# *Automatic inference of code transforms for patch generation*

# Automatic Inference of Code Transforms for Patch Generation

Fan Long
MIT EECS & CSAIL, USA
fanl@csail.mit.edu

Peter Amidon
UCSD, USA
peter@picnicpark.org

Martin Rinard
MIT EECS & CSAIL, USA
rinard@csail.mit.edu

## ABSTRACT

We present a new system, Genesis, that processes human patches to automatically infer code transforms for automatic patch generation. We present results that characterize the effectiveness of the Genesis inference algorithms and the complete Genesis patch generation system working with real-world patches and defects collected from 372 Java projects. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from previous successful patches.

## CCS CONCEPTS

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

## KEYWORDS

Patch generation, Code transform, Search space inference

**ACM Reference Format:**
Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 13 pages.
https://doi.org/10.1145/3106237.3106253

## 1 INTRODUCTION

Automatic patch generation systems [30, 33–35, 37, 38, 40, 48, 56, 61, 62] hold out the promise of significantly reducing the human effort required to diagnose, debug, and fix software defects. The standard *generate and validate* approach starts with a set of test cases, at least one of which exposes the defect. It deploys a set of *transforms* to generate a *search space* of *candidate patches*, then runs the resulting patched programs on the test cases to find *plausible patches* that produce correct outputs for all test cases. All previous generate and validate systems work with a set of manually crafted transforms [33–35, 37, 38, 48, 56, 61, 62] to patch bugs that fall within the scope of these transforms.

### 1.1 Genesis

We present Genesis, a novel system that infers code transforms for automatic patch generation systems [36]. Given a set of successful human patches drawn from available revision histories, Genesis generalizes subsets of patches to infer transforms that together generate a productive search space of candidate patches. Genesis can therefore leverage the combined patch generation expertise of many different developers to capture a wide range of productive patch generation strategies. Genesis applies the inferred transforms to successfully patch bugs in previously unseen applications. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

**Transforms:** Each Genesis transform has two *template abstract syntax trees (ASTs).* One template AST matches code in the original program. The other template AST specifies the replacement code for the generated patch. Template ASTs contain *template variables*, which match subtrees or subforests in the original or patched code. Template variables enable the transforms to abstract away application-specific details to capture common patterns implemented by multiple patches drawn from different applications.

**Generators:** Many useful patches do not simply rearrange existing code and logic; they also introduce new code and logic. Genesis transforms therefore implement *partial pattern matching* in which the replacement template AST contains free template variables that are not matched in the original code. Each of the free template variables is associated with a *generator*, which systematically generates new candidate code components for the free variable. This new technique, which enables Genesis to synthesize new code and logic in the candidate patches, is essential to enabling Genesis to generate correct patches for previously unseen applications.

**Search Space Inference with ILP:** A key challenge in patch search space design is navigating an inherent tradeoff between coverage and tractability [39]. On one hand, the search space needs to be large enough to contain correct patches for the target class of defects (coverage). On the other hand, the search space needs to be small enough so that the patch generation system can efficiently explore the space to find the correct patches (tractability) [39].

Genesis navigates this tradeoff by formulating and solving an integer linear program (ILP) whose solution maximizes the number of training patches covered by the inferred search space while acceptably bounding the number of candidate patches that the search space can generate.

### 1.2 Experimental Results

We use Genesis to infer patch search spaces and generate patches for three classes of defects in Java programs: null pointer (NP), out of bounds (OOB), and class cast (CC) defects. Working with a training set that includes 483 NP patches, 199 OOB patches, and 287 CC patches drawn from 356 open source applications, Genesis infers a search space generated by 108 transforms.

Our benchmark defects include 20 NP, 13 OOB, and 16 CC defects from 41 open source applications. All of the benchmark applications are systematically collected from github [2] and contain up to 235K lines of code. With the 108 inferred transforms, Genesis generates


*ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany*

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106253

correct patches for 21 out of the 49 defects (11 NP, 6 OOB, and 4 CC defects).

We compare Genesis with PAR [33, 44], a previous patch generation system for Java that works with manually defined patch templates. For the same benchmark set, the PAR templates generate correct patches for 10 fewer defects (specifically, 7 NP and 4 OOB defects).

We attribute these results to the ability of the automated Genesis inference algorithms to navigate complex patch transform tradeoffs at scale. Genesis works with hundreds to over a thousand candidate transforms to obtain productive search spaces generated by tens to over a hundred selected transforms — many more transforms than any previous generate and validate system. Deploying this many transforms enables Genesis to capture a broad range of patch patterns, with the transforms selected to ensure the overall tractability and coverage of the resulting patch search space.

### 1.3 Contributions

**Transforms with Template ASTs and Generators:** We present novel transforms with template ASTs and generators for free template variables. These transforms enable Genesis to abstract away patch- and application-specific details to capture common patch patterns and strategies implemented by multiple patches drawn from different applications. Generators enable Genesis to synthesize the new code and logic required to obtain correct patches for defects that occur in large real-world applications.

**Patch Generalization:** We present a novel patch generalization algorithm that, given a set of patches, automatically derives a transform that captures the common patch generation pattern present in the patches. This transform can generate all of the given patches as well as other patches with the same pattern in the same or other applications.

**Search Space Inference:** We present a novel search space inference algorithm. Starting with a set of training patches, this algorithm infers a collection of transforms that together generate a search space of candidate patches with good coverage and tractability. The inference algorithm includes a novel sampling algorithm that identifies promising subsets of training patches to generalize and an ILP-based solution to the final search space selection problem.

**Complete System and Experimental Results:** We present a complete patch generation system, including defect localization and candidate patch evaluation algorithms, that uses the inferred search spaces to automatically patch defects in large real-world applications. We also present experimental results from this complete system.

To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from previous successful patches. All experimental data (including the Genesis source code, inferred templates, and generated patches) are available at http://groups.csail.mit.edu/pac/patchgen/.

## 2 TRANSFORM INFERENCE

We next present, via an example, an overview of the Genesis transform inference algorithm. Genesis works with a training set of successful human patches to infer a set of patch generation transforms. In our example, the training set consists of 963 human patches collected from 356 github repositories.

**Patch Sampling and Generalization:** The Genesis inference algorithm works with sampled subsets of patches from the training set. For each subset, it applies a *generalization* algorithm to infer a *transform* that it can apply to generate candidate patches (Section 4.3). Figure 1 presents one of the sampled subsets of patches in our example: the first patch disjoins the clause `mapperTypeElement==null` to an if condition, the second patch conjoins the clause `subject!=null` to a return value, and the third patch conjoins the clause `Material.getMaterials(getTypeId())!=null` to an if condition. These patches are from three different applications, specifically mapstruct [20] 6d7a4d, modelmapper [22] d85131, and Bukkit [6] f13115. Genesis generalizes these patches to infer the transform $\mathcal{P}_1$ in Figure 1. When applied, $\mathcal{P}_1$ can generate all of the sampled three patches as well as other patches for other applications.

**Template Anatomy:** Each transform has a *template*. In our example, the template is $V_0 \implies ((V_3)op_2(\text{null}))op_1(V_0)$ (Figure 1 presents this template in graphical form). The transform has an *initial template AST* $\mathcal{T}_0$, which matches a boolean expression $V_0$ in the unpatched program. $V_0$ must occur within a function body (if all of the training patches had modified if conditions, $\mathcal{T}_0$ would have reflected that more specific context).

The transform also has a *replacement template AST* $\mathcal{T}_1$, which replaces the matched boolean expression $V_0$ with a patch of the form $((V_3)op_2(\text{null}))op_1(V_0)$. Here $V_3$, $op_2$, and $op_1$ are *unmatched template variables*. Each such variable is associated with a *generator*, which enumerates candidate code components for the variable.

**Generator Constraints:** *Generator constraints* control the components that the generator will enumerate. The generator constraints for $op_2$ and $op_1$ ($op_2 \in \{==, !=\}$ and $op_1 \in \{\&\&, ||\}$) simply specify sets of operators to enumerate. The generator constraints for $V_3$ control the AST subtrees that the generator will enumerate for $V_3$. $V_3 \in$ Expr states that $V_3$ must be an expression. nodes$(V_3) \subseteq$ Call $\cup$ Var states that $V_3$ can contain only method calls or variable references. $|V_3| \leq 2$ states that $V_3$ can contain at most 2 AST nodes.

vars$(V_3) \subseteq$ M states that any variables that appear in $V_3$ must also appear in the matched template AST $V_0$ (here M denotes the set of nodes in the original matched code). $|\text{vars}(V_3)| \leq 1$ states that at most 1 variable can appear in $V_3$. calls$(V_3) \subseteq$ M and $|\text{calls}(V_3)| \leq 2$ similarly constrain the method calls that may appear in $V_3$.

As these generator constraints illustrate, the Genesis patch generalization algorithm infers the *least general* Genesis transform that generates all of the sampled training patches. This strategy is critical for obtaining precisely targeted transforms that produce a tractable number of patches in the patch search space.

**Candidate Transforms:** Genesis repeatedly samples training patches to obtain the *candidate transforms* (from which Genesis will select the *selected transforms* that it uses for patch generation). In

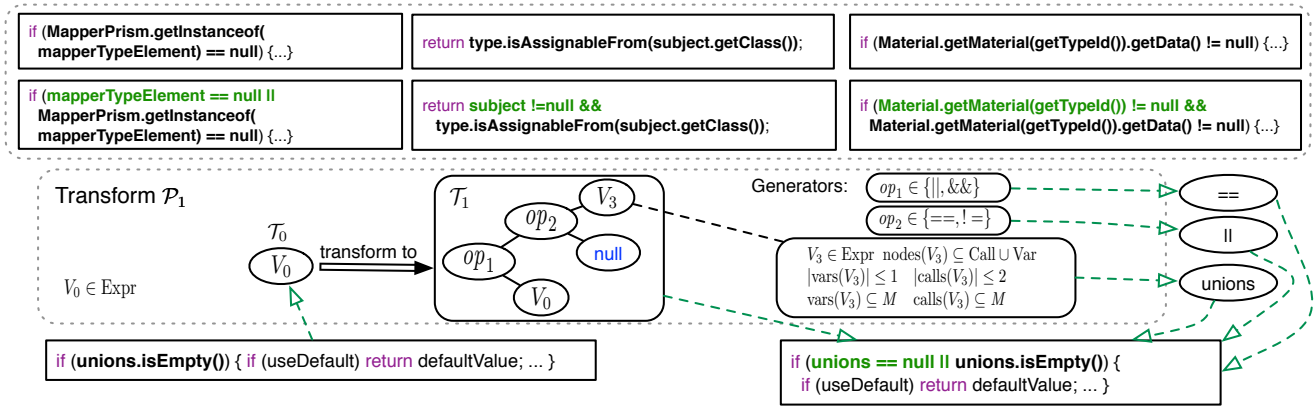


**Figure 1: Example inference and application of a Genesis transform. The training patches (original and patched code) are at the top, the inferred transform is in the middle, and the new patch that Genesis generates is at the bottom.**

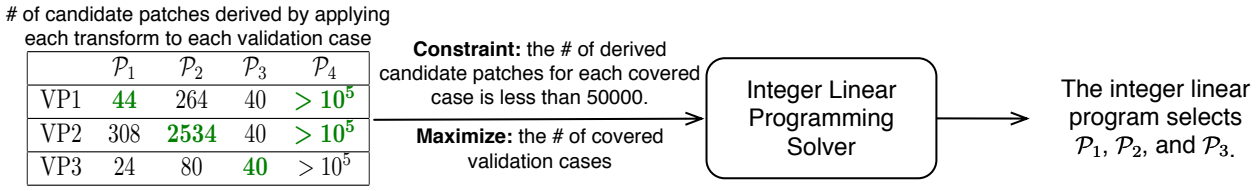


**Figure 2: Using integer linear programming to select an effective set of transforms.**

our example the candidate transforms include the previous transform $\mathcal{P}_1$ as well as a transform ($\mathcal{P}_2$) that adds a conditional (ternary) operator to guard the computation of an expression from NP defects, a transform ($\mathcal{P}_3$) that adds an if-guarded return or continue statement to skip the computation that triggers NP defects, and a transform $\mathcal{P}_4$ that replaces an arbitrary expression with a new expression. The new expression may contain binary operators, conditional operators, and up to six variables and six method calls from the enclosing function.

Not all of these transforms are equally useful. $\mathcal{P}_4$, for example, is an overly general transform that can generate an intractably large patch search space that Genesis cannot search effectively. $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$, on the other hand, are more targeted — because they were inferred from conceptually similar training patches, each generates a much smaller search space that nevertheless contains correct patches. And $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ effectively complement each other — their generated search spaces have relatively few patches in common.

**Search Space Inference:** To obtain an effective set of transforms, Genesis must discard overly general transforms such as $\mathcal{P}_4$ and include complementary and effectively targeted transforms such as $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$. Genesis drives the transform selection with a set of *validation patches* chosen from the training patches. Genesis starts by computing the number of validation patches that each candidate transform generates and the size of the search space that each candidate transform generates when applied to the pre-patch code for each validation patch.

The matrix in Figure 2 presents these numbers for the four candidate transforms $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$ and three validation patches VP1, VP2, and VP3 (in our example the validation patches are drawn from joda-time [17] revision bcb044, dynjs [8] revision 68df61, and orientdb [21] revision 51706f). Each number in the matrix is the number of candidate patches that a transform generates when applied to the pre-patch code of a validation patch. A **bold green number** indicates that a transform can generate the validation patch when applied to the pre-patch code of the patch. These numbers highlight the coverage vs. tractability tradeoff that the candidate patches present. With tractable search spaces, $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ all generate a single validation patch. $\mathcal{P}_4$, in contrast, generates two validation patches but at the cost of an intractably large search space.

Working with the information from the matrix, Genesis formulates an integer linear program (ILP) that maximizes the number of validation patches that the selected transforms can generate subject to the constraint that the total number of generated candidate patches from all selected transforms for each covered validation case is less than $5 \times 10^4$. In our example the ILP selects $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ as the selected transforms and excludes $\mathcal{P}_4$.

**Patch Generation:** For the NP defect from DataflowJavaSDK [7] revision c06125 (shown at the bottom of Figure 1), Genesis first uses a defect localization technique (Section 5) to produce a ranked list of potential statements to modify. The resulting ranked list includes the if condition shown at the bottom left of Figure 1. Genesis then applies all selected transforms, including $\mathcal{P}_1$, to the if condition to generate candidate patches.

Figure 1 shows how Genesis applies $\mathcal{P}_1$ to the if condition. Here the patch instantiates $V_3$ as the variable `unions`, $op_2$ as == and $op_3$

as || to disjoin the clause `unions == null` to the original if condition. The patch causes the enclosing function `innerGetOnly()` to return a predefined default value when `unions` is `null` (instead of incorrectly throwing a null pointer exception). This patch validates (produces correct outputs for all inputs in the DataflowJavaSDK JUnit [19] test suite), is correct, and matches the subsequent human developer patch for this defect.

## 3 INFERRED TRANSFORMS

The implemented Genesis system selects 108 transforms from a space of 577 sampled transforms. We next present an overview of these 108 transforms.

**Transforms That Target Boolean Expressions:** Many defects involve incorrect boolean expressions [27, 37, 38, 40]. It is therefore not surprising that many of the inferred transforms target boolean expressions. Specifically, 17 of the 108 transforms (including the example transform discussed in Section 2) conjoin or disjoin a generated subexpression to a boolean condition in the original program.

**Conditional Execution:** 6 transforms conditionally execute existing matched code. 3 of the 6 implement a direct null pointer check, with the matched code executing only if the pointer is not null. There are also composite transforms, such as a transform that 1) initializes a variable to the result of a method call and 2) avoids a null pointer error by wrapping the intialization and subsequent relevant code in a null pointer check:

$$V_2\ V_1 = V_0; \text{if}(V_5 op_4 V_6)\{V_8\}; V_9 \Longrightarrow$$
$$\text{if}(V_{10}! = \text{null})\{V_2\ V_1 = V_{10}.V_{12}(); \text{if}(V_5 op_4 V_6)\{V_8\}\}; V_9$$

**Inserted If Then Else:** 8 transforms wrap existing code in an if then else statement. The generated condition tests for a previously unhandled case, generated code handles the case on one branch, and the transform places existing code in the other branch. One transform, for example, inserts a null pointer check and generates code to return an empty array if the check succeeds. Another generates an equality check and sets a variable to a different value on the new branch:

$$V_0 = V_1; \Longrightarrow \text{if}(V_3 == V_4)\{V_0 = V_6;\}\text{else}\{V_0 = V_1;\}$$

**Inserted If Then:** 12 transforms insert conditionally executed generated code — the condition tests for a previously unhandled case. The generated code executes when the case occurs. 6 of the 12 transforms directly check for various null pointer cases, for example:

$$V_0 \Longrightarrow \text{if}(V_1 == \text{null})\{V_4\}; V_0$$

**Replace Code:** 29 transforms replace existing code with newly generated code. The transforms differ in 1) the form of the code they replace and generate and 2) the generator constraints. There are also several transforms that replace most but not all of the existing code. The following method call transform, for example, replaces the invoked method and parameters, but keeps the original receiver:

$$V_2.V_1(V_0) \Longrightarrow V_2.V_6(V_5)$$

**Try/Catch/Continue:** One transform wraps existing code in a try construct with an empty catch block. Like failure-oblivious computing [53], the patch discards the exception and continues execution:

$$V_0 \Longrightarrow \text{try}\{V_0\}\text{catch}(V_2)\{\}$$

**For Loop Off By One:** One transform corrects off by one errors in for loops, specifically by enumerating combinations of starting values and loop termination conditions as follows:

$$\text{for}(V_1\ V_4 = 0; V_0 < V_2; V_0 + +)\{V_5\} \Longrightarrow$$
$$\text{for}(V_1\ V_4 = V_7; V_0 op_6 V_2; V_0 + +)\{V_5\}, V_7 \in \{0, 1\}, op_6 \in \{<, \leq\}$$

**Change Declared Type:** One transform changes the declared type (and potentially also the initializer) of a variable declaration. This transform generates patches that eliminate class cast exceptions, specifically by moving the declared type up in the class hierarchy ($V_2$ is the original declared type and $V_4$ is the new declared type):

$$V_2\ V_1 = V_0 \Longrightarrow V_4\ V_1 = V_3$$

**Other Transforms:** Genesis also infers a variety of more specialized transforms that, for example, combine null check insertions with method receiver replacement or a return of null. It also infers 20 transforms that, in our judgement, are specific to the defects in the training set and are unlikely to be useful for other defects. Even though these transforms are unlikely to correct any other defects, because they are so specific, they impose negligible search overhead.

**Discussion:** In comparison with previous manually developed transforms [33, 37, 38], the Genesis transforms are more numerous, more diverse, and target a wider range of defects more precisely and tractably. Some transforms target specific defect classes such as off by one defects in for loops. Other transforms apply general templates (for example, replacing one expression with another expression), with the generator constraints controlling the enumeration to deliver a tractable search space. While the inferred templates often correspond to intuitive patch generation patterns that can correct a wide range of defects, the generator constraints typically more closely reflect the specific characteristics of the patches in the training set. For example, many generator constraints focus the transform on introducing instanceof checks (to patch class cast exceptions), null pointer checks (to patch null pointer defects), or checks involving comparison operators such as $<$, $\leq$, $>$, or $\geq$ (to patch out of bounds defects). As the above discussion highlights, the combination of transform inference and generator constraints enables Genesis to infer a rich, precisely targeted, but still tractable patch search space.

## 4 INFERENCE SYSTEM

We next present the Genesis inference system. Given a set of training pairs $D$, each of which corresponds to a program before a change and a program after a change, Genesis infers a set of transforms $\mathbb{P}$ that, working together, generate the search space of patches.

### 4.1 Preliminaries

Genesis works with abstract syntax trees (ASTs) of programs. We model the programming language that Genesis works with as a context free grammar (CFG) with abstract syntax trees (AST) as the parse trees for the CFG.

*Definition 4.1 (CFG).* A *context free grammar (CFG)* $G$ is a tuple $\langle N, \Sigma, R, s\rangle$ where $N$ is the set of non-terminals, $\Sigma$ is the set of terminals, $R$ is a set of production rules of the form $a \rightarrow b_1 b_2 b_3 \ldots b_k$ where $a \in N$ and $b_i \in N \cup \Sigma$, and $s \in N$ is the starting non-terminal of the grammar. The language of $G$ is the set of strings derivable from the start non-terminal: $\mathcal{L}(G) = \{w \in \Sigma^* \mid s \Rightarrow^* w\}$.

*Definition 4.2 (AST).* An *abstract syntax tree (AST)* $T$ is a tuple $\langle G, X, r, \xi, \sigma \rangle$ where $G = \langle N, \Sigma, R, s \rangle$ is a CFG, $X$ is a finite set of nodes in the tree, $r \in X$ is the root node of the tree, $\xi : X \to X^*$ maps each node to the list of its children nodes, and $\sigma : X \to (N \cup \Sigma)$ attaches a non-terminal or terminal label to each node in the tree.

*Definition 4.3 (AST Traversal and Valid AST).* Given an AST $T = \langle G, X, r, \xi, \sigma \rangle$ where $G = \langle N, \Sigma, R, s \rangle$, $\text{str}(T)$ denotes the terminal string obtained via traversing $T$. $T$ is a valid AST of $G$ iff $\text{str}(T) \in \mathcal{L}(G)$.

We next define AST forests and AST slices, which we will use to present the Genesis inference algorithm. An AST forest is similar to an AST except it contains multiple trees and a list of root nodes. An AST slice is a special forest inside a large AST which corresponds to a list of adjacent siblings.

*Definition 4.4 (AST Forest).* An AST forest $T$ is a tuple $\langle G, X, L, \xi, \sigma \rangle$ where $G$ is a CFG, $X$ is the set of nodes in the forest, $L = \langle x_1, x_2, \ldots, x_k \rangle$ is the list of root nodes of trees in the forest, $\xi$ maps each node to the list of its children nodes, and $\sigma$ maps each node in $X$ to a non-terminal or terminal label.

*Definition 4.5 (AST Slice).* An AST slice $S$ is a pair $\langle T, L \rangle$. $T = \langle G, X, r, \xi, \sigma \rangle$ is an AST; $L = \langle r \rangle$ is a list that contains only the root node or $L = \langle x_{c_i}, \ldots, x_{c_j} \rangle$ is a list of AST sibling nodes in $T$ such that $\exists x' \in X : \xi(x') = \langle x_{c_1}, \ldots, x_{c_i}, \ldots, x_{c_j}, \ldots, x_{c_k} \rangle$ (i.e., $L$ is a sublist of $\xi(x')$).

Given two ASTs $T$ and $T'$, where $T$ is the AST before the change and $T'$ is the AST after the change, Genesis computes AST difference between $T$ and $T'$ to produce an AST slice pair $\langle S, S' \rangle$ such that $S$ and $S'$ point to the sub-forests in $T$ and $T'$ that subsume the change. For brevity, in this section we assume $D = \{\langle S_1, S'_1 \rangle, \langle S_2, S'_2 \rangle, \ldots, \langle S_m, S'_m \rangle\}$ is a set of AST slice pairs, i.e., Genesis has already converted AST pairs of changes to AST slices.

**Notation and Utility Functions:** For a map $M$, $\text{dom}(M)$ denotes the domain of $M$. $M[a \mapsto b]$ denotes the new map which maps $a$ to $b$ and maps other elements in $\text{dom}(M)$ to the same values as $M$. $\emptyset$ denotes an empty set or an empty map.

$\text{nodes}(\xi, L)$ denotes the set of nodes in a forest, where $\xi$ maps each node to a list of its children and $L$ is the list of the root nodes of the trees in the forest.

$\text{inside}(S)$ denotes the set of non-terminals of the ancestor nodes of an AST slice $S$.

$\text{nonterm}(L, X, \xi, \sigma, N)$ denotes the set of non-terminals inside a forest, where $L$ is the root nodes in the forest, $X$ is a finite set of nodes, $\xi$ maps each node to a list of children nodes, $\sigma$ attaches each node to a terminal or non-terminal label, and $N$ is the set of non-terminals.

$\text{diff}(A, B)$ denotes the number of different terminals in leaf nodes between two ASTs, AST slices, or AST forests. If $A$ and $B$ differ in not just terminals in leaf nodes, $\text{diff}(A, B) = \infty$. $A \equiv B$ denotes that $A$ and $B$ are equivalent, i.e., $\text{diff}(A, B) = 0$.

## 4.2 Template AST Forest, Generator, Transforms

**Template AST Forest:** We next introduce the template AST forest, which can represent a set of concrete AST forests or slices. The key difference between template and concrete AST forests is that template AST forests contain template variables, each of which can match against any appropriate AST subtree or AST sub-forest.

*Definition 4.6 (Template AST Forest).* A template AST forest $\mathcal{T}$ is a tuple $\langle G, V, \gamma, X, L, \xi, \sigma \rangle$, where $G = \langle N, \Sigma, R, s \rangle$ is a CFG, $V$ is a finite set of template variables, $\gamma : V \to \{0, 1\} \times \text{Powerset}(N)$ is a map that assigns each template variable to a bit of zero or one and a set of non-terminals, $X$ is a finite set of nodes in the subtree, $L = \langle x_1, x_2, \ldots, x_k \rangle$, $x_i \in X$ is the list of root nodes of the trees in the forest, $\xi : X \to X^*$ maps each node to the list of its children nodes, and $\sigma : X \to N \cup \Sigma \cup V$ attaches a non-terminal, a terminal, or a template variable to each node.

For each template variable $v \in V$, $\gamma(v) = \langle b, W \rangle$ determines the kind of AST subtrees or sub-forests which the variable can match against. If $b = 0$, $v$ can match against only AST subtrees not sub-forests. If $b = 1$, then $v$ can match against both subtrees and sub-forests. Additionally, $v$ can match against an AST subtree or sub-forest only if its roots have non-terminals in $W$.

Intuitively, each non-terminal in the CFG of a programming language typically corresponds to one kind of syntactic unit in programs at a certain granularity. Template AST forests with template variables enable Genesis to achieve a desirable abstraction over concrete AST trees during the inference. They also enable Genesis to abstract away program-specific syntactic details so that Genesis can infer useful transforms from changes across different applications.

*Definition 4.7 (The $\models$ and $\models_{\texttt{slice}}$ Operators for Template AST Forests).* Figure 3 presents the formal definition of the operator $\models$ for a template AST forest $\mathcal{T} = \langle G, V, \gamma, X, L, \xi, \sigma \rangle$. $\mathcal{T} \models \langle T, M \rangle$ denotes that $\mathcal{T}$ matches the concrete AST forest $T$ with the template variable bindings specified in $M$, where $M$ is a map that assigns each template variable in $V$ to an AST forest.

Figure 3 also presents the formal definition of the operator $\models_{\texttt{slice}}$. Similarly, $\mathcal{T} \models_{\texttt{slice}} \langle S, M \rangle$ denotes that $\mathcal{T}$ matches the concrete AST slice $S$ with the variable bindings specified in $M$.

The first rule in Figure 3 corresponds to the simple case of a single terminal node. The second and the third rules correspond to the cases of a single non-terminal node or a list of nodes, respectively. The two rules recursively match the children nodes and each individual node in the list.

The fourth and the fifth rules correspond to the case of a single template variable node in the template AST forest. The fourth rule matches the template variable against a forest, while the fifth rule matches the variable against a tree. These two rules check that the corresponding forest or tree of the variable in the binding map $M$ is equivalent to the forest or tree that the rules are matching against.

**Generators:** Generators enumerate new code components:

*Definition 4.8 (Generator).* A generator $\mathcal{G}$ is a tuple $\langle G, b, \delta, W \rangle$, where $G = \langle N, \Sigma, R, s \rangle$ is a CFG, $b \in \{0, 1\}$ indicates the behavior of the generator, $\delta$ is an integer bound for the number of tree nodes, and $W \subseteq N$ is the set of allowed non-terminals during generation.

Generators exhibit two kinds of behaviors. If $b = 0$, the generator generates a sub-forest with less than $\delta$ nodes that contains only non-terminals inside the set $W$. If $b = 1$, the generator copies an existing sub-forest from the original AST tree with non-terminal labels in $W$ and then replaces up to $\delta$ leaf nodes in the copied sub-forest.

$$\frac{\begin{array}{c}G = \langle N, \Sigma, R, s\rangle \\ \mathcal{T} = \langle G, V, \gamma, X, L, \xi, \sigma\rangle \quad L = \langle x_1, x_2, \ldots, x_k\rangle \\ T = \langle G, X', L', \xi', \sigma'\rangle \quad L' = \langle x'_1, x'_2, \ldots, x'_{k'}\rangle\end{array}}{}$$

$$\frac{k = k' = 1 \qquad \sigma(x_1) = \sigma'(x'_1) \in \Sigma}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c}k = k' = 1 \qquad \sigma(x_1) = \sigma'(x'_1) \in N \\ \langle G, V, \gamma, X, \xi(x_1), \xi, \sigma\rangle \models \langle\langle G, X', \xi'(x'_1), \xi', \sigma'\rangle, M\rangle\end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c}k = k' > 1 \\ \forall i \in \{1, 2, \ldots, k\}\left(\langle G, V, \gamma, X, \{x_i\}, \xi, \sigma\rangle \models \langle\langle G, X', \{x'_i\}, \xi', \sigma'\rangle, M\rangle\right)\end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c}k = 1 \qquad \sigma(x_1) = v \in V \\ M(v) \equiv T \qquad \gamma(v) = \langle 1, W\rangle \qquad (\cup_{i=1}^{k'} \sigma'(x'_i)) \subseteq (W \cup \Sigma)\end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\begin{array}{c}k = k' = 1 \qquad \sigma(x_1) = v \in V \\ M(v) \equiv T \qquad \gamma(v) = \langle 0, W\rangle \qquad \sigma'(x'_1) \in (W \cup \Sigma)\end{array}}{\mathcal{T} \models \langle T, M\rangle}$$

$$\frac{\mathcal{T} \models \langle\langle G, X', L', \xi', \sigma'\rangle, M\rangle}{\mathcal{T} \models_{slice} \langle\langle\langle G, X', r', \xi', \sigma'\rangle, L'\rangle, M\rangle}$$

**Figure 3: Definitions of $\models$ and $\models_{\texttt{slice}}$**

$$\frac{\begin{array}{c}G = \langle N, \Sigma, R, s\rangle \qquad S = \langle T, L\rangle \\ T = \langle G, X, r, \xi, \sigma\rangle \quad T' = \langle G, X', L', \xi', \sigma'\rangle\end{array} \quad \begin{array}{c}|\text{nodes}(\xi', L')| \leq \delta \qquad \text{nonterm}(L', X', \xi', \sigma', N) \subseteq W \\ \hline \langle\langle G, 0, \delta, W\rangle, S\rangle \Longrightarrow T'\end{array}}{}$$

$$\frac{\begin{array}{c}\exists x' \in X\ (L'' \text{ is a sublist of } \xi(x')) \\ \text{diff}(\langle G, X, L'', \xi, \sigma\rangle, T') \leq \delta \qquad \forall x'' \in L'\ (\sigma'(x'') \in W)\end{array}}{\langle\langle G, 1, \delta, W\rangle, S\rangle \Longrightarrow T'}$$

**Figure 4: Definition of the $\Longrightarrow$ for the generator $\mathcal{G} = \langle G, b, \delta, W\rangle$**

*Definition 4.9 (Generation Operator $\Longrightarrow$).* Figure 4 presents the formal definition of the operator $\Longrightarrow$ for a generator $\mathcal{G}$. Given $\mathcal{G}$ and an AST slice $S = \langle T, L\rangle$ as the context, $\langle\mathcal{G}, S\rangle \Longrightarrow T'$ denotes that the generator $\mathcal{G}$ generates the AST forest $T'$.

The first rule in Figure 4 handles the case where $b = 0$. The rule checks that the number of nodes in the result forest is within the bound $\delta$ and the set of non-terminals in the forest is a subset of $W$. The second rule handles the case where $b = 1$. The rule checks that the difference result forest and an existing forest in the original AST is within the bound and the root labels are in $W$.

**Transforms:** Finally, we introduce transforms, which generate the search space inferred by Genesis. Given an AST slice, a transform generates new AST trees.

*Definition 4.10 (Transform).* A transform $\mathcal{P}$ is a tuple $\langle A, \mathcal{T}_0, \mathcal{T}_1, B\rangle$. $A \subseteq N$ is a set of non-terminals that denote the context where this transform can apply; $\mathcal{T}_0 = \langle G, V_0, \gamma_0, X_0, L_0, \xi_0, \sigma_0\rangle$ is the template AST forest before applying the transform; $\mathcal{T}_1 = \langle G, V_1, \gamma_1, X_1, L_1, \xi_1, \sigma_1\rangle$ is the forest after applying the transform; $B$ maps each template variable $v$ that appears only in $\mathcal{T}_1$ to a generator (i.e., $\forall v \in V_1 \setminus V_0$, $B(v)$ is a generator).

$$\frac{\begin{array}{c}S = \langle\langle G, X, r, \xi, \sigma\rangle, L\rangle \qquad A \subseteq \text{inside}(S) \\ \mathcal{T}_0 \models \langle S, M\rangle \qquad B = \{v_1 \mapsto \mathcal{G}_1, v_2 \mapsto \mathcal{G}_2, \ldots, v_m \mapsto \mathcal{G}_m\} \\ \forall_{i=1}^{m}\left(\langle\mathcal{G}_i, S\rangle \Longrightarrow T''_i\right) \qquad M' = \{v_1 \mapsto T''_1, v_2 \mapsto T''_2, \ldots v_k \mapsto T''_m\} \\ \mathcal{T}_1 \models \langle T', M \cup M'\rangle \qquad \langle S, T'\rangle \triangleright T \qquad \text{str}(T) \in \mathcal{L}(G)\end{array}}{\langle\langle A, \mathcal{T}_0, \mathcal{T}_1, B\rangle, S\rangle \Longrightarrow T}$$

$$\frac{\begin{array}{c}1 \leq i \leq j \leq k \\ S = \langle\langle G, X, r, \xi, \sigma\rangle, L\rangle \quad L = \langle x_i, \ldots, x_j\rangle \quad \xi(x') = \langle x_1, x_2, \ldots, x_k\rangle \\ T' = \langle G, X', L', \xi', \sigma'\rangle \quad L' = \langle x''_1, x''_2, \ldots, x''_{k'}\rangle \qquad X \cap X' = \emptyset \\ L'' = \langle x_1, \ldots, x_{i-1}, x''_1, x''_2, \ldots, x''_{k'}, x_{j+1}, \ldots, x_k\rangle\end{array}}{\langle S, T'\rangle \triangleright \langle G, X \cup X', r, (\xi \cup \xi')[x' \mapsto L''], \sigma \cup \sigma'\rangle}$$

$$\frac{S' = \langle T', L'\rangle \qquad \langle\mathcal{P}, S\rangle \Longrightarrow T'}{\langle\mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'}$$

**Figure 5: Definition of $\Longrightarrow$ and $\Longrightarrow_{\texttt{slice}}$ for the transform $\mathcal{P}$**

*Definition 4.11 ($\Longrightarrow$ and $\Longrightarrow_{\texttt{slice}}$ Operators).* Figure 5 presents the formal definition of the $\Longrightarrow$ and $\Longrightarrow_{\texttt{slice}}$ operators for a transform $\mathcal{P}$. $\langle\mathcal{P}, S\rangle \Longrightarrow T'$ denotes that applying $\mathcal{P}$ to the AST slice $S$ generates the new AST $T'$. $\langle\mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'$ denotes that applying $\mathcal{P}$ to the AST slice $S$ generates the AST of the slice $S'$.

Intuitively, in Figure 5 $A$ and $\mathcal{T}_0$ determine the context where the transform $\mathcal{P}$ can apply. $\mathcal{P}$ can apply to an AST slice $S$ only if the ancestors of $S$ have all non-terminal labels in $A$ and $\mathcal{T}_0$ can match against $S$ with a variable binding map $M$. $\mathcal{T}_1$ and $B$ then determine the transformed AST tree. $\mathcal{T}_1$ specifies the new arrangement of various components and $B$ specifies the generators to generate AST sub-forests to replace free template variables in $\mathcal{T}_1$. Note that $\langle S, T'\rangle \triangleright T$ denotes that the obtained AST tree of replacing the AST slice $S$ with the AST forest $T'$ is equivalent to $T$.

## 4.3 Transform Generalization

The generalization operation for transforms takes a set of AST slice pairs $D$ as input and produces a set of transforms, each of which can at least generate the corresponding changes of the pairs in $D$.

*Definition 4.12 (Generator Generalization).* Figure 6 presents the definition of the generalization function $\psi(D)$. Given a set of of AST slice pairs $D = \{\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\}$ from the same CFG grammar $G$, where $S_i$ is the generation context AST slice and $S'_i$ is the generated result AST slice, $\psi(D) = \{\mathcal{G}_1, \mathcal{G}_2, \ldots \mathcal{G}_k\}$ denotes the set of the generators generalized from $D$.

In Figure 6, $\mathbb{A}$ is the formula for a generator that generates from scratch (i.e., $b = 0$) and $\mathbb{B}$ is the formula for a generator that generates via copying from the existing AST tree (i.e., $b = 1$). The formula $\mathbb{A}$ produces the generator by computing the bound of the number of nodes and the set of non-terminals in the supplied slices. The formula $\mathbb{B}$ produces the generator by computing 1) the bound of the minimum diff distance between each supplied slice and an arbitrary existing forest in the AST tree and 2) the set of non-terminals of the root node labels of the supplied slices.

*Definition 4.13 (Transform Generalization).* Figure 7 presents the definition of $\Psi(D)$. Given a set of pairs of AST slices $D = \{\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\}$ where $S_i$ is the AST slice before a change and $S'_i$ is the AST slice after a change, $\Psi(D)$ is the set of transforms generalized from $D$.

$\mathbb{S} = \langle S_1, S_2, \ldots, S_m \rangle$ $G = (N, \Sigma, R, s)$ $x'$ is a fresh node $v'$ is a fresh template variable
$\forall i \in \{1, 2, \ldots, m\}:$ $S_i = \langle T_i, L_i \rangle$ $L_i = \langle x_{i,1}, x_{i,2}, \ldots, x_{i,k_i} \rangle$ $T_i = \langle G, X_i, r_i, \xi_i, \sigma_i \rangle$ $c_i = \sigma_i(x_{i,1})$

| $\Psi'(\mathbb{S}, M) =$ | **Conditions for $k$ and $c$** | **Other Conditions** |
|---|---|---|
| $\langle\langle G, \emptyset, \emptyset, \langle\rangle, \emptyset, \emptyset\rangle, M\rangle$ | $\forall i \in \{1, \ldots, m\}\ k_i = 0$ | |
| $\langle\langle G, \emptyset, \emptyset, \{x'\}, \langle x'\rangle, \{x' \mapsto \emptyset\}, \{x' \mapsto d\}\rangle, M\rangle$ | $d \in \Sigma \quad \forall i \in \{1, \ldots, m\}$ $k_i = 1 \quad c_i = d$ | |
| $\langle\langle G, V, \gamma, X' \cup \{x'\}, \langle x'\rangle, \xi'[x' \mapsto L'], \sigma'[x' \mapsto d]\rangle, M'\rangle$ | $d \in N \quad \forall i \in \{1, \ldots, m\}$ $k_i = 1 \quad c_i = d$ | $\mathbb{S}' = \langle\langle T_1, \xi_1(x_{1,1})\rangle, \langle T_2, \xi_2(x_{2,1})\rangle, \ldots, \langle T_m, \xi_m(x_{m,1})\rangle\rangle$ $\Psi'(\mathbb{S}', M) = \langle \mathcal{T}, M'\rangle \quad \mathcal{T} = \langle G, V, \gamma, X', L', \xi', \sigma'\rangle$ |
| $\langle\langle G, \{v\}, \{v \mapsto \langle 0, W\rangle\}, \{x'\}, \langle x'\rangle, \{x' \mapsto \emptyset\}, \{x' \mapsto v\}\rangle, M\rangle$ | $\exists i, i' \in \{1, \ldots, m\}$ $c_i \neq c_{i'}$ | $M(v) = \langle 0, W, \langle S'_1, S'_2, \ldots, S'_m\rangle\rangle$ $\forall i \in \{1, \ldots, m\}\ \left(S_i \equiv S'_i\right)$ |
| $\langle\langle G, \{v'\}, \{v' \mapsto \langle 0, W\rangle\}, \{x'\}, \langle x'\rangle, \{x' \mapsto \emptyset\}, \{x' \mapsto v'\}\rangle, M'\rangle$ | $\forall i \in \{1, \ldots, m\} \quad k_i = 1$ $\exists i', i'' \in \{1, \ldots, m\}$ $(c_{i'} \neq c_{i''})$ | $\forall v \in \mathrm{dom}(M)$ $M(v) = \langle 0, W', \langle S'_1, S'_2, \ldots, S'_m\rangle\rangle \quad \exists i \in \{1, 2, \ldots, m\}\ \left(S_i \not\equiv S'_i\right)$ $W = N \cap (\cup_{i=1}^{m} \{\sigma_i(x_{i,1})\}) \quad M' = M[v' \mapsto \langle 0, W, \mathbb{S}\rangle]$ |

**Figure 8: Definition of $\Psi'$**

$G = (N, \Sigma, R, s)$ $D = \langle\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\rangle$
$\forall i \in \{1, 2, \ldots, m\}:$
$S_i = \langle T_i, L_i\rangle$ $T_i = \langle G, X_i, r_i, \xi_i, \sigma_i\rangle$
$S'_i = \langle T'_i, L'_i\rangle$ $T'_i = \langle G, X'_i, r'_i, \xi'_i, \sigma'_i\rangle$
$L'_i = \langle x'_{i,1}, x'_{i,2}, \ldots, x'_{i,k'_i}\rangle$

$$\psi(D) = \begin{cases} \{\mathbb{A}, \mathbb{B}\} & \forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, k'_i\}, \sigma'(x'_{i,j}) \in N \\ \{\mathbb{A}\} & \text{otherwise} \end{cases}$$

where:
$\mathbb{A} = \langle G, 0, \max_{i=1}^{m} |\mathrm{nodes}(\xi'_i, L'_i)|, \bigcup_{i=1}^{m} \mathrm{nonterm}(S'_i)\rangle$
$\mathbb{B} = \langle G, 1, \max_{i=1}^{m} \mathbb{C}_i, \bigcup_{i=1}^{m} \bigcup_{j=1}^{k'_i} \{\sigma'(x'_{i,j})\}\rangle$
$\mathbb{C}_i = \min_{L''_i} \mathrm{diff}(\langle T_i, L''_i\rangle, S'_i), \exists x'' \in X_i,\ L''_i$ is a sublist of $\xi_i(x'')$

**Figure 6: Definition of the generator inference operator $\psi$**

$$\begin{aligned}
&\Psi(\langle\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\rangle) = \\
&\{\langle \cap_{i=1}^{m} \mathsf{inside}(S_i), \mathcal{T}_0, \mathcal{T}_1, B\rangle \mid \\
&\quad \langle \mathcal{T}_0, M\rangle = \Psi'(\langle S_1, S_2, \ldots, S_m\rangle, \emptyset), \\
&\quad \langle \mathcal{T}_1, M'\rangle = \Psi'(\langle S'_1, S'_2, \ldots, S'_m\rangle, M), \\
&\quad B = \{v_i \mapsto \mathcal{G}_i \mid \\
&\qquad\quad v_i \in \mathrm{dom}(M') \setminus \mathrm{dom(M)}, \\
&\qquad\quad M'(v_i) = \langle b_i, W_i, \langle S''_{i,1}, S''_{i,2}, \ldots, S''_{i,m}\rangle\rangle, \\
&\qquad\quad P_i = \{\langle S_1, S''_{i,1}\rangle, \langle S_2, S''_{i,2}\rangle, \ldots, \langle S_m, S''_{i,m}\rangle\}, \\
&\qquad\quad \mathcal{G}_i \in \psi(P_i)\}\}
\end{aligned}$$

**Figure 7: Definition of the generalization function $\Psi$**

The formula for $\Psi$ in Figure 7 invokes $\Psi'$ twice to compute the template AST forest before the change $\mathcal{T}_0$ and the template AST forest after the change $\mathcal{T}_1$. It then computes $B$ by invoking $\psi$ to obtain the generalized generators for AST sub-slices that match against each free template variable in $\mathcal{T}_1$. Figure 8 presents the definition of $\Psi'$. Intuitively, $\Psi'$ is the generalization function for template AST forests. $\Psi'(\mathbb{S}, M) = \langle \mathcal{T}, M'\rangle$ takes a list of AST slices $\mathbb{S}$ and an initial variable binding map $M$ and produces a generalized template AST forest $\mathcal{T}$ and an updated binding map $M'$.

The first two rows in Figure 8 correspond to the formulas for the cases of empty slices and slices with a single terminal, respectively. The two formulas simply create an empty template AST forest or a template AST forest with a single non-terminal node. The third row corresponds to the formula for the case of a single non-terminal. The formula recursively invokes $\Psi'$ on the list of children nodes of each slice and creates a new node with the non-terminal label in the result template AST forest as the root node.

The fourth and fifth rows correspond to the formulas for the cases where each slice is a single tree and the root nodes of the slice trees do not match. The fourth formula handles the case where there is an existing template variable in $M$ that can match the slice trees. The formula creates a template AST forest with the matching variable. The fifth formula handles the case where there is no existing template variable in $M$ that can match the slice trees. The formula creates a template AST forest with a new variable and updates the variable binding map to include the variable. For brevity, we omit three additional formulas for recursively handling AST forests.

## 4.4 Sampling Algorithm

Given a training database $D$, we could obtain an exponential number of transforms with the generalization function $\Psi$ described in Section 4.3, i.e., we can invoke $\Psi$ on any subset of $D$ to obtain a different set of transforms. Not all of the generalized transforms are useful. The goal of the sampling algorithm is to use the generalization function to systematically obtain a set of productive candidate transforms for the inference algorithm to consider.

Figure 1 presents the pseudo-code of our sampling algorithm. As a standard approach in other learning and inference algorithms to avoid overfitting, Genesis splits the training database into a training set $D$ and a validation set $E$. Genesis invokes the generalization functions only on pairs in the training set $D$ to obtain candidate transforms. Genesis uses the validation set $E$ to evaluate generalized transforms. $\mathbb{W}$ in Figure 1 is a work set that contains the candidate subset of $D$ that the sampling algorithm is considering to use to obtain generalized transforms. The algorithm runs five iterations. At each iteration, the algorithm first computes a fitness score for each candidate subset, keeps the top $\alpha$ candidate subsets, (we empirically set $\alpha$ to 1000 in our experiments) and eliminates the rest from $\mathbb{W}$ (lines 3-7). The algorithm then attempts to update $\mathbb{W}$ by augmenting each subset in $\mathbb{W}$ with one additional pair in $D$ (see lines 8-10). $\mathrm{fitness}(\mathbb{W}, \mathbb{S}, D, E)$ denotes the fitness score of the subset $\mathbb{S}$ based on the coverage and tractability of transforms generalized from $\mathbb{S}$.

## 4.5 Search Space Inference Algorithm

**ILP Formulation:** Given a set of candidate transforms $\mathbb{P}'$, the goal is to select a subset $\mathbb{P}$ from $\mathbb{P}'$ that successfully navigates the patch search space coverage vs. tractability tradeoff.

**Input** : a training set of pairs of AST slices $D$ and a set of pairs of AST slices $E$
**Output** : a set of transforms $\mathbb{P}'$

$\mathbb{W} \leftarrow \{\{\langle S, S'\rangle, \langle S'', S'''\rangle\} \mid \langle S, S'\rangle \in D, \langle S'', S'''\rangle \in D, \langle S, S'\rangle \neq \langle S'', S'''\rangle\}$
**for** $i = 1$ **to** 5 **do**
  $f \leftarrow \{\mathbb{S} \mapsto \text{fitness}(\mathbb{W}, \mathbb{S}, D, E) \mid \mathbb{S} \in \mathbb{W}\}$
  $\mathbb{W}' \leftarrow \{\mathbb{S} \mid \mathbb{S} \in \mathbb{W}, f(\mathbb{S}) > 0\}$
  Sort elements in $\mathbb{W}'$ based on $f$
  Select top $\alpha$ elements in $\mathbb{W}'$ with largest $f$ value as a new set $\mathbb{W}''$
  $\mathbb{W} \leftarrow \mathbb{W}''$
  **if** $i \neq 5$ **then**
    **for** $\mathbb{S}$ **in** $\mathbb{W}''$ **do**
      **for** $\langle S, S'\rangle$ **in** $D$ **do**
        $\mathbb{W} \leftarrow \mathbb{W} \cup \{\mathbb{S} \cup \langle S, S'\rangle\}$
$\mathbb{P}' \leftarrow \cup_{\mathbb{S} \in \mathbb{W}} \Psi(\mathbb{S})$
**return** $\mathbb{P}'$

**Algorithm 1:** Sampling algorithm sample$(D, E)$

$$\mathbb{P}' = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\} \qquad E = \{\langle S_1, S_1'\rangle, \ldots, \langle S_n, S_n'\rangle\}$$
$$C_{i,j} = |\{\text{str}(T) \mid \langle \mathcal{P}_j, S_i\rangle \Longrightarrow T\}|$$
$$G_{i,j} = \begin{cases} 1 & \langle \mathcal{P}_j, S_i\rangle \Longrightarrow_{\texttt{slice}} S_i' \\ 0 & \text{otherwise} \end{cases}$$

Variables: $x_i$, $y_i$ Maximize:$\eta \cdot \sum_{i=1}^{n_0} x_i + \sum_{i=n_0+1}^{n} x_i$ Satisfy:
$\forall i \in \{1, \ldots, n\} : \zeta - (\zeta - \beta) \cdot x_i - \sum_{j=1}^{k} C_{i,j} y_j \geq 0$
$\forall i \in \{1, \ldots, n\} : \sum_{j=1}^{k} G_{i,j} y_j - x_i \geq 0$
$\forall i \in \{1, \ldots, n\} : x_i \in \{0, 1\}$
$\forall i \in \{1, \ldots, k\} : y_i \in \{0, 1\}$
Result Transform Set: $\mathbb{P} = \{\mathcal{P}_i \mid y_i = 1\}$

**Figure 9: Integer linear programming formulas for selecting transforms given a set of candidate transforms $\mathbb{P}'$ and a validation set of AST slice pairs $E$**

Figure 9 presents our formulation of the transform selection problem as an integer linear program (ILP). $E$ is the set of training and validation patch pairs. In Figure 9, the first $n_0$ pairs are training patches (i.e. $\langle S_1, S_1'\rangle \ldots \langle S_{n_0}, S_{n_0}'\rangle$), the remaining pairs are validation patches. $C_{i,j}$ corresponds to the size of the search space derived from the $j$-th transform when applied to the $i$-th AST slice pair in $E$. $G_{i,j}$ indicates whether the space derived from the $j$-th transform contains the corresponding change for the $i$-th AST slice pair.

The variable $x_i$ indicates whether the result search space covers the $i$-th AST slice pair. The variable $y_i$ indicates whether the ILP solution selects the $i$-th transform. The ILP optimization goal is to maximize the weighted sum of $x$, where $\eta = 0.1$ is a parameter that controls the weight of covering training patches. The intuition is to prioritize the coverage of validation patches because the validation patches are hidden during the generalization step.

The first group of constraints is for tractability. The $i$-th constraint specifies that the derived final search space size (i.e. $\Sigma_{j=1}^{k} C_{i,j} y_j$), when applied to the $i$-th AST pair in $E$, should be less than $\beta$ if the space covers the $i$-th AST pair (i.e. $x_i = 1$) or less than $\zeta$ if the space does not cover the $i$-th AST pair (i.e. $x_i = 0$). We empirically set $\beta = 5 \times 10^4$ and $\zeta = 10^8$. The second group of constraints is for coverage. The $i$-th constraint specifies that if the final search space covers the $i$-th AST slice pair in $E$ (i.e. $x_i = 1$), then at least one of the selected transforms should cover the $i$-th pair.

**Inference Algorithm:** Starting from a training set of AST slice pairs $D$, Genesis first removes 25% of the AST slice pairs from $D$ to form the validation set $E$. It then runs the sampling algorithm to produce a set of candidate transforms $\mathbb{P}'$. It finally solves the above ILP with Gurobi [31], an off-the-shelf solver, to obtain the set of transforms $\mathbb{P}$ that generates the Genesis patch search space.

## 5 IMPLEMENTATION

We have implemented Genesis for Java programs. We use the spoon library [46] to parse Java programs. Our current implementation supports any Java application that operates with the maven project management system [5] and JUnit [19] testing framework.

Given a program $p$, a set of test cases, at least one of which exposes a defect in $p$, and an inferred search space $\mathbb{P}$, Genesis first uses a defect localization algorithm to identify a ranked list of suspicious locations (as AST snippets) in $p$ that are relevant to the defect. For each suspicious AST snippet $S$, Genesis applies each transform in $\mathbb{P}$ to generate candidate patches. It validates each candidate patch against the test cases and appends it to the generated patch list if it passes all test cases. Genesis is designed to work with arbitrary defect localization algorithms. Our current implementation derives the suspicious locations with stack traces obtained from test cases that trigger Java exceptions.

Genesis applies its inferred transforms to each of the suspicious statements in the ranked defect localization list. For each transform, Genesis computes a cost score which is the average number of candidate patches the transform needs to generate to cover a validation case. For each suspicious statement, Genesis prioritizes candidate patches that are generated by transforms with lower cost scores.

## 6 EXPERIMENTAL RESULTS

**Collect Training Patches and Errors:** We used a script to search the top 1000 github Java projects (ranked by number of stars), a list of 50968 github repositories from the MUSE corpus [3], and the github issue database for human patches for null pointer (NP), out of bounds (OOB), and class cast (CC) defects. We collected 1012 human patches from 372 different applications, including 503 NP patches, 212 OOB patches, and 303 CC patches.

**Training and Benchmark Patches:** From these collected patches, we identified benchmark patches that 1) have a JUnit [19] test suite in the repository that Genesis can run automatically, 2) the JUnit test suite contains at least one test case that can expose and reproduce the patched defect in our experimental environment, 3) the JUnit test suite contains at least 50 test cases in total, and 4) the test suite does not cause non-deterministic behavior. From the collected 1012 patches we identified 49 benchmark patches including 20 NP patches, 13 OOB patches, and 16 CC patches. We removed the 49 benchmark patches from the 1012 collected patches to obtain 963 training patches including 483 NP, 199 OOB, and 287 CC patches.

**Search Space Inference:** We ran the inference algorithm on the training patches for each class of defects to infer transforms for that class of defects. We also ran the inference algorithm on all of the training patches together to infer transforms for patching all

**Table 1: Genesis search space inference results.**

| | NP | OOB | CC | Combined |
|---|---|---|---|---|
| Sampled Transforms | 1354 | 978 | 769 | 577 |
| Selected Transforms | 52 | 35 | 28 | 108 |
| Inference Time | 1464m | 1181m | 861m | 2886m |

**Table 2: Genesis patch generation results.**

| Errors | Genesis First 1/5/10 (All) | Genesis (Combined) First 1/5/10 (All) | PAR First 1/5/10 (All) |
|---|---|---|---|
| 20 NP | 8/10/11 (11) | 8/11/11 (11) | 7/7/7 (7) |
| 13 OOB | 3/4/5 (6) | 5/5/5 (6) | 4/4/4 (4) |
| 16 CC | 3/3/4 (5) | 1/2/4 (4) | 0/0/0 (0) |
| Total | 14/17/20 (22) | 14/18/20 (21) | 11/11/11 (11) |

three classes of defects combined. Table 1 presents the results. The first row "Sampled Transforms" presents the number of candidate transforms produced by the sampling algorithm for each search space. The second row "Selected Transforms" presents the number of selected transforms in each inferred search space. We note that the number of selected transforms (tens for the targeted search spaces to over a hundred for the combined search space) is substantially larger than the number of manually generated transforms that previous search spaces work with [33–35, 37, 38, 48, 56, 61, 62]. This fact highlights the ability of the automated Genesis inference system to work effectively with large, detailed, and appropriately targeted sets of candidate transforms.

**PAR Template Implementation:** PAR [33, 44] deploys a set of transform templates to fix bugs in Java programs, with the templates manually derived by humans examining real-world patches. We implemented the PAR templates for NP, OOB, and CC defects under our own framework (most of the reported PAR patches are generated by the PAR NP and OOB templates [44]). To circumvent any ambiguities in the PAR template descriptions, we implemented the templates, with our best efforts, to enable the templates to generate correct patches for as many benchmark defects as possible.

**Patch Generation:** We ran Genesis on all of the benchmark defects with 1) the inferred transforms for patching each class of defects, 2) the inferred transforms for patching all three classes of defects combined, and 3) the PAR templates. For each defect, Genesis produces (a possibly empty) ranked list of validated patches. We performed all of the patch generation experiments on Amazon EC2 m4.xlarge instances with Intel Xeon E5-2676 processors, 4 vCPU, and 16 GB memory. We collected all patches that validated within 5 hours.

We next manually analyzed each benchmark defect along with the corresponding developer patch from the repository to understand the root cause of the defect. We then analyzed the ranked list of generated patches to identify the first correct patch in each list (if any). Note that in our experiments all of the generated patches that we identify as correct are semantically equivalent to the corresponding developer patch (and differ in at most error or log messages).

**Patch Generation Results:** Table 2 summarizes the patch generation results. The first column presents the number and type of each class of defect. The next columns present patch generation results for Genesis working with 1) the transforms for patching each class of defect, 2) the transforms for patching all three classes of defects combined, and 3) the PAR templates. Each entry is of the form X/Y/Z (W), where X is the number of defects for which the first patch to validate is correct, Y is the number of defects for which one of the first 5 patches to validate is correct, Z is the number of defects for which one of the first 10 patches to validate is correct, and W is the number of defects for which one of the validated patches is correct. Moving to the combined transforms drops only a single correct CC patch, highlighting the ability of Genesis to infer meaningful patches for multiple classes of defects from a single combined corpus of human patches. The PAR templates correctly patch roughly half as many defects as the Genesis transforms.

Consistent with previous results [39], there are many more validated patches than correct patches — for the combined search space, there are 8 NP defects, 4 OOB defects, and 1 CC defect with more than 20 validated patches. The maximum numbers of validated patches for a single NP, OOB, or CC defect are 62, 166, and 74 validated patches, respectively.

**Defect Localization Oracle:** To isolate the effect of defect localization, we also run Genesis and PAR with an oracle that identifies, for each defect, the correct line of code to patch. With the oracle, the combined Genesis space generates correct patches for two more NP defects (HikariCP [12] revision ce4ff92, first correct patch ranked 13th, and spring-data-rest [1] revision aa28aeb, first correct patch ranked 1st) and one more CC defect (jade4j [14] revision 114e886, first correct patch ranked 1st). PAR does not generate any additional correct patches with the oracle.

**Correct Patches:** In general, the correct patches either 1) apply a standard defect-specific patch pattern that Genesis successfully inferred, or 2) modify an existing condition in the unpatched program. Four of the NP patches insert a null pointer check that returns if the checked variable is null. Another three insert a null pointer check that conditionally executes existing code only if the checked variable is not null. Another throws an exception if the checked variable is null; yet another inserts a null pointer check that executes generated code when the check fires.

Three of the OOB patches insert a check for either a variable less than zero or an object field equal to zero, then return a generated value (either null or zero) if the check fires. One of the remaining OOB patches conditionally executes existing code if an object field is not zero. Two of the correct CC patches insert a try/catch statement that catches and ignores class cast exceptions. One of the remaining CC patches changes the declared type of a local variable.

The remaining NP, OOB, and CC patches change existing conditions in various ways. For example, one OOB patch corrects an off by one defect by changing < to <=; another changes a condition to compare an expression against a variable instead of against zero.

**PAR Comparison:** The PAR NP templates include "add an if statement to guard an existing statement" and "add an if guarded return statement" with null check boolean conditions. For the 11 NP defects for which Genesis generates correct patches, one NP defect (error-prone-370933 [9]) is outside the PAR search space because the correct patches add "if (...) throw ..." statements. Three more NP defects (DataflowJavaSDK-c06125 [7], javaslang-faf9ac [15], and Activiti-3d624a [4]) are outside the PAR search space because the correct patches change condition expressions in a non-trivial way that is not equivalent to adding an if-guard.

The PAR OOB templates include "add an if statement to guard an existing statement" and "add an if guarded return statement" with range check conditions. The templates also consider "increases or

decreases a variable by one" and "add an if guarded assignment statement to enforce index lower and upper bounds of a variable". For six OOB defects for which Genesis generates correct patches, two OOB cases (jgit-929862 [16] and jPOS-df400a [18]) are outside the PAR search space because the correct patches change conditions in a way different from the standard range checks.

The PAR CC templates include "add an if statement to guard an existing statement" and "add an if guarded return statement" with `instanceof` type checks. They also include "change the casting type of a cast operator". These templates do not generate correct patches for any of the benchmark CC defects. For two of these five defects (jade47 [14] revision 114e88 and HdrHistogram-030aac [11]), the correct patches change existing expressions in a way that is not equivalent to adding a type check guard. The correct patches for two more CC defects (htmlelements-bf3f27 [13] and hamcrest-bean-84586d [10]) insert a try-catch statement to catch and ignore class cast exceptions (the developers introduced these try-catch statements in the patched revision and these statements are still present in the latest revision of these repositories). The correct patch for the remaining defect (jade4j-dd4739 [14]) modifies the declared type of a local variable to avoid class cast exceptions.

**Discussion:** These results highlight how automating transform inference can produce more effective patch search spaces. Automating the inference makes it possible to work with larger sets of more detailed transforms that, working together, can more effectively navigate the inherent tradeoff between coverage and tractability. In comparison with manual transforms, the inferred transforms can be more specific (with focused generator parameters). But because there are so many more inferred transforms (over 100 for the combined training set) together they cover many more patch patterns and can successfully patch more defects. By automating transform inference and formulating the transform selection problem as an integer linear program, Genesis can effectively work with hundreds to thousands of candidate transforms to select tens of final transforms.

## 7 RELATED WORK

**Generate And Validate Systems:** GenProg [35, 62], AE [61], and RSRepair [48] use a variety of search algorithms (genetic programming, stochastic search, random search) in combination with transforms that delete, insert, and swap existing program statements. Kali [49] applies a single transform that simply deletes code. NOPOL [30] and ACS [63] apply transforms to fix errors in branch conditions. Prophet [37] and SPR [38] apply a set of predefined parameterized transformation schemas to generate candidate patches. Prophet processes a corpus of successful human patches to learn a model of correct code to rank plausible patches; SPR uses a set of hand-code heuristics for this purpose. Genesis differs in that it does not work with a fixed set of human-specified transforms. It instead automatically processes previous correct patches to infer a set of transforms that together define its patch search space.

**Constraint Solving Systems:** Prophet [37], SPR [38], Qlose [26], NOPOL [30], SemFix [45], and Angelix [40] all use constraint solving to generate new values for potentially defective expressions (often defective conditions). ClearView [47] enforces learned invariants to eliminate security vulnerabilities. Angelic Debugging [25] finds new values for potentially incorrect subexpressions that allow the program to produce correct outputs for test inputs. PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair PHP programs that generate HTML [56]. Specification-based data structure repair [28, 29, 32, 64] takes a data structure consistency specification and an inconsistent data structure, then synthesizes a repair that produces a modified data structure that satisfies the consistency specification. Genesis differs in that it works with automatically inferred transforms, with generators playing the role of constraint solvers to generate expressions that enable parameterized transforms to produce correct patches.

**Probabilistic Models for Programs:** There is a rich body of work on applying probabilistic models and machine learning learning techniques to programs, specifically for identifying correct repairs [37], code refactoring [51], and code completion [24, 50, 52]. These techniques learn a probabilistic model from a training set of patches or programs and then use the learned model to identify the best repair or token for a defective or partial program. Instead of learning individual patches, Genesis infers transforms that can be applied to generate multiple candidate patches. Genesis does not use probabilistic models. It instead obtains transforms with a novel patch generalization algorithm and formulates the transform selection problem as an integer linear program.

**Code Transfer:** CodePhage [57], $\mu$Scalpel [23], and CodeCarbonCopy [58] transfer code across applications. While Genesis also leverages code from multiple applications, it infers transforms from previous successful patches instead of transferring code.

**Edit, Template, and Transformation Extraction:** SYDIT [42] and Lase [43] extract edit scripts from one (SYDIT) or more (Lase) example edits. The script is a sequential list of modification operations that insert statements or update existing statements. SYDIT and Lase then generate changes to other code snippets in the same application with the goal of automating repetitive edits. RASE [41] uses Lase edit scripts to refactor code clones. FixMeUp [59, 60] extracts and applies access control templates to protect sensitive operations. REFAZER (first published [54] after the first publication of this research [36]), implements an algorithm for learning syntactic program transformations from examples [55]. The REFAZER transformations were used to perform repetitive edits on large code bases and to correct defects in student submissions, and were mostly not useful across assignments. Genesis differs in that it processes patches from multiple applications to derive generalized application-independent transforms that it can apply to fix bugs in yet other (previously unseen) applications. The Genesis transforms also include generators that enable transforms to generate new code (as opposed to simply reusing existing matched code).

## 8 CONCLUSION

Previous generate and validate patch generation systems work with a fixed set of transforms defined by their human developers. By automatically inferring transforms from successful human patches, Genesis makes it possible to leverage the combined expertise and patch generation strategies of developers worldwide to automatically patch bugs in new applications.

**Table 3: Genesis results with the combined search space.**

| Type | Repository | Revision | Init. Time | Search Space Size | Explored Space Size | Search Time | Validated Patches | First Correct Patch | | | PAR Result |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Generation Time | Validated Rank | Space Rank | |
| NP | caelum-stella | 2ec5459 | 2m | 62433 | 62433 | 76m | 25 | <1m | 1 | 147 | Correct |
| NP | caelum-stella | 2d2dd9c | 1m | 33198 | 33198 | 47m | 28 | <1m | 1 | 553 | Correct |
| NP | caelum-stella | e73113f | <1m | 33592 | 33592 | 48m | 28 | <1m | 1 | 528 | Correct |
| NP | HikariCP | ce4ff92 | 3m | 163998 | 79985 | >5h | 33 | - | - | - | No |
| NP | nutz | 80e85d0 | 2m | 675603 | 245793 | >5h | 0 | - | - | - | No |
| NP | spring-data-rest | aa28aeb | 7m | 153943 | 18773 | >5h | 0 | - | - | - | No |
| NP | checkstyle | 8381754 | 3m | 592851 | 110058 | >5h | 29 | 12m | 3 | 3261 | Correct |
| NP | checkstyle | 536bc20 | 2m | 839914 | 119964 | >5h | 49 | <1m | 1 | 26 | Correct |
| NP | checkstyle | aaf606e | 2m | 681420 | 117819 | >5h | 0 | - | - | - | No |
| NP | checkstyle | aa829d4 | 1m | 0 | 0 | <1m | 0 | - | - | - | No |
| NP | jongo | f46f658 | 1m | 325561 | 41504 | >5h | 0 | - | - | - | No |
| NP | DataflowJavaSDK | c06125d | 3m | 86731 | 78301 | >5h | 1 | 10m | 1 | 4653 | No |
| NP | webmagic | ff2f588 | 1m | 184724 | 115693 | >5h | 0 | - | - | - | No |
| NP | javapoet | 70b38e5 | <1m | 280469 | 136400 | >5h | 0 | - | - | - | No |
| NP | closure-compiler | 9828574 | 4m | $> 10^6$ | 31940 | >5h | 5 | 16m | 1 | 14 | Correct |
| NP | truth | 99b314e | <1m | 84076 | 84076 | 56m | 0 | - | - | - | No |
| NP | error-prone | 3709338 | 2m | 665832 | 3350 | >5h | 9 | 2m | 1 | 473 | No |
| NP | javaslang | faf9ac2 | <1m | $> 10^6$ | 392392 | >5h | 18 | 2m | 2 | 12242 | No |
| NP | Activiti | 3d624a5 | 4m | 462310 | 2142 | >5h | 62 | 3m | 4 | 31 | No |
| NP | spring-hateoas | 48749e7 | <1m | 25633 | 25633 | 38m | 43 | <1m | 1 | 268 | Correct |
| OOB | Bukkit | a91c4c6 | <1m | 430352 | 118319 | >5h | 4 | 2m | 1 | 728 | Correct |
| OOB | RoaringBitmap | 29c6d59 | 4m | 537740 | 70293 | >5h | 0 | - | - | - | No |
| OOB | commons-lang | 52b46e7 | 2m | 136402 | 8347 | >5h | 0 | - | - | - | No |
| OOB | HdrHistogram | db18018 | <1m | 344483 | 134113 | >5h | 140 | - | - | - | No |
| OOB | spring-hateoas | 29b4334 | <1m | 37105 | 37105 | 41m | 0 | - | - | - | No |
| OOB | wicket | b708e2b | 7m | 233586 | 102339 | >5h | 29 | 160m | 12 | 46506 | Correct |
| OOB | coveralls-maven-plugin | 20490f6 | <1m | 7298 | 7298 | 6m | 0 | - | - | - | No |
| OOB | named-regexp | 82bdfeb | <1m | 0 | 0 | <1m | 0 | - | - | - | No |
| OOB | jgit | 929862f | 3m | 140077 | 140077 | 193m | 3 | 84m | 1 | 54732 | No |
| OOB | jPOS | df400ac | 3m | 222560 | 222560 | 299m | 17 | 26m | 1 | 18022 | No |
| OOB | httpcore | dd00a9e | 2m | 300612 | 20984 | >5h | 166 | 1m | 1 | 427 | Correct |
| OOB | vectorz | 2291d0d | <1m | 184636 | 184636 | 268m | 32 | <1m | 1 | 2 | Correct |
| OOB | maven-shared | 77937e1 | 2m | 0 | 0 | <1m | 0 | - | - | - | No |
| CC | jade4j | dd47397 | <1m | 239966 | 120217 | >5h | 1 | 169m | 1 | 65325 | No |
| CC | jade4j | 114e886 | <1m | 437323 | 87192 | >5h | 0 | - | - | - | No |
| CC | HdrHistogram | 030aac1 | <1m | 49997 | 49997 | 166m | 74 | 2m | 7 | 2662 | No |
| CC | pdfbox | 93c0b69 | 1m | 208714 | 142352 | >5h | 0 | - | - | - | No |
| CC | tree-root | fef0f36 | <1m | 43785 | 43785 | 39m | 0 | - | - | - | No |
| CC | spoon | 48d3126 | 9m | 0 | 0 | <1m | 0 | - | - | - | No |
| CC | pebble | 942aa6e | 2m | 255076 | 103554 | >5h | 0 | - | - | - | No |
| CC | fastjson | c886874 | 1m | $> 10^6$ | 258758 | >5h | 0 | - | - | - | No |
| CC | htmlelements | bf3f275 | 1m | 164348 | 144123 | >5h | 12 | 36m | 7 | 16270 | No |
| CC | spring-cloud-connectors | 56c6eca | 1m | 151568 | 151568 | 244m | 5 | - | - | - | No |
| CC | joinmo | a5ee885 | 1m | 689426 | 158816 | >5h | 4 | - | - | - | No |
| CC | buildergenerator | d9d73b3 | <1m | 58520 | 58520 | 84m | 0 | - | - | - | No |
| CC | mybatis-3 | 809c35d | 8m | 1184696 | 136625 | >5h | 0 | - | - | - | No |
| CC | antlr4 | 9e7b131 | 3m | 5917 | 5917 | 48m | 1 | - | - | - | No |
| CC | hamcrest-bean | 84586d9 | <1m | 1081802 | 148401 | >5h | 4 | 30m | 2 | 12029 | No |
| CC | raml-java-parser | 49aab8f | <1m | 87054 | 87054 | 162m | 0 | - | - | - | No |

## 9 REPLICATION PACKAGE

All experimental data (including the Genesis source code, inferred templates, and generated patches) are available at http://groups.csail.mit.edu/pac/patchgen/. Table 3 shows the results from the replication package using the inferred combined search space. There is a row in the table for each defect. The "Type", "Repository", and "Revision" columns present the defect type, the Github repository name, and the revision, respectively. The "Init. Time" column presents the amount of time required to initialize the search for that defect. The "Search Space Size" column presents the size of the search space for that defect, the "Explored Space Size" column presents the size of the search space that the algorithm explores within the five hour timeout, the "Search Time" column presents the amount of time spent exploring the space, and "Validated Patches" presents the number of candidate patches that validate (produce correct outputs for all test cases). The next three columns present statistics for the first generated correct patch, specifically how long it takes to generate the patch ("Generation Time"), the rank of the first correct patch in the sequence of validated patches ("Validated Rank"), and the rank of the correct patch in the sequence of candidate patches ("Space Rank"). The last column ("PAR Result") presents whether PAR can generate correct patches for the defect.

## ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments on earlier drafts of the paper. This research was supported by DARPA (Grant FA8750-14-2-0242).

## REFERENCES


[1] Spring Data REST. http://projects.spring.io/spring-data-rest/. (????).
[2] GitHub. https://github.com/. (2008).
[3] Mining and Understanding Software Enclaves (MUSE) Program. https://wiki.museprogram.org. (2016).
[4] Activiti. http://activiti.org/. (2017).
[5] Apache maven. https://maven.apache.org/. (2017).
[6] Bukkit. https://bukkit.org. (2017).
[7] Dataflow Java SDK. https://github.com/GoogleCloudPlatform/DataflowJavaSDK. (2017).
[8] dyn.js. http://dynjs.org/. (2017).
[9] Error Prone. http://errorprone.info/. (2017).
[10] Hamcrest Bean. https://github.com/eXparity/hamcrest-bean. (2017).
[11] HdrHistogram. https://github.com/HdrHistogram/HdrHistogram. (2017).
[12] HikariCP. https://brettwooldridge.github.io/HikariCP/. (2017).
[13] Html Elements framework. https://github.com/yandex-qatools/htmlelements. (2017).
[14] jade4j. https://github.com/neuland/jade4j. (2017).
[15] Javaslang. http://www.javaslang.io/. (2017).
[16] JGit - Eclipse. https://eclipse.org/jgit/. (2017).
[17] Joda-Time. http://www.joda.org/joda-time/. (2017).
[18] jPOS. http://www.jpos.org/. (2017).
[19] JUnit. http://junit.org/. (2017).
[20] MapStruct - Java Bean Mappings, the Easy Way! http://mapstruct.org/. (2017).
[21] OrientDB. http://orientdb.com/orientdb/. (2017).
[22] Simple, Intelligent, Object Mapping. http://modelmapper.org/. (2017).
[23] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 257–269.
[24] Pavol Bielik, Veselin Vechev, and Martin Vechev. 2016. PHOG: Prababilistic Model for Code. In *Proceedings of the 33rd International Conference on Machine Learning*.
[25] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11')*. ACM, New York, NY, USA, 121–130. DOI: http://dx.doi.org/10.1145/1985793.1985811
[26] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer-Aided Verification (CAV)*.
[27] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy if Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*. ACM, New York, NY, USA, 30–39. DOI: http://dx.doi.org/10.1145/2593735.2593740
[28] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. 2006. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. 233–244.
[29] Brian Demsky and Martin C. Rinard. 2006. Goal-Directed Reasoning for Specification-Based Data Structure Repair. *IEEE Trans. Software Eng.* 32, 12 (2006), 931–951.
[30] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. *CoRR* abs/1505.07002 (2015). http://arxiv.org/abs/1505.07002
[31] Inc. Gurobi Optimization. Gurobi Optimizer Reference Manual. (2015). http://www.gurobi.com
[32] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. 2005. Repairing Structurally Complex Data. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*. 123–138.
[33] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13')*. IEEE Press, 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893
[34] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*. 213–224.
[35] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, 3–13. http://dl.acm.org/citation.cfm?id=2337223.2337225
[36] Fan Long, Peter Amidon, and Martin Rinard. 2016. *Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems.* Technical Report MIT-CSAIL-TR-2016-010. http://hdl.handle.net/1721.1/103556
[37] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 298–312.
[38] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178. DOI: http://dx.doi.org/10.1145/2786805.2786811
[39] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 702–713.
[40] 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, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 691–701.
[41] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. 2015. Does Automated Refactoring Obviate Systematic Editing?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15')*. IEEE Press, Piscataway, NJ, USA, 392–402. http://dl.acm.org/citation.cfm?id=2818754.2818804
[42] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11')*. ACM, New York, NY, USA, 329–342. DOI: http://dx.doi.org/10.1145/1993498.1993537
[43] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 502–511.
[44] Martin Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 234–242. DOI: http://dx.doi.org/10.1145/2568225.2568324
[45] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13')*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890
[46] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* (2015), na. DOI: http://dx.doi.org/10.1002/spe.2346
[47] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*. ACM, 87–102. DOI: http://dx.doi.org/10.1145/1629575.1629585
[48] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 254–265. DOI: http://dx.doi.org/10.1145/2568225.2568254
[49] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Anlysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2015*.
[50] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16')*. ACM, New York, NY, USA, 761–774. DOI: http://dx.doi.org/10.1145/2837614.2837671
[51] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15')*. ACM, New York, NY, USA, 111–124. DOI: http://dx.doi.org/10.1145/2676726.2677009
[52] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14')*. ACM, New York, NY, USA, 419–428. DOI: http://dx.doi.org/10.1145/2594291.2594321
[53] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*. 303–316.
[54] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjoern Hartmann. Learning Syntactic Program Transformations from Examples. (Aug. 2016). arXiv:arXiv:1608.09000
[55] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*.
[56] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 277–287.

[57] Stelios Sidiroglou, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Multi-Application Code Transfer. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

[58] Stelios Sidiroglou-Douskos, Eric Lantinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.

[59] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2011. RoleCast: finding missing security checks when you do not know what checks are. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 1069–1084.

[60] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications.. In *NDSS*.

[61] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE'13*. 356–366.

[62] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09')*. IEEE Computer Society, 364–374. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070536

[63] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426. DOI: http://dx.doi.org/10.1109/ICSE.2017.45

[64] Razieh Nokhbeh Zaeem, Muhammad Zubair Malik, and Sarfraz Khurshid. 2013. Repair Abstractions for More Efficient Data Structure Repair. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. 235–250.