Improving Source-Code Representations to Enhance Search-based Software Repair

Pemma Reiter*
Arizona State University
Tempe, AZ, USA
pdreiter@asu.edu

Antonio M. Espinoza*
Arizona State University
Tempe, AZ, USA
amespi22@asu.edu

Adam Doupé
Arizona State University
Tempe, AZ, USA
doupe@asu.edu

Ruoyu Wang
Arizona State University
Tempe, AZ, USA
fishw@asu.edu

Westley Weimer
University of Michigan
Ann Arbor, MI, USA
weimerw@umich.edu

Stephanie Forrest
Arizona State University
Tempe, AZ, USA
stephanie.forrest@asu.edu

ABSTRACT

Automatically improving and repairing software using search-based methods is an active research topic. Many current systems use existing source code as the ingredients of repairs, either through evolutionary computation derived random mutation or other heuristic operators. However, these code transformation operators are not always well-matched to the granularity of the source code on which they operate. This paper proposes a static source-to-source preprocessing step to produce code with more uniform granularity that exposes relevant program components to the repair process. This approach, called Program Repair Enhancement via Preprocessing (PREP), has been applied to three different repair tools, each of which uses different code transformation operators and search algorithms. In every case, applying PREP before the search allows the tool to repair software defects that were previously unattainable by that tool. PREP finds 88 unique previously-unreported correct repairs across these tools. This result is significant because it is applicable to most search-based software improvement methods, and it addresses the fundamental issue of how to match the granularity of the representation to the granularity of operators.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

automated program repair, transformation, rewriting, search-based software engineering

*Both authors contributed equally to this research.

GECCO '22, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9237-2/22/07…$15.00
https://doi.org/10.1145/3512290.3528864

ACM Reference Format:
Pemma Reiter, Antonio M. Espinoza, Adam Doupé, Ruoyu Wang, Westley Weimer, and Stephanie Forrest. 2022. Improving Source-Code Representations to Enhance Search-based Software Repair. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528864

1 INTRODUCTION

Automated program repair (APR) is an active sub-field of software engineering, with dedicated tracks in top conferences exploring a variety of approaches, ranging from search-based methods [7, 16] to constraint solving [14, 15] and machine learning [10]. Despite the intensity and diversity of these efforts, today's APR tools typically generate semantically-correct repairs at a rate below 50% [26]. Further, most methods do not attempt to synthesize program code de novo, and, instead, define code transformation operators, such as mutation, which manipulate existing code, following what is known as the statement-level redundancy assumption [13] or the plastic surgery hypothesis [2]: A substantial fraction of human-generated software repairs are composed entirely of tokens that exist in the program, providing a rationale for code transformations that manipulate existing code fragments.

If the ingredients of APR search methods are adequate, why do the implemented tools fail to find more repairs? There are many possible explanations, such as search budgets, details of operators, and so forth. The obstacle we explore in this paper is code representation—APR operator incongruity. *Code representation* refers to an APR tool's internal representation of input source code, consisting of *elements*. *APR operators*, henceforth *operators*, access elements of a code representation and perform actions such as copy, delete, or replace. *Incongruity* occurs when the representation elements are poorly matched with the operators.

We identify two types of incongruity:

(1) **Statement Incongruity**: Occurs when relevant representation elements for creating patches are not accessible to operators, although they are present in the program.
(2) **Type Incongruity**: Occurs when representation elements of similar, compatible types are not used by operators.

In this paper, we focus on changing the code representation, leaving both APR tools and operators intact. Our insight is that simple syntactic changes to code representation can make more elements accessible to operators and reduce incongruity.

Instead of directly modifying the code representation, we employ *source-to-source code transformation* in a pre-processing step that restructures the source code to expose more elements to the operators. This allows us to be compatible with existing source-level repair algorithms, including those that apply a patch representation [16, 28, 29] (a list of edits) to source code, and those that mutate a code representation [7] (e.g., an abstract syntax tree). To address statement incongruity, we extract nested elements from complex statements. To address type incongruity, we statically cast similarly-typed elements (expressions and variables) to existing elements (variables). These transformations preserve original program semantics.

The implementation of our method is called *Program Repair Enhancement via Preprocessing* (PREP). Since PREP operates on the source code, it complements existing APR algorithms, as we show in our evaluation. We apply PREP to three tools which represent different underlying algorithm classes: f1x, GenProg, and Prophet. We evaluate PREP on multiple APR tools over two sets of buggy programs: the Codeflaws dataset and the DARPA Cyber Grand Challenge [5] dataset (CGC).

To summarize, this paper makes the following contributions:

- A novel method for improving the congruence between operators and representations in source-level APR. The method uses general and type-aware source-to-source static code transformations to address statement and type incongruities (Section 3).
- An implementation of the method as *PREP* (Program Repair Enhancing via Preprocessing) and an evaluation of PREP on two datasets (Codeflaws and the DARPA Cyber Grand Challenge) across three independently-developed APR tools that target the C programming language (f1x, Prophet, and GenProg). For Codeflaws, we find that PREP, allows these tools to increase correct repairs found by 2.6%, 7.6%, and 8.8% respectively. For CGC, we find that PREP improves vulnerability mitigation results by 26.6%, 100%, and 13.6% respectively (Sections 4 and 5).
- A quantitative approach for assessing how congruent an APR tool's operators are to its code representation. Considering three quite different APR tools, we find that f1x is the most congruent and GenProg the least (Section 5).

These results are significant because they can be applied to most search-based software improvement methods, and they address the fundamental issue of how to match program representation to popular operators used in both evolutionary computation-based and other approaches for APR. To further open and reproducible science, our prototypes, the curated benchmark dataset, and all of our experimental results are available at https://github.com/amespi22/code_rewrite.

2 MOTIVATING EXAMPLE

To motivate the utility of source code transformations, consider the code snippet from the DARPA Cyber Grand Challenge program Palindrome, shown in Listing 1. This code contains a bug on line 8, where the hardcoded value of 128 in the third parameter is too large and admits an overrun of the allocated 64 bytes.

```
1  int cgc_check(){
2      int len = -1;
3      int i;
4      int pal = 1;
5      char string[64];
6      for (i = 0; i < sizeof(string); i++)
7          string[i] = '\0';
8      if (cgc_receive_delim(0, string, 128, '\n') != 0)
9          return -1;
10 ...
11 }
```

Listing 1: CGC Challenge : Palindrome with stack-based buffer overflow.

```
1      int  tlv3;
2      char * tlv4;
3      int tlv5;
4      char tlv6;
5      tlv3   = 0;
6      *tlv4  = string;
7      tlv5   = 128;
8      tlv6   = '\n';
9      tlv1 = cgc_receive_delim(tlv3, tlv4, tlv5, tlv6);
10     if (tlv1 != 0) {
11         return (-1);
12     }
```

Listing 2: The result of simple source-to-source, static code transformations.

The human-generated patch replaces this hardcoded value with `sizeof(string)`.

Ideally, an automated tool would mimic the human-generated patch. However, this requires that (1) the element `sizeof(string)` be available in the source code or repair templates, (2) the tool can manipulate function parameters, and (3) the tool can access the repair ingredient.

In our example, condition (1) is satisfied by line 6 in the Palindrome source. The conditions (2) and (3) are not so easily met, however, as they depend on the operators. Although some search-based methods define operators that manipulate function parameters and other operands directly, when implemented naively this can greatly expand the size of the search space, and many current research prototypes do not operate at that level of granularity. They may also lack access to the required repair element, e.g., because of scope or typing issues. From our example, `sizeof(string)` could be unavailable for a number of reasons: `sizeof(string)` is nested within a larger, atomic expression (i.e., the conditional check of a loop), the type of `sizeof(string)` is not available, or the type for `sizeof` (i.e., `size_t`) is not considered compatible with type `int`. PREP seeks to remedy both disparities—those between mutation operators and code constructs (statement incongruity) and those between the required `int` type and `size_t` (type incongruity).

If one could expose these elements to the operators via a source-to-source code transformation, they would become available and potentially help many of today's APR methods. For example, line 8 could be transformed from Listing 1 to the code shown in Listing 2, and patch ingredients of similar and valid types could be generated from the standard C-style elements of Listing 3. Simple trans-

```
1  void fix_ingred_service_1_0_2(){
2      char string [ 64 ];
3      bzero(&string,( 64*sizeof(char) ));
4      int len;
5      bzero(&len,sizeof(int));
6      {int len; len = (int)(sizeof ( string )); }
7      ...
8      {int tlv5; tlv5 = (int)(sizeof ( string )); }
9      ...
10 }
```

Listing 3: Static type-casting fix ingredients examples for Palindrome.

formations such as these preserve the semantics of the program while simultaneously allowing the contents of the function call to be manipulated by many existing APR operators.

To illustrate this point, we ran GenProg [7], an early EC-based APR tool, on the Palindrome bug, and it failed to find a repair using the original source code of Listing 1. However, when we reran GenProg with the semantically-equivalent transformed code from Listings 2 and 3, it found a correct repair. The repair was found by a copying a mutation that used line 7 of Listing 2 as the destination and the declaration in line 8 of Listing 3 as the source expression.

This sort of simple mutation operator is available in many current research tools, including those that do not use EC. Our proposed transformation allows the third argument of `cgc_receive_delim` to be replaced without changing any such underlying mutation operators. Our evaluation of PREP includes `Palindrome` (Section 5.2). Of the three APR algorithms we studied, GenProg and Prophet both correctly repair (i.e., find a patch that is equivalent to the human-supplied patch) Palindrome by using the semantically-equivalent code generated by our code transformations.

3 CODE TRANSFORMATIONS

We propose to use source-to-source code transformations to reduce statement and type incongruity between representations and operators for automated program improvement. In this section we describe our transformations (summarized by examples in Table 1). Our transformations address both statement (T1–T3) and type (T4) incongruence. Due to compound statements, T2–T3 transformations are applied recursively and stop when no further transforms are applicable.

Decouple assignments from declarations (T1). This transformation decomposes complex assignments and declarations, allowing operators to access subsets of compound assignments as elements. Our code transformations are tool agnostic, and we do not assume that they separate assignments from declarations. We illustrate this in Transform 1.

```
[typ] [var] = [expr] ;
---------------------------------------
[typ] [var] ;
[var] = [expr] ;
```

Transform 1: Decoupling assignments from declarations. ("Before" code is shown above the dashed line, "after" code is shown below.)

Decouple function calls from conditional statements (T2). When function calls are embedded within conditional statements, it may be difficult for APR tools to change their parameters. By extracting the function call and replacing it with an equivalently-typed temporary variable equal to the return value (show this in Transform 2), PREP enables manipulation of conditional statements containing function calls. The APR tool can then mutate the function call independently (e.g., replacing it with a type-equivalent variable). Additionally, this transformation allows an APR tool the ability to replace function call parameters with any equivalently typed variables when applied in conjunction with transformations T3 and T4. This increases the repair search space while preserving program semantics.

```
if ( [func] ([args]) )
---------------------------------------------------------------------
[typ] [tmp_var] = [func] ([args]);    s.t. typ = return_type(func)
if ( [tmp_var] )
```

Transform 2: Decoupling function calls from conditional statements.

Decouple content from function call parameters (T3). Human-written repairs often replace a function call parameter with another value located elsewhere in the code (e.g., Section 2). Due to statement incongruity, such repairs may not be expressible by APR tools. This transformation extracts all function call parameters, declares temporary variables of the same type and assigns values equal to the original parameters, and replaces the original parameters with their respective temporary variables (shown in Transform 3). This enables APR tools that perform mutation operations such as swap or append to modify any function call parameter.

```
[func] ([args])
-----------------------------------------------------------------------
[typ_i] [tmp_i] = [val_i];    ∀ (typ_i,val_i) ∈ {val_0,...,val_N}=[args]
|                                s.t. typ_i = parameter_type(func,i)
[func]([tmp_0],...,[tmp_N])
```

Transform 3: Decoupling content from function call parameters.

Type compatibility and static casting (T4). Many APR tools aggressively screen potential patch element components. For example, patch elements are commonly required to match in name and type. Aggressive screening benefits standard APR tools by eliminating early compilation failures in strongly-typed languages. However, many screening approaches may conservatively rule out useful repairs because they use strict notions of type equivalence that do not account for type congruity or safe casting (cf. physical typing [3]). This transformation casts statements and expressions of equivalent types to a target type to generate usable type-compatible elements. We illustrate this in Transform 4. This allows APR tools to access a richer set of patch ingredients while remaining type safe.

To preserve semantic equivalence across all transformations, as well as the original source's intended scope and functionality, we

```
{ [typ_b] [var_b]; [var_b] = ([typ_b]) [e_a]; }
|                          ∀ S ∈ Scopes
|                          ∀ (typ_b, var_b) ∈ Variable_declarations(S),
|                          ∀ (typ_a, e_a) ∈ Expressions(S)
|                            s.t. typ_b = variable_type(var_b),
|                                 typ_a = resolved_type(e_a),
|                            and  equivalently_typed(typ_a, typ_b) is True
```

Transform 4: Creating type-compatible elements through static casting.

Type	Transformations Example	
T1	`type name = value` ->	`type name` `name = value`
T2	`if(func(args))` ->	`type tmp_var = func(args)` `if (tmp_var)`

T3	`func(arg1, arg2)` ->	`type tmp1 = arg1`
		`type tmp2 = arg2`
		`func(tmp1,tmp2)`

T4	`{type tmp1; tmp1 = (type) expr2; }`	

Table 1: Transformation types and examples.

transform conditional and loop statements with a single statement as their body into multi-line scope statements by introducing blocks.

4 EXPERIMENTAL DESIGN AND SETUP

We selected three APR tools that operate on C code, each of which uses different methods for creating patches to buggy code. GenProg is an EC-based method with mutation operators that rely on the statement-level redundancy assumption. Prophet uses code templates that are discovered with machine learning. Finally, f1x relies on constraint solving. By testing each of these tools on the same datasets we can examine the degree to which each is limited by the incongruity problem and the degree to which our transformations can improve off-the-shelf methods.

4.1 Datasets

We evaluate our static code transformations on two datasets: Codeflaws [21] and the DARPA Cyber Grand Challenge Dataset (CGC) [5]. Codeflaws was intentionally created for program repair tools and obtained from the Codeforces online database [4]. The Codeflaws dataset provides a buggy source file, the corresponding human-repaired source file, a build mechanism, and test content (input, output, and evaluation scripts). It provides heldout test content, which supports testing patches for overfitting. We consider any patch that passes all heldout tests to be "correct." Codeflaws directly supports multiple APR tools.

The CGC content was created by security-focused software contractors as challenge problems for autonomous Cyber Reasoning Systems for the 2016 DARPA Cyber Grand Challenge (CGC). It contains a collection of 32-bit binary programs, their corresponding source code, at least one negative, i.e., proof-of-vulnerability (POV) test, and a mechanism for generating new test cases.

Using a Linux variant [22] of the CGC benchmark, we have modified the CGC dataset to run with program repair tools, including the program source code, a build mechanism, and test content. Each program may have multiple POVs associated with it; we use the term *scenario* to refer to each program–POV pair. The CGC dataset has a less strict repair requirement than Codeflaws: a program's vulnerability is considered mitigated if the program no longer raises an exception when the POV is executed.

For the CGC dataset evaluation, we first screened for a subset of scenarios that could easily support all three APR tools. Specifically, C programs (baseline and developer-supplied patch) must (1) compile with both gcc and clang, (2) exhibit expected behavior on negative and positive tests, and (3) correspond only to a single program executable used to evaluate all test content (100 C programs). Of these 100 valid C-programs [18], we selected a subset (27) from those which a repair algorithm operating on the entire program source failed to identify a repair within its 8hr budget. These 27 programs correspond to 55 scenarios for the PREP evaluation, covering most valid programs' vulnerability types.

4.2 Tool Configurations

Codeflaws. The Codeflaws dataset includes the tool configuration files required to run both GenProg and Prophet but does not include the required files for f1x. To keep the configuration of f1x as simple as possible and admit fair comparison to the other two tools, we used the same positive and negative tests that are used for GenProg and Prophet, a test timeout of 1 second and f1x's option to create all patches. Unlike GenProg, which was run locally on a Ubuntu 18.04 machine, we used the default f1x Docker container to ensure it ran with its intended system configuration. Similar to f1x, we used a Docker container for Prophet in our Codeflaws evaluation.

CGC. For the CGC dataset we used standard configurations for GenProg, Prophet, and f1x. We used the developer-identified set of buggy source files as the input source code. All test content was generated from the same testing sources, but we tailored the test harnesses individually to meet the invocation requirements of each APR tool.

Our CGC GenProg evaluation runs with both the single-edit mutation search option and with the genetic algorithm (GA) search [7, 24] option using GenProg's standard population size of 40 for ten generations. Our runs also expand GenProg's default locally scoped strict type matching with the `--semantic-check name` parameter. This parameter enables GenProg to use same-named, exactly-typed externally-scoped variables and their respective statements as ingredients for mutation.

Our CGC Prophet evaluation uses standard profiling as the localizer method [10]. We extended Prophet's implementation for multilib build environments, specifically adding support for 32-bit libraries. This was necessary to allow Prophet to evaluate the 32-bit CGC dataset.

Our CGC f1x evaluation uses its default configuration, identical to the Codeflaws f1x evaluation. Similar to Prophet, we extended f1x for the CGC dataset by adding support for 32-bit libraries. Additionally, f1x's testing requirements did not directly support the CGC's Python test harness. Equivalent f1x oracles were generated

	Baseline		PREP	
	Plausible	**Correct**	**ΔPlausible**	**ΔCorrect**
f1x	2190	1379	46	36
Prophet	2475	840	75	64
GenProg	1185	1170	101	99
Total Unique	2761	2046	27	88

Table 2: New repairs enabled by PREP. Codeflaws results for f1x, Prophet and GenProg. 'Δ' indicates the increase discovered after applying PREP.

from the positive test content as well as the DARPA POVXML files from which the POV executables were generated. However, DARPA did not provide POVXML files for seven of the 55 CGC scenarios (shown as 'NEG' in Table 3) prohibiting us from testing them with f1x.

Implementation. PREP is implemented in Python using the antlr4 [1] parser generator to parse C. Our implementation is available from https://github.com/amespi22/code_rewrite. We evaluated the CGC dataset using an f1x docker image built using Ubuntu 16.04 (Binutils 2.26), a Prophet docker image built using Ubuntu 18.04 (Binutils 2.30); while GenProg evaluations were conducted on Ubuntu 18.04 (Binutils 2.30) development machines.

5 RESULTS

We first report experimental results for Codeflaws and CGC. Using these results, we then consider their implications for the different tools in terms of the congruity between their representation and operators.

5.1 Codeflaws

PREP's transformations improved the number of plausible and correct patches for all three tools.

f1x: Because f1x uses symbolic execution to generate patches, and focuses on conditional statements (including loops), we did not expect our transformations to have a large effect on its performance. As Table 2 shows, however, f1x found new patches with PREP that it was unable to find in the baseline. PREP and f1x together discovered 36 correct patches that the baseline f1x did not, 16 of which are unique across all tools tested (Figure 1).

Prophet: When Prophet was run with PREP, it discovered 64 new correct patches above its baseline, of which 23 are unique (Figure 1).

GenProg: When GenProg was run with PREP, it discovered 99 new correct patches that the baseline GenProg missed, 45 of which are unique. Only two of the 101 candidate repairs failed to pass the heldout tests, i.e., almost all of the discovered patches were correct.

Collectively PREP, when paired with these three off-the-shelf repair techniques, discovered 88 new correct patches and 27 unique plausible patches, as shown in Table 2. By 'unique', we mean that the correct patch was found by only one PREP-enabled tool and was not found by any of the baselines. Many of the plausible patches found by the baseline runs and by the PREP-enabled tools were not correct according to the dataset's definition. For example, there were 2,761 unique plausible patches in the baseline, of which 2,046 were found to be correct, leaving 715 plausible yet incorrect (overfit) patches. Using PREP, each tool was able to correctly repair some of these overfit patches.

Of the new patches discovered with PREP, most were unique to each tool and there was very little overlap, as shown in Figure 1. This indicates that PREP is not just helping one tool discover a patch that other tools can find, but in several cases, it is helping them to discover unique patches. Overall, PREP improved each tool's correct repair success: GenProg had a relative increase of 8.8%, f1x of 2.6%, and Prophet of 7.6%. This improvement is particularly relevant because techniques such as GenProg are often associated with lower-quality repairs that overfit to visible tests (e.g., [17]): techniques that can reduce overfitting and instead produce correct patches are highly desired.

5.2 Cyber Grand Challenge

In the CGC evaluation, we first applied PREP and then provided the result as input to f1x, GenProg, and Prophet. There were twenty-two repairs of interest, i.e., CGC Scenarios (from Table 3). For four of these repairs, applying PREP alone mitigated the vulnerability (e.g., introducing well-typed temporary variables may cause a compiler to emit code with a different stack layout, potentially disrupting certain buffer overruns). We focus on the remaining repairs in which the vulnerability was mitigated by a repair tool operating on PREP-transformed code. For 15 of the top 22 rows of Table 3, Prophet was unable to find a repair on the original source code, but discovered a repair with PREP. In row 1 (cotton_swab_arithmetic.1), GenProg was unable to find a repair when provided only the original source code, but found a repair with PREP. For two scenarios (Music_Store_Client.2 and Music_Store_Client.3), GenProg did not find a vulnerability mitigation within the 8hr time period, but found them with PREP. We see that PREP was not as effective at improving f1x results, aiding in the discovery of only one patch (HackMan.1).

Overall, the PREP-enabled tools found mitigations for eighteen new scenarios, which they failed to find when using the original source code: six new uniquely-mitigated scenarios were found with PREP. For the CGC results, we manually reviewed the APR-generated repairs. This review found that two vulnerability mitigations were equivalent to the developer supplied repair—these correct repairs were found by both GenProg and Prophet for `Palindrome` using PREP-enabled source only. While GenProg took advantage of all four transformations in its correct repair, Prophet utilized T1–T3.

With PREP, we observed an increased number of timeouts in the given search budget (DNF for Did Not Finish), particularly for large programs such as `CGC_Planet_Markup_Language_Parser`. This is perhaps not so surprising, because our transformations increase the representation granularity, and therefore, increase the search space for operators that rely on existing code ingredients. There are cases in which the baseline tool found a repair with the original source code but failed with the PREP-transformed code. We analyzed these repairs carefully, particularly for Prophet, and found that the baseline repairs were overfit in each case. With the original source, the tool found patches that (1) changed the control flow of the source code with terminating statements, (e.g., `exit`),

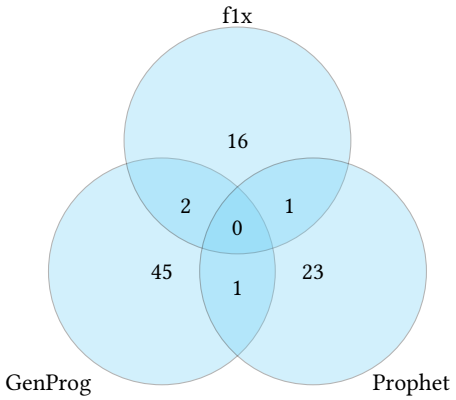

Figure 1: PREP enables each tested tool to find unique correct repairs in the Codeflaws dataset. Each number is a count of how many correct (not simply plausible) repairs the tool discovered with PREP that it failed to discover without PREP. Patches that were also found in the non-PREP baseline are not counted.

or (2) removed the initialization of variables or reset them (e.g., `malloc`).

Determining why these particular patches were overfit without PREP and correct with PREP is nuanced. We note, however, that the code transformations can change stack behavior, which influence outcomes for the tools. The transformations targeting fault-locations, T1–T3, introduce new local variables, which can increase stack usage and influence the nature of existing stack-based vulnerabilities. Our fix ingredient transformation, T4, initializes variables using `bzero`, effectively zeroing out portions of the stack. Taken together, these factors could reduce the chance of a tool mistakenly removing a variable initialization or `malloc`, increasing the chance of avoiding an incorrect repair, particularly with variables used for boundary checks.

Overall, PREP affected the relative plausible repair rate for GenProg by 13.6%, f1x by 26.6%, and Prophet by 100.0% (includes PREP-only mitigations). We note that even though PREP increases the search space while the search budget/configuration remained constant, the overall change was positive for each tool. Although it appears that Prophet outperformed GenProg and f1x, note that the total vulnerabilities mitigated by GenProg was 80% in the baseline whereas prophet and f1x were both 27.3%. In comparison to Codeflaws and its correctness criteria, passing visible CGC tests requires less effort.

These results suggest that our proposed code transformations enable existing tools to expand their reach by providing a representation that is more congruent with their code transformation operators.

5.3 Representation and Operator Congruence

PREP addresses two sources of incongruity between representations and operators: statement incongruity and type incongruity. For statement, we focused on a problem that plagues many software repair tools—function parameters, which can occur both outside

Scenario.POV	PREP only	f1x base	f1x PREP	GenProg base	GenProg PREP	Prophet base	Prophet PREP
cotton_swab_arithmetic.1		✓	✓	✗	✓	✗	✗
Diary_Parser.1	✓*	✗	✓*	ERR	✓*	✗	✓*
Diary_Parser.3		ERR	ERR	ERR	ERR	✗	✓
Diary_Parser.4		ERR	ERR	ERR	✓	✗	DNF
FablesReport.4		DNF	DNF	✓	✓	✗	✓
FISHYXML.1		✗	✗	✓	DNF	✗	✓
FSK_BBS.1		DNF	DNF	ERR	ERR	✗	✓
Griswold.4		✓	✓	✓	✓	✗	✓
HackMan.1		✗	✓	✓	✓	✗	✓
HIGHCOO.1		✗	✗	✓	✓	✗	✓
Music_Store_Client.2		ERR	ERR	DNF	✓	✗	✗
Music_Store_Client.3		✗	✗	DNF	✓	✗	✗
online_job_application2.1		NEG	NEG	✓	✓	✗	✓
On_Sale.2	✓*	NEG	✓*	ERR	✓*	DNF	✓*
Palindrome.1		✗	✗	✓	✓	✗	✓
Palindrome2.1	✓*	NEG	✓*	✓	✓*	✗	✓*
SCUBA_Dive_Logging.1		ERR	ERR	✓	✓	✗	✓
SCUBA_Dive_Logging.2		NEG	NEG	✓	DNF	ERR	✓
simplenote.1		✗	✗	✓	DNF	✓	✗
simplenote.2		✓	✓	✓	✓	✓	✗
stack_vm.1		✗	✗	ERR	DNF	✗	✓
The_Longest_Road.1	✓*	✗	✓*	✓	✓*	✗	✓*
CGC_Planet_Ma..._Parser.1		DNF	DNF	✓	DNF	✗	DNF
CGC_Planet_Ma..._Parser.2		DNF	DNF	✓	✓	DNF	DNF
CGC_Planet_Ma..._Parser.3		DNF	DNF	✓	✓	✗	✗
CGC_Planet_Ma..._Parser.4		DNF	DNF	✓	✓	DNF	ERR
CGC_Planet_Ma..._Parser.5		DNF	DNF	✓	✓	✗	DNF
CGC_Planet_Ma..._Parser.6		DNF	DNF	✓	✓	✗	DNF
CGC_Planet_Ma..._Parser.7		DNF	DNF	✓	✓	✗	DNF
FablesReport.1		DNF	DNF	✓	✓	✗	✗
FablesReport.2		DNF	DNF	✓	✓	✗	✗
FablesReport.3		DNF	DNF	✓	✓	✗	✗
FablesReport.5		DNF	DNF	✓	✓	✗	✗
FISHYXML.2		✗	✗	✓	DNF	✗	DNF
Griswold.1		✓	✓	✓	✓	✗	✗
Griswold.2		✓	✓	✓	DNF	✗	✗
Griswold.3		✓	✓	✓	✓	✗	✗
HackMan.2		NEG	NEG	✓	✓	✓	DNF
KTY_Pretty_Printer.1		✓	✓	✓	✓	✓	✓
KTY_Pretty_Printer.2		✓	✓	✓	DNF	✓	DNF
KTY_Pretty_Printer.3		✓	✓	✓	DNF	✓	✓
KTY_Pretty_Printer.4		✗	✗	✓	DNF	✓	✓
KTY_Pretty_Printer.5		✗	ERR	✓	✓	✓	✓
Minimalis..._Manager_3M.1		✓	✓	✓	✓	✓	✓
Minimalis..._Manager_3M.2		✓	✓	✓	✓	✓	✓
Movie_Rental_Service.1	✓*	✓	✓*	ERR	✓*	✓	✓*
Music_Store_Client.1		✓	✓	✓	✓	✗	✗
On_Sale.1		NEG	NEG	ERR	ERR	DNF	DNF
QuadtreeConways.1		✓	ERR	✓	✓	✓	✓
Rejistar.1		NEG	NEG	✓	✓	ERR	DNF
SOLFEDGE.1		✗	✗	✓	✓	✓	✓
SOLFEDGE.2		✗	✗	✓	✓	✓	✓
SPIFFS.1		DNF	DNF	✓	✓	✓	✓
Street_map_service.1		✓	✓	✓	DNF	ERR	✗
WordCompletion.1		✗	✗	✓	DNF	✗	✗

Table 3: CGC Results. Baseline versus PREP results for f1x robust configuration, GenProg evaluation for all single-edit mutations and GA search, and Prophet with profile-based localization. '✓' indicates that a patch was found, '✗' that no patch was found, 'DNF' that the tool did not finish in the 8hr. search budget, 'ERR' indicates tool errors, 'NEG' when no f1x-compatible negative test was available, '*' identifies examples where PREP mitigate the negative test (POV) on its own with no repair tool.

and inside conditional statements (`if` and `case`). We can use the number of new patches that a tool discovers with PREP as a proxy for the congruence of its operators with the program elements that are accessible from its representation. Using the Codeflaws results (Table 2), we can rank these tools from most to least congruent: f1x (36), Prophet (64), and GenProg (99). This finding aligns with, and provides an additional dimension of support for, recent discussions of APR overfitting (e.g., [17, 27]) and repair search spaces (e.g., [11, 19]).

6 RELATED WORK

Because PREP is the first tool that we know of to perform static source-to-source transformations to aid APR tools, related work falls under two categories, static code transformations and APR tools—many of which are evolutionary computation (EC) or genetic algorithm (GA) based [6, 7, 16, 25, 28, 29].

6.1 Static transformations

The work most similar to ours is Comby [23], a tool that performs static code transformations for general use, e.g., refactoring, repair, or rewriting. Comby performs static code transformations for multiple target languages by focusing on context free language properties utilizing templates and a parser combinator. We were unable to leverage Comby in this context as the transformations we make require type information (a known limitation of Comby). Comby has the ability to extract function arguments for use in a code rewrite, however there is no mechanism to retrieve the argument's type, which is required to make a proper variable declaration in our rewriting methods. In addition, Comby has no mechanism with which to make our T4 transformation.

6.2 APR tools

There are three main approaches APR tools take: search-based (e.g., [7, 20]), templated (e.g., [8]), and semantic approaches such as symbolic execution (e.g., [14, 15]). Hybrid models such as Prophet and ARJA-e [29] exist and combine two or more approaches to reap the benefits of each. Although our code transformation method is generic, the templated approach is similar in principle to ours. These templated approaches typically identify static code patterns that are historically related to bugs and apply templated fixes to the buggy code via transformations. In addition, tools such as ARJA [28] and the work of Oliveira et al. [16] tackle the incongruity problem by addressing patch representation.

Oliveira et al. [16] discuss increasing fault granularity by creating a new patch representation (as compared to GenProg). This method does not increase the granularity of the possible patch atoms (the smallest element that can be used in a single edit patch) but allows the GA to more easily mix atoms when searching for a patch. While this method embeds a finer-grained approach, its specialized patch representation and crossover operators only function within the APR tool application, and results in uncompilable program variants.

Similar to Oliveira et al., ARJA [28] focuses on a lower-granularity patch representation. Additionally, ARJA reduces the search space through the application of rules which screen out edit operations deemed "meaningless" as well as increasing the likelihood of successful compilation through fix ingredient screening. In this screening, ARJA also applies a "type-matching" method that translates variables or methods from a patch statement to a compatible in-scope variable or method.

ARJA-e [29] is a hybrid approach that leverages both the same premise as ARJA, the statement-level redundancy assumption (the plastic surgery hypothesis [2]) and repair templates adapted from PAR [8], and integrates edits of different granularities into their patch representation.

PAR [8] is a patch-based approach, which defines ten fix templates from fix patterns identified from human-written patches. PAR applies a fix template to a fault location and evaluates whether or not the context is appropriate, e.g., a "null pointer checker template" ensures that the fault location contains an object reference and rewrites the AST with a null check of that object reference. Different levels of granularity of the program's AST are indirectly supported through some fix templates, like "Parameter Replacer" and "Expression Replacer", which replace variables or expressions in statements or method parameters with type compatible elements from the same scope. Another state-of-the-art template-based repair tool, TBar [9] identifies additional fix patterns, totaling 15, yet faces similar scope limitations with respect to replacement variables and expressions. While such template-based APR tools do have granularity-aware fix patterns, these are applied to the internal representation of the input program by the tools themselves.

Our strategy is different from ARJA, ARJA-e, PAR, and TBar, in that our granularity-increasing transformations are independent from an APR tool. Additionally, our transformation strategy identifies type-compatible fix ingredients agnostic of scope, then generates assignments between same-named declared variables and these ingredients, type-casting between compatible standard C types (e.g., considering `size_t`, `uint16_t`, and `unsigned short int` as compatible in a notion more aligned with physical typing [3]).

7 DISCUSSION

We discuss the broader implications of our results and its limitations.

Of preprocessing on congruence. Our evaluation shows that simple source code transformations can improve the ability of multiple tools — using a variety of repair methods — to find repairs for software defects. Including PREP, the total number of correct repairs found by GenProg improved by 2.5%, Prophet by 1.6%, and f1x by 0.9%. This improvement is similar to incremental, tool-specific advances reported for APR algorithms, e.g., SPR to Prophet (+0.5% correct) [12] and ARJA [28] to ARJA-e [29] (+9.4%), but PREP's improvements generalize across multiple tools.

Of scope. We chose to add type-compatible fix ingredients (T4) in a distinct external scope to ensure that the transformations are semantically equivalent. This implementation choice limits collisions with existing code elements, avoiding the need to conduct additional dataflow analysis before inserting new code. Although this choice did not impact GenProg, which has flexibility regarding ingredients and scope, the other tools limit fix ingredients by scope. For example, Prophet limits `Value Replacement` expressions to

the same basic block. This means that for Prophet to take advantage of T4 ingredients, the type-compatible fix ingredients would need to be introduced into the appropriate scope.

On completeness of transformations. We initially selected PREP's four transformations by studying the motivating example, using GenProg which failed to find a valid solution from all single-edits of the original program. We then investigated why other APR tools did not repair programs that clearly had the set of expressions necessary for the repair. Notably, PREP's four transformation rules are not complete; for example, it does not include a rule to decompose binary operations with complex operands into independent expressions. However, each such code transformation also increases the search space, creating a trade-off between generality and efficiency. PREP represents our best guess about this trade-off, providing a small, general set of transformations that yield improvement across all tools that we studied.

Of code representation. Our results show that even small changes to the source code can greatly affect the code representation that a tool operates over. PREP effectively forces the granularity of the code representation to be more congruent with many operators. It is well-known that existing software-repair methods typically find repairs that involve one or at most two code mutations. The transformations we propose in PREP lead to a more uniform representation granularity because they separate out composite program elements into their constituent parts, e.g., decoupling function calls from conditionals. We speculate that such changes could enable EC-based repair tools to find more complex, epistatic, repairs by making it easier to recombine program elements [19]. These transformations can potentially disrupt an existing congruence, e.g., if a composite statement comprises a correct repair element and is congruent with an operator. Although we encountered a few such cases in our evaluation, on investigation those examples corresponded to repairs that were overfit to the test cases (see Section 5.2).

Despite their widespread use in industry, test cases can often lead to overfitting by providing an inadequate specification of correct functionality. Although we did not change the content of any tests in our evaluation, we found several examples of overfit patches in the baseline that had been reported for the different tools. These were blocked (not discovered) in the PREP-enabled evaluation. This shows how the source-level representation, particularly the runtime ramifications of the representation, can affect how tools search for a valid repair (cf. [11, 17, 27]).

During the Codeflaws GenProg evaluation, we also discovered a limitation with its configuration, which caused it to incorrectly interpret global variables that use macro definitions. Because PREP's preprocessing expands macros (required for T1–T4), PREP mitigated this particular issue.

On the stack. Although our transformations produce semantics-preserving code, they alter how stack resources are used during run time. Our transformations introduce new local variables within existing functions (T1–T3) and call new functions which initialize local variables with default values (T4). These transformations can change how the stack is used, depending on the resulting source code and compilation parameters. How compilers manage the stack during compilation is directed at a high level by the user through command-line options, directives, or pragmas.

For example, stack behavior can be changed by adding optimizations (at least `-O1`) or applying parameters, such as `-fcombine-stack-adjustments`. When new local variables are defined in the source code, a specific location is reserved on the stack based on the type and size of the variable. By contrast, when constant values such as strings or integers are used as function parameters, the compiler does not reserve additional stack space, but instead directly loads the value or address of the value into registers according to the calling convention. Although this location could be reused for other content when leaving the reserved variable's scope, this implies that the stack behavior will be different, potentially mitigating some existing stack-based bugs and vulnerabilities.

Although stack usage is transient, the introduction of new functions that initialize a number of local variables can change the values of stack locations, effectively resetting part of the stack to some default value. This can cause APR tools to not select patches that rely on stack dynamics. During testing, we observed that some positive and negative tests were volatile with respect to stack behavior.

Caveats. We ran f1x using two test oracle configurations on the CGC dataset. For the first evaluation, f1x found patches that did not pass the positive tests when evaluated outside of the f1x Docker container. We conjectured that this was caused by a virtualization or containerization instability, and we updated the test oracle to evaluate tests with 3 consecutive runs to ensure consistency and reduce any nondeterminism. In this second test oracle, if any run failed, that failure was reported, but all three test runs were required to report a pass for that test. This second oracle is the basis of the results we reported for f1x on the CGC dataset.

8 CONCLUSION

The source-to-source transformations encoded in PREP were developed to improve the performance of search-based APR tools by reducing incongruities between code representations and mutation operators. When the tools are executed with PREP they discover patches that were previously not discovered, finding **88 new unique correct patches** for the Codeflaws dataset. Because it operates on source code, PREP is general and applicable across tools, and it can be used in tandem with existing repair methods because it does not require modifications to the underlying algorithm. Although we implemented the PREP prototype for C, we believe that our approach would produce similar improvements for other imperative or object-oriented languages. We hope that our work not only improves the power of existing source-based software repair methods, but that it also draws attention to fundamental questions about how best to tailor search-based methods for the domain of software repair.

ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their valuable feedback. The authors gratefully acknowledge the partial support of NSF (CCF 1908633, OAC 2115075), DARPA (FA8750-19C-0003, N6600120C4020), AFRL (FA8750-19-1-0501), and the Santa Fe Institute.

REFERENCES

[1] Antlr Project. 2022. *ANTLR (ANother Tool for Language Recognition).* https://github.com/antlr/antlr4

[2] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–317.

[3] Satish Chandra and Thomas W. Reps. 1999. Physical type checking for C. *ACM SIGSOFT Softw. Eng. Notes* 24, 5 (1999), 66–75. https://doi.org/10.1145/381788.316183

[4] Codeforces. 2022. *Problemset.* https://codeforces.com/problemset/?order=BY%20SOLVED%20DESC

[5] DARPA. 2020. *Cyber Grand Challenge Sample Challenges.* https://github.com/CyberGrandChallenge/samples

[6] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1443–1450.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A generic method for automatic software repair. *ACM Trans. on Software Engineering* 38, 1 (2012). Featured article award.

[8] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[9] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[10] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Principles of Programming Languages*. 298–312.

[11] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering*. ACM, 702–713.

[12] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[13] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th international conference on software engineering*. 492–495.

[14] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4 (2018), 1–37.

[15] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[16] Vinicius Paulo L Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering* 23, 5 (2018), 2980–3006.

[17] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*. ACM, 24–36.

[18] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. 2022. Automatically Mitigating Vulnerabilities in x86 Binary Programs via Partially Recompilable Decompilation. *arXiv preprint arXiv:2202.12336* (2022).

[19] Joseph Renzullo, Westley Weimer, Melanie E. Moses, and Stephanie Forrest. 2018. Neutrality and epistasis in program space. In *Genetic Improvement*. ACM, 1–8.

[20] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Programming Language Design and Implementation*. 43–54.

[21] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 180–182.

[22] Trail of Bits. 2020. *DARPA Challenges Sets for Linux, Windows, and macOS.* https://github.com/trailofbits/cb-multios

[23] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Programming Language Design and Implementation*. 363–378.

[24] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 356–366.

[25] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[26] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825.

[27] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. In *International Conference on Software Engineering*. ACM, 24.

[28] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* 46, 10 (2018), 1040–1067.

[29] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53.