statement
	exists-removal	Remove an <code>exist</code> statement
	bv-conversion	Replace a bit-vector constant with a simpler constant
	eval-false	Replace an equality starting with <code>false</code> with a negation <code>not</code>
	double-not-elimination	Eliminate a double not expression
	double-neg-elimination	Eliminate a double negation expression
	check-sat-assuming-replacement	Replace a <code>check-sat-assuming</code> function with a regular <code>check-sat</code>
	merge-functions	Merge functions having the same type into one
	convert-fun-to-const	Convert a function into a constant
	concat-to-zero-extend	Replace a <code>zero-extend</code> bit-vector with <code>concat</code> function
substitute-with-const	Substitute a node with a constant	

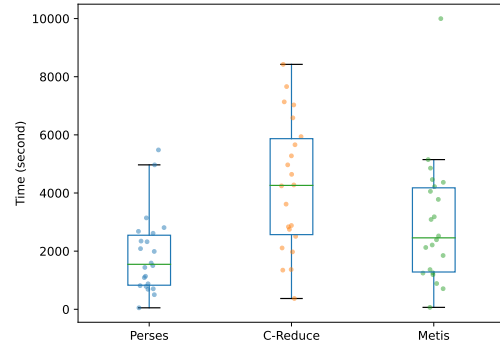
1. **Size:** The total number of lexical tokens in the result programs.
2. **Queries:** The total number of property tests (ψ §1 in the definition of program reduction) being called.
3. **Time:** The total runtime of the reducers.

Table 5.2 presents the output size, time and query count for each test case in the benchmarks. **Metis** has smaller outputs on every test case compared to **Perses** but is larger than outputs of **C-Reduce**. The difference is more significant for larger test cases including `gcc-61047`, `clang-23309` and `gcc-66375`. To explore more details from experiment results,

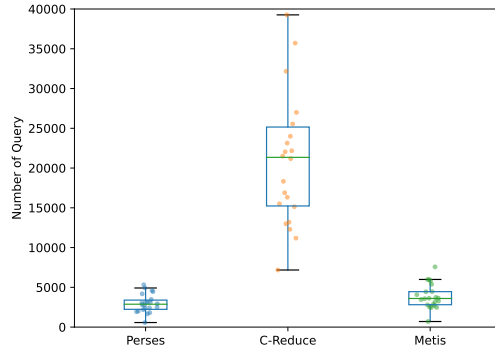
we create box plots on the size of the reduction program, execution time and execution query count.



(a) Reduced program size.



(b) Execution time.



(c) Execution query count.

Figure 5.1: Evaluation results of program reduction algorithms on C.

Figure 5.1a presents the distribution of the final size from each reducer. Generally, **Metis** produces more compact programs than Perses, yet it does not outperform C-Reduce in terms of size. On average, **Metis** generates programs with 150 tokens, which are 35.8% smaller than those of Perses (234 tokens) and 79.7% larger than those of C-Reduce (83 tokens). This is anticipated because **Metis** is built upon Perses and offers additional reduction opportunities through DSL-based transformations. Although DSL-based transformations are more lightweight to implement, they do not cover as many transformations as C-Reduce does. Apart from the program transformation, C-Reduce provides multi-level optimizations

with some well-designed heuristic algorithms that might help produce smaller reduction results.

Figure 5.1b and Figure 5.1c present the time and query counts for the three reducers. We plot time and query counts together since they are positively correlated. More queries require more reduction time generally. On average, **Metis** takes 3,850 queries, which requires 2,954 seconds.

Compared to Perses (2,970 queries and 1,892 seconds), **Metis** takes 29.6% more queries and 56.1% more time, exploring additional reduction opportunities at the cost of increased time and query count. In comparison with C-Reduce (21,883 queries and 4,255 seconds), **Metis** saves 82.4% in query count and 30.6% in time, offering a more efficient reduction process.

RQ1: Compared to Perses, **Metis** reduces 35.8% more tokens at the cost of 56.1% more time. Even though still not able to compete with C-Reduce in program size, **Metis** saves 82.4% of queries and is 30.6% faster.

5.3 RQ2: Can **Metis** effectively reduce programs in different programming languages?

To confirm the generality of **Metis** for programming languages beyond C, we apply **Metis** to SMT-LIB, a language designed for SMT solvers. We selected this language due to the availability of a reducer called ddSMT. In our evaluation, we compare the performance of Perses, **Metis**, and ddSMT on the SMT-LIB language on benchmarks from ddSMT [5]. The timeout is 3600s for each test case. Additionally, we apply **Metis** to the output of ddSMT to assess whether **Metis** can enable further reduction beyond the results obtained from SRs.

We collect 194 valid programs that Perses can parse from the benchmark set of ddSMT, and Figure 5.2a displays the program sizes generated by each algorithm. On average, the results from **Metis** (142.6 tokens) are 54.2% smaller than those from Perses (311.7 tokens). Although the average results from **Metis** are not superior to those from ddSMT (116.6 tokens), both reducers exhibit similar performance in producing the smallest output. ddSMT generates 102 smallest outputs, while **Metis** generates 112 smallest outputs (including ties). Moreover, by applying **Metis** to the programs generated by ddSMT, the size can be further

reduced by 9.6 tokens, resulting in an average of 107 tokens. The outcome of the significance test indicates that the p -value for the comparison between the output sizes of **Metis** and **Perses** is remarkably low at approximately $3.8e^{-30}$. This suggests a highly significant difference between **Metis** and **Perses**. Conversely, the p -value obtained for the comparison between **Metis** and **ddSMT** stands at 0.076. This value implies that the observed distinction between **Metis** and **ddSMT** is not statistically significant.

In terms of runtime, **Metis** (236 seconds) brings 29.7% overhead on **Perses** (182 seconds) but 28.9% faster than **ddSMT** (332 seconds). When applying **Metis** to the results obtained from **ddSMT**, the average runtime is 342 seconds, which only adds a minor overhead of 3%.

Figure 5.2c is the visualization of the runtime and the reduction size of **Metis** and **ddSMT**. The distributions of two reducers are highly coincided.

RQ2: **Metis** effectively reduces programs in SMT-LIB, generating significantly smaller programs than **Perses** with only a minor runtime trade-off. **Metis** has similar output size as **ddSMT** with 28.9% less runtime. Moreover, **Metis** can further reduce the outputs of **ddSMT** by 9.6 tokens, with a minimal overhead of just 3%.

5.4 RQ3: What is the effectiveness of different reduction settings?

To evaluate the performance of **Metis** under various conditions, we conducted experiments with the C benchmarks in RQ1 using different window sizes, specifically 1, 3, 5 and 10. The window size represents the search space within which **Metis** explores multiple potential transformations. This experiment runs on the results of **Perses** for simplicity because **Metis** always runs after **Perses**. In previous experiments, we set the window size at 30 for the smallest reduction size.

A window size of 1 indicates that **Metis** applies only the first transformation it encounters. Therefore, a window size of 3 implies that **Metis** considers up to three transformations simultaneously, ultimately selecting and applying the smallest one, as detailed in §4.

The experimental results reveal that the average reduction size for a window size of 1 is 171.8. The average reduction size for window sizes 3 is 153.6. Additionally, the average reduction sizes for window sizes 5 and 10 are quite similar, both being approximately 152. This is because a window size 5 is adequate for our experiments. It is noteworthy that the

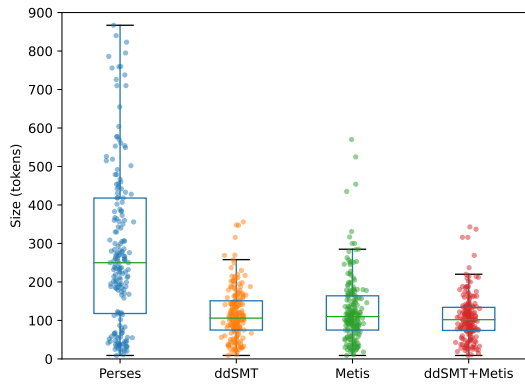
average reduction size for window sizes 5 and 10 is 35.5% smaller than that of a window size of 1.

Regarding the runtime, it is observed that an increase in window size leads to an additional runtime overhead which is not outstanding in current experiments. Precisely, the average runtime is measured at 227 seconds for a window size of 1, 237 seconds for a window size of 3, 235 for a window size of 5 and 243 for a window size of 10. The 17% overhead is introduced when comparing window sizes of 1 and 10.

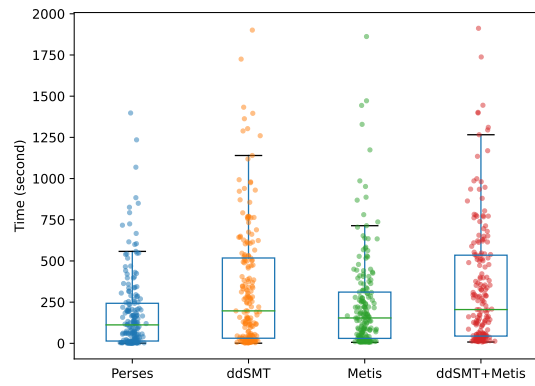
RQ3: Our scheduling and verification strategy reduces program size by 35.5% with 17% runtime overhead compared to the naive method.

Table 5.2: The first column is the name of the bug. The second column ($O(\#)$) is the original size of input program in the number of tokens. Each reducer has three columns that are the remaining size of the program in the number of tokens ($R(\#)$), the number of queries ($Q(\#)$) and the runtime in seconds ($T(s)$). The last two rows are the mean and median of each column.

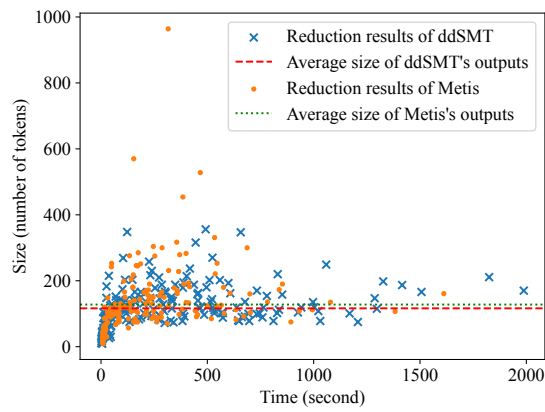
Bug	O(#)	Perses			C-Reduce			Metis			C(%) <i>w.r.t.</i>	
		R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	Perses	C-Reduce
clang-22382	21068	144	2380	502	70	13186	1367	116	2756	547	-19.44%	65.71%
clang-22704	184444	78	1812	1136	42	11189	2109	68	1975	1169	-12.82%	61.90%
clang-23309	38647	464	4604	1503	118	32167	4245	195	7206	2085	-57.97%	65.25%
clang-23353	30196	98	2813	688	74	12299	1347	86	3064	730	-12.24%	16.22%
clang-25900	78960	239	2501	812	90	16330	1975	160	3325	955	-33.05%	77.78%
clang-26350	123811	195	3092	2810	90	22175	5660	108	3709	3056	-44.62%	20.00%
clang-26760	209577	116	2183	1440	43	12992	2882	85	2517	1546	-26.72%	97.67%
clang-27137	174538	180	4916	5482	50	25544	8425	74	5409	5764	-58.89%	48.00%
clang-27747	173840	117	1661	877	68	15138	2838	100	2161	1033	-14.53%	47.06%
clang-31259	48799	406	2419	1591	168	26997	5278	249	4565	3488	-38.67%	48.21%
gcc-59903	57581	316	4181	3144	105	48652	7135	296	5052	3388	-6.33%	181.90%
gcc-60116	75224	443	5322	2086	168	39262	5939	282	7948	3053	-36.34%	67.86%
gcc-61047	17179	270	1935	788	62	16892	2749	103	2920	1300	-61.85%	66.13%
gcc-61383	32449	272	3478	2681	84	22049	4278	220	4441	4012	-19.12%	161.90%
gcc-61917	85359	150	2939	1090	65	23138	3617	119	3439	1181	-20.67%	83.08%
gcc-64990	148931	240	3132	1990	65	23996	4642	216	3731	2129	-10.00%	232.31%
gcc-65383	43942	153	2004	705	51	15510	2508	73	2416	811	-52.29%	43.14%
gcc-66186	47481	328	3022	2324	115	18329	4968	188	3979	3369	-42.68%	63.48%
gcc-66375	65488	440	3139	2346	56	21181	7661	186	5503	4319	-57.73%	232.14%
gcc-70127	154816	301	2772	2615	84	21507	6585	182	3702	3807	-39.53%	116.67%
gcc-70586	212259	159	4456	4968	130	35710	7030	141	4847	5858	-11.32%	8.46%
gcc-71626	6133	51	572	51	46	7179	371	47	672	65	-7.84%	2.17%
median	70356	217	2876	1547	72	21344	4262	130	3706	2107	-29.89%	65.48%
mean	92306	235	2970	1892	84	21883	4255	150	3879	2439	-31.12%	82.14%



(a) Sizes of the reduced programs.



(b) Execution time.



(c) Execution time and size of reduced programs of **Metis** and ddSMT

Figure 5.2: Evaluation results of program reduction algorithms on SMT-LIB.

Table 5.3: This table shows the results of the benchmarks under different settings, i.e., window size of 1, 3, 5, and 10, respectively.

Bug	O(#)	window size is 1			window size is 3			window size is 5			window size is 10		
		R(#)	Q(#)	T(#)	R(#)	Q(#)	T(#)	R(#)	Q(#)	T(#)	R(#)	Q(#)	T(#)
clang-22382	144	144	206	24	116	374	45	116	374	45	116	374	47
clang-22704	78	78	88	19	68	163	32	70	163	33	68	163	33
clang-23309	464	213	2634	555	197	2568	557	195	2578	563	195	2600	593
clang-23353	98	86	249	41	86	249	42	86	249	41	86	249	43
clang-25900	239	216	987	141	160	822	143	160	824	144	160	824	147
clang-26350	195	144	685	227	108	598	237	108	606	240	108	617	247
clang-26760	116	85	330	104	85	336	107	85	344	106	85	334	109
clang-27137	180	79	421	285	74	404	284	74	206	279	74	403	285
clang-27747	117	109	335	107	109	342	111	100	498	154	100	500	164
clang-31259	406	269	2222	1890	249	2146	1896	249	2146	1876	249	2146	1900
gcc-59903	316	296	856	235	296	861	237	296	866	235	296	868	243
gcc-60116	443	306	2710	962	282	2615	959	282	2622	961	282	2628	979
gcc-61383	272	220	962	1328	220	963	1329	220	964	1329	220	963	1334
gcc-61917	150	119	498	91	119	498	92	119	498	90	119	498	95
gcc-64990	240	240	309	67	240	311	68	216	596	135	216	599	137
gcc-65383	153	97	467	121	73	411	106	73	411	105	73	412	108
gcc-66186	328	194	926	1033	188	947	1037	188	951	1036	188	955	1042
gcc-66375	440	206	2433	1970	186	2364	1978	186	2364	1969	186	2362	1982
gcc-70127	301	301	404	411	182	909	1187	182	916	1189	182	927	1195
gcc-70586	159	159	208	449	141	390	902	141	390	889	141	390	898
gcc-71626	51	47	98	15	47	100	20	47	100	13	47	100	19
median	195	159	467	227	141	498	237	141	596	235	141	599	243
mean	233	172	858	480	154	875	541	152	889	544	152	901	552

Chapter 6

Related Work

Program Reducers. Delta Debugging [22] is a significant research methodology that has opened new horizons in program reduction. DD demonstrates high generality. However, due to the limited knowledge of test inputs, DD can experience performance issues, mainly when working with structural data. Hierarchical delta debugging (HDD) [9] is a significant development in the field of program reduction. HDD introduces an additional preprocessing layer to Delta Debugging, allowing it to operate over the parse tree of a program layer by layer. This approach significantly improves performance compared to traditional Delta Debugging techniques that operate over text token sequences. Following the development of HDD, two different types of reducers emerged: ARs and SRs. ARs retain the generality of HDD and can be used across multiple programming languages. Perses [13] is an example of a more effective and efficient AR, which generates only syntax-valid programs. Several endeavors have been undertaken to enhance the efficiency, effectiveness and usability of Perses. Vulcan [20] employs general, powerful and language-agnostic transformations to further improve the effectiveness of Perses. RCC [15] proposes a domain-specific caching scheme and further improves the memory efficiency of Perses. Perses^{ad hoc} [14] provides an easier way to adopt new languages for Perses, which greatly reduces the engineering cost. Moreover, PPR [23] is a tool that helps program reduction from another perspective, such that uses the Pairwise Program Reduction method to highlight the critical bug-triggering differences between a pair of bug-triggering and non-bug-triggering programs.

SRs leverage domain knowledge and tools to perform more detailed program analysis and transformations. C-Reduce [10] is a state-of-the-art SR designed for C and C++ programs. It uses Clang for parsing and a hybrid solution of KCC and Frama-C for test validation. C-Reduce includes thirty language-specific transformations, such as removing

unused functions or variables, inlining small functions and copy propagation. With the help of these transformations, SRs generally outperform ARs.

Program Transformation Tools. The advent of program transformation frameworks came to the fore in the late 20th century and early 2000s, featuring a range of proposed styles. Frameworks, such as OBJ [4] and its successors, are based on order-sorted equational logic of term rewriting and are powerful tools with broad applications. Stratego/XT [17] enters the landscape as a distinctive language and toolset dedicated to program transformation. Much like **Metis**, the Stratego language provides rewriting rules for articulating basic transformations. Additionally, it offers highly programmable strategies for scheduling the application of these rules. However, this increased power also comes with a higher learning cost. As discussed in §3, transformations on parse trees strike a balance between generality and expressiveness in the realm of program reduction. This approach allows for easier support of new languages. And frameworks above contain more language details which can escalate the learning curve and reduce their overall generality. Both OBJ and Stratego/XT which constraints a brand-new language or a complicated framework structure, demand substantial foundational knowledge to use and potentially pose barriers to some users.

The Dyck-Extended Language with its Parser Combinators [16] offers a lightweight syntactic rewriting tool. This program transformation framework operates on syntax trees and can handle grammar merged from multiple languages, enhancing the generality of rewriting rules. Therefore, it becomes an ideal choice for systematically modifying code-bases. However, it has limited capabilities in performing intricate transformations, such as those mentioned in the motivating examples, primarily due to the constraints of a pure declarative template. To overcome this challenge, **Metis** uses a combination of declarative code templates for pattern matching and iterative functions to assist the rewriting process. This dual approach strikes a balance between generality and expressiveness in the domain of program reduction.

More recently, program transformation by examples has been used to refine code systematically. REFAZER [11] is a tool that learns transformations from examples with machine learning techniques and suggests code edits for student programming assignments. It can synthesize program transformation rules from examples. LASE (Learning from Examples for Syntax-Level Code Refactoring) [8] is a code refactoring tool that creates edit scripts from multiple sets of examples and applies the transformation to similar code. While these tools provide straightforward representations of program transformation, they only support one specific language and have limited expressiveness. It would be hard to infer complicated transformations from a few examples. Therefore, **Metis** uses MTL to avoid these limitations.

Chapter 7

Discussion

7.1 Limitations

Although **Metis** is a powerful and versatile tool, it has some inherent limitations. Operating on the parse tree, **Metis** encounters difficulties in describing transformations when the logical structure of the program is missing. For instance, nodes in AST represent logical structures like expressions, functions or variables, while the parse tree only contains syntactical details for parsing. Some nodes in the parse tree do not have the corresponding meaning to the programming language itself. Therefore, a subset of transformations employed in SRs cannot be readily applied to **Metis**. For instance, C-Reduce incorporates transformations such as combining variables, which involves merging declarations of multiple variables within the same context to reduce program size. It also uses the aggregate-to-scalar transformation, which replaces accesses to a member of an aggregate with a new scalar variable. Additionally, C-Reduce performs dead code elimination by removing unreachable code. This limitation stems from such transformations that often require context or semantic information that the parse tree does not provide.

Another limitation of **Metis** is that it treats rules in different languages individually. As a result, users are required to manually adopt the same rule for different programming languages, increasing in engineering costs. This drawback arises because Perses uses different grammars and parsers for various languages instead of employing a combined parser, which could streamline the process.

7.2 Threats to Validity

Construct Validity. The evaluation of **Metis**'s performance is centered around a comparison of average reduction time, average reduction size and the smallest outputs. While these metrics provide valuable insights, it is worth considering the potential for additional statistical analyses to further enrich the assessment.

Internal Validity. The selection of benchmarks for these experiments draws from real-world bug reports, a context where **Metis** would realistically be employed. This congruence between the benchmarks source and the intended usage scenario contributes to bolstering the internal validity of the results.

External Validity. While **Metis** boasts support for multiple programming languages, this evaluation primarily focuses on its application within C and SMT-LIB contexts. The usability and performance of **Metis** with languages outside this scope have not been directly assessed. However, it is worth noting that the versatility of **Metis**'s design, which operates based on general structures and transformation rules, implies that its effectiveness is not confined to specific languages. The current set of experiments is believed to adequately reflect this versatility.

Chapter 8

Conclusion and Future Work

In this study, **Metis**, a transformative program reduction framework, is introduced. Engineered for convenience and versatility, this framework serves as a comprehensive solution for designing language-specific reducers for different programming languages to achieve smaller reduction results. **Metis** features a DSL interface tailored for creating rewriting rules in declarative patterns and iterative scripts with helper functions.

To evaluate the efficiency and effectiveness of **Metis**, eight transformation rules for C and fourteen transformation rules for SMT-LIB are designed and implemented. The evaluation result demonstrates that **Metis** can achieve 35.8% less average output size in C benchmarks and 54.2% less average output size in SMT-LIB benchmarks than language-agnostic reducer without notable overhead. Compared to the language-specific reducer, **Metis** has better time efficiency. **Metis** does not outperform C-Reduce but has similar results as ddSMT in the average size of output.

The future work can be broadly divided into two aspects. The first aspect involves further improving **Metis**. At present, **Metis** supports C and SMT-LIB only. We will collect more transformations in other programming languages and do corresponding experiments. The second aspect of future work is enhancing the usability of **Metis**. This includes automatically deriving transformations in DSL from pairs of before and after examples provided by the user. Although existing methods have been used in other areas, it is still difficult to infer complicated transformations from a few examples.

References

- [1] ANTLR. The antlr parser generator, 2017.
- [2] LLVM Bugzilla. Bug 25900, 2015. https://bugs.llvm.org/show_bug.cgi?id=25900, accessed: 2023-07-21.
- [3] Moshe Dubiner, Zvi Galil, and Edith Magen. Faster tree pattern matching. *J. ACM*, 41(2):205–213, mar 1994.
- [4] Joseph Goguen, Timothy Winkler, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing obj. *Software engineering with OBJ: algebraic specification in action*, 11 1999.
- [5] Gereon Kremer, Aina Niemetz, and Mathias Preiner. ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2021.
- [6] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.
- [7] LLVM/Clang. Clang documentation – libtooling, 2022. <https://clang.llvm.org/docs/LibTooling.html>, accessed: 2022-10-28.
- [8] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511, 2013.
- [9] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Con-*

- ference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006, pages 142–151. ACM, 2006.
- [10] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery.
 - [11] Reudismam Rolim, Gustavo Soares, Loris D’Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjoern Hartmann. Learning syntactic program transformations from examples, 2016.
 - [12] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 294–305. ACM, 2016.
 - [13] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 361–371, New York, NY, USA, 2018. Association for Computing Machinery.
 - [14] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. Ad hoc syntax-guided program reduction. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE Tool)*, 2023.
 - [15] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. On the caching schemes to speed up program reduction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12 2023.
 - [16] Rijnard van Tonder and Claire Le Goues. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 363–378, New York, NY, USA, 2019. Association for Computing Machinery.
 - [17] Eelco Visser. *Program Transformation with Stratego/XT*, pages 216–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

- [18] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, nov 1997.
- [19] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.
- [20] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. Pushing the limit of 1-minimality of language-agnostic program reduction. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.
- [21] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.
- [22] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
- [23] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. Ppr: Pairwise program reduction. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2023.
- [24] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361. ACM, 2017.