L2S: a Framework for Synthesizing the Most Probable Program under a Specification

YINGFEI XIONG and BO WANG, Peking University

In many scenarios, we need to find the most likely program that meets a specification under a local context, where the local context can be an incomplete program, a partial specification, natural language description, etc. We call such a problem *program estimations*. In this paper, we propose a framework, *LingLong Synthesis Framework*, or *L2S* in short, to address this problem. Compared with existing work, our work is novel in the following aspects. (1) We propose a theory of expansion rules to describe how to decompose a program into choices. (2) We propose an approach based on abstract interpretation to efficiently prune off the program sub-space that does not satisfy the specification. (3) We prove that the probability of a program is the product of the probabilities of choosing expansion rules, regardless of the choosing order. (4) We reduce the program estimation problem to a path finding problem, enabling existing path-finding algorithms to solve this problem.

L2S has been applied to program generation and program repair. In this paper, we report our instantiation of this framework for synthesizing conditional expressions (L2S-Cond) and repairing conditional statements (L2S-Hanabi). The experiments on L2S-Cond show that each option enabled by L2S, including the expansion rules, the pruning technique, and the use of different path-finding algorithms, plays a major role in the performance of the approach. The experiments on L2S-Hanabi show that it outperforms ACS, a state-of-the-art APR system for repairing conditional statements, on the Defects4J benchmark. L2S-Hanabi increases the number of repaired bugs from 17 to 28 and the precision from 74% to 85%. The performance of L2S-Hanabi on Bugs.jar is consistent with that on Defects4J.

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

Additional Key Words and Phrases: Program Estimation, Program Synthesis, Program Repair, Expansion Rules

ACM Reference Format:
Yingfei Xiong and Bo Wang. 2021. L2S: a Framework for Synthesizing the Most Probable Program under a Specification. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 47 pages. https://doi.org/10.1145/3487570

1 INTRODUCTION

In many tasks, we need to synthesize a program automatically to solve a problem. Some problems have clearly defined specifications: when the program meets the specification, the program is considered correct. Traditional program synthesis techniques are mainly designed to deal with such problems [20]. However, many problems do not have such a complete specification. For example, in test-based program repair [63] and program by examples [18], only a set of tests is available to validate the correctness of the patched program. Other related fields are code completion [46] and program synthesis from natural languages [51], where code is generated based on a partial

Authors' address: Yingfei Xiong, xiongyf@pku.edu.cn; Bo Wang, wangbo_15@pku.edu.cn, Key Lab of High Confidence Software Technologies, Ministry of Education; Department of Computer Science and Technology, EECS, Peking University, Haidian, Beijing, 100871.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2021/1-ART1 $15.00
https://doi.org/10.1145/3487570

program and/or natural language specifications. In all the above cases, it is not enough to generate a program that satisfies the specification. For the former two scenarios, existing studies [18, 40, 49] have revealed repairing only for passing the tests often results in incorrect patches. In the latter two scenarios, there is not even a partial formal specification, and returning an arbitrary compilable program is definitely not desirable.

A more desirable solution to these cases, as we argue in this paper, is to find the program that is most likely to be written under the current context. More formally, given a space of programs $\mathtt{Prog}$, a specification $\mathtt{spec}$, and a context $\mathtt{context}$, we would like to find a program $\mathtt{prog}$ such that $\mathtt{prog} \in \mathtt{Prog}$ satisfies $\mathtt{spec}$, i.e., $\mathtt{prog} \vdash \mathtt{spec}$, and maximizes the conditional probability $\mathtt{Pr}(\mathtt{prog} \mid \mathtt{context})$. In other words, we need to calculate the following formula.

$$\arg\max_{\mathtt{prog}\in\mathtt{Prog}\ \wedge\ \mathtt{prog}\vdash\mathtt{spec}} \mathtt{Pr}(\mathtt{prog} \mid \mathtt{context})$$

The program space can be specified by grammar rules, the specification can be a test suite or a logic specification, and the context can be the specification, an incomplete program, and/or natural language description, etc. To distinguish from traditional program synthesis where a precise specification exists, we call this problem *program estimation.*

Solving the program estimation problem is not easy, and involves three subproblems: (1) calculate the conditional probability $\mathtt{Pr}(\mathtt{prog} \mid \mathtt{context})$, (2) locate the program with the maximum probability, and (3) ensure that the located program satisfies the specification. Since the program estimation is difficult to be solved precisely, the first two subproblems are usually approached with approximation: the conditional probability is estimated from a training set of context-program pairs $\mathtt{Data}$, and the algorithm finds a program whose probability is as large as possible.

Existing studies [5, 19, 57, 72] are mainly conducted from the machine learning perspective. These approaches decompose a program into a set of choices, where for each choice the number of options is small and can be handled by a learned classifier. For example, in an application that generates a sequence of API calls [57], first, a model is used to estimate how much likely each API method is called next, and then another model is used to estimate how much likely each variable is used as an argument of the method. Afterwards, the probability of the whole program is the product of each single choice. Finally, a beam search algorithm [44] is used to find the best combination of all the choices.

However, the current approaches still have multiple limitations.

- First, most existing approaches are designed case by case, and it is not clear what possibilities exist to decompose a program into a set of choices in general.
- Second, the existing approaches usually only ensure the generated program to satisfy certain structural properties, such as syntactic correctness, but it is not clear how to satisfy more complex specifications, such as type correctness and functional correctness.
- Third, existing approaches directly assume the probability of the program is the product of the probabilities of the choices [5], and it is not clear under what conditions the assumption holds.
- Fourth, most existing approaches use beam search to find the program with the maximum probability, and it is not clear what other search algorithms can be used.

In this paper, we propose a framework, *LingLong Synthesis Framework*[1], or *L2S* in short to address the program estimation problem. L2S is a framework generalizing existing approaches for program estimation, and we propose the following novel components to address the above limitations.

[1]LingLong is a Chinese word meaning flexible and delicate

- To address the first limitation, we propose a theory of expansion rules. Expansion rules generalize AST rules, and allow describing different ways of generating a program. L2S is built upon expansion rules to describe different ways of decomposing a program into choices.
- To address the second limitation, we introduce an approach based on abstract interpretation to efficiently prune off the program subspaces that cannot satisfy the specification early on.
- To address the third limitation, we show that the probability of the program is the product of the probabilities of choosing the expansion rule at each expansion position (explained later), and is not related to the order of choosing expansion positions, which allows an efficient calculation of program probability from the probabilities of choosing the rules.
- To address the fourth limitation, we show that the program estimation problem whose choices are decomposed following expansion rules can be viewed as a path-finding problem, where the vertexes are partial and complete programs and the weights on paths are the probabilities of the programs, allowing to employ any algorithm for path finding problem to find the program with the maximum probability.

L2S has already been applied in code generation: Sun et al. [59, 60] instantiated the L2S framework to generate programs from natural language description, which significantly outperformed existing work. In this paper, we give a complete introduction to the L2S framework and report our newest instantiation of L2S to automated program repair [63].

Our instantiation is designed for repairing buggy conditional expressions, where we synthesize a new conditional expression based on the context (the surrounding code) to replace the buggy one. More concretely, we first build a tool, L2S-Cond, for synthesizing Java conditional expressions by viewing the surrounding code as the context and the typing rules as the specification. Then we build a repair tool, L2S-Hanabi[2], by first localizing a buggy conditional expression and then synthesizing a new conditional expression by L2S-Cond to replace the buggy one, following the framework of ACS, an existing approach for repairing conditional expressions [68].

We first evaluate different ways of instantiating L2S to form L2S-Cond. The results suggest that the options enabled by L2S, including different sets of expansion rules, different machine learning techniques, different path-finding algorithms, and whether to prune off infeasible program subspaces, play a major role in the performance of L2S-Cond. The best set of options correctly predict 61.7% of the conditional expression in the top 5 candidates, and 75.1% in the top 200 candidates.

Next, we evaluate L2S-Hanabi on 272 real-world bugs from six diverse projects in two benchmarks, i.e., Defects4J [29] and Bugs.jar [54]. L2S-Hanabi correctly repairs 32 defects, with a precision of 76.2%, outperforming ACS [68], the state-of-the-art approach for repairing conditions with 64.7% more bugs repaired and slightly higher precision.

Our implementation and experimental data can be found at https://wangbo15.github.io/LingLong/.

In summary, this paper presents the following main contributions.

- We present a novel theory of expansion rules, which is a generalization of grammar rules to describe different ways of generating a program.
- We present a framework, L2S, that converts the program estimation problem into a path finding problem on graphs, allowing us to predict the probability of a program along an expansion path and use different path-finding algorithms to solve this problem.
- We propose an approach based on abstract interpretation to efficiently prune off the program subspaces that cannot satisfy the specification early on.
- We instantiate the L2S framework for repairing buggy conditional expressions by systematically exploring different sets of expansion rules and proposing a novel encoding method

[2]Hababi is the Japanese word for fireworks, indicating the tool follows a bottom-up fashion which will be illustrated later.

for identifiers. The evaluation shows that L2S-Hanabi significantly outperforms existing approaches.

This paper is a significantly extended version of a previous workshop paper [67][3]. First, the L2S framework has been completely reworked, with a novel theory of expansion rules, a novel approach based on abstract interpretation to prune off the space, an updated theorem of the probability, and a novel characterization of the program as a path-finding problem. Second, the application to APR has been improved to include a more sophisticated way to train the machine learning models and systematic exploration of the design space of L2S-Cond. Third, the evaluation has been extended to include a comprehensive evaluation on two real-world bug benchmarks, as well as detailed analysis and discussion about the synthesized conditions in the repair experiments. Fourth, the paper has been largely rewritten to improve the presentation, including a set of completely re-designed formal notations and definitions.

In the remainder of the paper, we will introduce L2S step by step. We first give the approach overview with a motivating example (Section 2), and then introduce L2S in detail (Section 3). Then we instantiate the framework as L2S-Cond for generating Java conditional expressions (Section 4). Based on L2S-Cond, we build our tool L2S-Hanabi for repairing conditional statements and evaluate L2S-Hanabi on 272 real-world bugs (Section 5). Finally, we discuss related work (Section 6), analyze the threats to validity (Section 7), and conclude the paper (Section 8).

2 MOTIVATING EXAMPLE AND APPROACH OVERVIEW

In this section we give an overview of L2S with a motivating example. After the introduction of the motivating example (Section 2.1), we shall first introduce L2S with respect to the three subproblems: (1) calculating the conditional probability of a program (Section 2.2), (2) generating the program with maximum probability (Section 2.3), and (3) ensuring that the program satisfies the specification (Section 2.4). We shall first consider a basic decomposition method that follows the grammar (Section 2.2-2.1), and then introduce other potential decompositions based on the theory of expansion rules (Section 2.5).

2.1 Motivating Example

Conditional expressions are a common source of bugs: an existing study [58] has shown that 43% of real-world bugs are related to buggy `if` conditions. As a result, several existing program repair approaches [39, 68, 70] focus on repairing buggy conditions. A typical approach for condition repair first localizes a buggy conditional expression using fault localization approaches [28, 73], and then synthesizes a new condition to replace the buggy one. Therefore, it is critical to synthesize the condition correctly.

Synthesizing a condition is a typical program estimation problem. The context `context` is the project source code surrounding the target condition. The specification `spec` is the syntactic and type constraint of the programming language and the tests in the program, i.e., the synthesized condition must be syntactically and type correct, and passes the tests in the program. The training set `Data` includes pairs of conditional expressions and their context, which can be easily collected from existing open source projects. Finally, there is a characterization of the program space `Prog`. L2S follows existing program synthesis approaches [3] to use grammars to define the program space. A simplified example grammar for conditional expressions is given below, where T is the root symbol.

[3]L2S was known as *Learning to Synthesize* in the workshop paper.

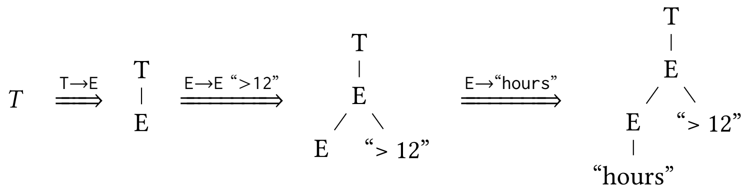


Fig. 1. Decomposing program into choices

$$\mathtt{T} \rightarrow E$$
$$E \rightarrow E \text{ " } > 12\text{" } \mid E \text{ " } > 0\text{" } \mid E \text{ " } > \text{ " } E \mid E \text{ " } + \text{ " } E \mid \text{"}\textit{hours}\text{"} \mid \text{"}\textit{value}\text{"}$$

2.2 Calculating the conditional probability

A standard way of decomposing a program into choices is to follow its top-down expansion order defined by the grammar [50, 72]. For example, an expression "`hours > 12`" can be considered as the result of the three choices shown in Figure 1. The first choice chooses among all grammar rules starting with `T` to expand the root symbol, and in this case we have only one option. The second choice chooses among all grammar rules starting with `E` to expand the node E. The third choice again chooses among all grammar rules starting with `E` to expand the newly introduced node E. After decomposing a program into choices, the probability of the program is the product of the probabilities of all choices.

In L2S, the probabilities of the choices are specified by a probabilistic model. The conditional probability has the form `Pr(rule|context, prog, position)`, where `context` represents the context for the program generation, `prog` represents the current program that has been generated, `position` represents the position of the node to be expanded, and `rule` represents the choice of a rule starting from the symbol at `position`. Such conditional probabilities could be predicted by machine learning models trained on the a training set. When we have the conditional probability of each choice, the probability of the whole program is the product of the probabilities of all choices.

L2S does not enforce any concrete machine learning methods, and users could choose the methods that fit best the problem. Furthermore, different machine learning methods could be specified for different non-terminals for best results. Since the input only provides the set of programs and their context, we need to parse the programs to reproduce the choices. At each choice, the chosen option is the positive instance and all other options are negative instances.

In practice, we use two ways to train the machine learning models. For the non-terminals whose expansions are common at different contexts, such as `Statement` → `IfStatement` | `WhileStatement` | `BasicStatement`, we train a classifier for the multi-class problem, where each class corresponds to a choice. For the non-terminals whose expansions may change at different contexts, such as `Variable` → "`hours`" | "`values`", we train a binary classifier whose output is the probability of the current rule.

However, the above example is special as in each step we have only one non-terminal node that needs to be expanded. In general cases, there may be more than one non-terminal node that can be extended. As a result, the choices involve not only which grammar rule to use but also which non-terminal node to expand. For example, Figure 2 shows an expansion process that involves four choices. The first partial program has two non-terminal nodes that need to be expanded, E_1 and E_2.

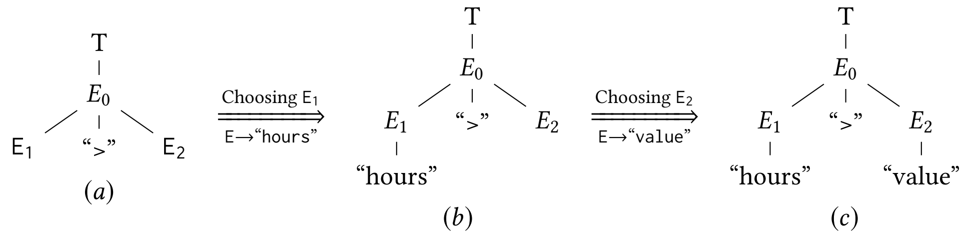


Fig. 2. Choices including both grammar rules and non-terminals

The first choice chooses node E_1, the second choice chooses grammar rule $E \longrightarrow$ "*hours*", the third choice chooses node E_2, and the last choice chooses grammar rule $E \longrightarrow$ "*value*".

The existence of choices on non-terminal nodes gives us a question: is the probability of the program still equal to the product of the probabilities of all choices? If so, how can we compute the probabilities of the node selections? Later in this paper we will show that, even if the choices of nodes are involved, the probability of the program is equal to the product of only the probabilities of choosing each grammar rule. In other words, the probabilities of choosing non-terminal nodes are omitted. If we choose to expand E_2 first and E_1 second, we still should get the same probability. This property gives us the freedom of setting up a policy to choose the non-terminal node to be expanded first so as to best fit the machine learning models.

2.3 Generating the Program with Maximum Probability

In the previous section, we have described how to estimate the probability of a single program. Now we introduce how to generate the program that satisfies the specification and has the maximum probability, a combinatorial optimization problem.

L2S views this optimization problem as a path finding problem on directed graphs. Here the vertexes of the graph are partial or complete programs, and the edges are labelled with a non-terminal node in the program and a grammar rule. A program p connects to another program p' via an edge labelled with non-terminal node v and grammar rule g if applying g to p to expand v produces p'. For example, Figure 2 shows a path on such a graph.

The gain of an edge is the probability of choosing the grammar rule as predicted by the machine learning models. The gain of a path is the product of the gain of the edges in the path. The start vertex is the empty program. A goal vertex is a complete program that satisfies the specification. Our task is to find a path from the start vertex to a goal vertex where the gain of the path is as high as possible.

L2S does not enforce an algorithm for solving the path finding problem, and the application could choose the path-finding algorithm that best fits the problem domain. For example, the classic Dijkstra's algorithm can be used. If we can find a proper heuristic function, the A* algorithm [22] can also be used. Approximation algorithms, such as beam search algorithm [44] and Monte Carlo tree search [6], could also be used.

2.4 Ensuring the satisfaction of the specification

The path finding problem requires that a goal vertex, or a goal program, satisfies the specification. A basic method to implement it is that, every time the path-finding algorithm locates a complete program, we check whether this program satisfies the specification or not. If it satisfies, we reach a goal, otherwise we continue the search.

While this method is simple to implement, it is not very efficient. It is often the case that a partial program can never be expanded into a goal program, but the path-finding algorithm still keeps expanding it. It would be desirable if we can determine whether a partial program can lead to a goal program, and prune off those that cannot early.

Determining whether a partial program leads to a goal program is not trivial, because the partial programs contain non-terminal nodes that have not been expanded. For example, let us suppose the specification is an input-output example: the output of the conditional expression should be `"true"` when `hours = 1` and `value = 10`. It is easy to see that the program in Figure 2(c) does not satisfy the example, but it is not easy to know whether the program in Figure 2(b) can lead to a complete program that satisfies the example or not, as there is an unexpanded node E_2.

To solve this problem, L2S introduces an offline static analysis procedure over the grammar rules. By using a proper abstract domain following abstract interpretation [12], we can find an upper bound of the values that a non-terminal can take in all possibly expansions, and if the upper bound of root node does not contain the expected output, we know that the current program can never lead to a goal program. For the running example, we can use intervals as the abstract domain for integers. By performing a static analysis over the grammars, we can find out when E is used as an integer, its value falls into the interval $[1, +\infty]$. This is because only addition is allowed in the grammar and both `hours` and `value` are larger than or equal to 1. Also we know that the interval for `hours` is $[1, 1]$, then $[1, 1] > [1, +\infty]$ only leads to `false`, and thus we know that the partial program can never lead to a goal program.

Please note that determining whether a partial program can lead to a goal program also takes time, and thus it is important to design an efficient abstract domain such that the benefit outweighs the cost.

2.5 Other Expansion Orders

The above method of decomposing a program into choices follows the top-down expansion orders of the grammar. L2S also allows decomposing a program into choices following other expansion order of the grammar. For example, given a conditional expression "`hours > 12`", we may assume the expansion starts from the leftmost leaf node by choosing which variable to be tested in the conditional expression among all variables. After we choose "`hours`", we further choose its parent node, i.e., what test should be applied on variable "`hours`", and the choice is "`> 12`".

Allowing different ways of decomposing a program is important because the search algorithms for solving the path-finding problem may be greatly affected by the set of choices. For example, in the application of synthesizing a conditional expression, it is often easy to predict which variable would be used in the conditional expression, rather than predicting which tests would be performed. With a properly trained machine learning model, we may often predict one variable, e.g., "`hours`", have a much higher probability to be tested than other variables, e.g., this variable has just assigned the return value of a method whose return value needs to be checked. If we start from the choice of variables, search algorithms such as Dijkstra's algorithm and A* search could focus their searches on the highly probable variables and efficiently explore the search space. Furthermore, machine learning models may also exhibit different accuracies at different sets of choices.

L2S supports different sets of choices by generalizing grammar rules into expansion rules. Given a grammar rule, we give indexes to the non-terminals in the rule, from left to right and starting from zero. For example, $\mathtt{N} \rightarrow \mathtt{N}$ " + " $\mathtt{N}$ is numbered as $\mathtt{N}^0 \rightarrow \mathtt{N}^1$ " + " $\mathtt{N}^2$, where the superscripts indicate the indexes of the non-terminals. An expansion rule takes the form of $\langle \mathtt{rule}, \mathtt{i} \rangle$, indicating when the ith non-terminal is presented, the rule can be applied to generate all other symbols in the rule. When i is τ, the rule can be applied to an empty tree to generate the first set of symbols. For

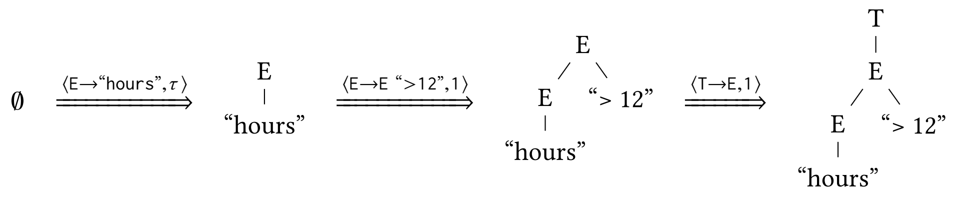


Fig. 3. Decomposing program with bottom-up expansion order

example, Figure 3 shows how to expand "`hours > 12`" from the leftmost leaf using two expansion rules.

By using a different set of expansion rules, we can control how a program is decomposed into choices. For example, if we would like to always expand a program in the top-down order, as in the original grammar, we can produce a set of expansion rules by appending 0 to each original grammar rule. If we would like to expand a program from a leaf node as in the above example, we can use the following expansion rules.

$$\begin{array}{llll} \langle E \rightarrow \text{“}hours\text{”}, & \tau\rangle & \langle E \rightarrow \text{“}hours\text{”}, & 0\rangle \\ \langle E \rightarrow \text{“}value\text{”}, & \tau\rangle & \langle E \rightarrow \text{“}value\text{”}, & 0\rangle \\ \langle E \rightarrow E \text{“} > 12\text{”}, & 1\rangle & \langle E \rightarrow E \text{“} > 12\text{”}, & 0\rangle \\ \langle E \rightarrow E \text{“} > 0\text{”}, & 1\rangle & \langle E \rightarrow E \text{“} > 0\text{”}, & 0\rangle \\ \langle E \rightarrow E \text{“} + \text{”} E, & 1\rangle & \langle E \rightarrow E \text{“} + \text{”} E, & 0\rangle \\ \langle E \rightarrow E \text{“} > \text{”} E, & 1\rangle & \langle E \rightarrow E \text{“} > \text{”} E, & 0\rangle \\ \langle T \rightarrow E, & 1\rangle & & \end{array} \quad (1)$$

The rules on the left expand the tree upwards. The rules on the right expand the remaining non-terminals generated during the process downward.

Please note that, to ensure that a program can be decomposed into choices, the set of expansion rules should be *complete*. To ensure the probability of the program is the product of the probabilities of the choices, the set of the expansion rules should be *unique*. Later we will introduce the definition of completeness and uniqueness and describe a method to generate a set of complete and unique expansion rules.

After generalizing grammar rules into expansion rules, the path-finding problem and the pruning method based on static analysis still can be similarly defined, as we will explain in detail in the next section.

3 APPROACH DETAILS

In this section we introduce the details of the L2S framework. As introduced before, the core idea of L2S is to generalize grammar rules into expansion rules (3.1), such that the probability of the whole program is the product of the probability of each expansion rule, easing probability prediction (3.2). Also, the problem of the finding the best program is converted into a path finding problem such that existing path finding algorithms can be used (3.3). Finally, L2S also introduces a static analysis on grammar rules to prune off infeasible paths (3.4).

3.1 Expansion Rules

We shall start with the basic definition for grammars and abstract syntax trees (ASTs) (Section 3.1.1) and then show how to generalize them into expansion rules and expansion trees (Section 3.1.2). Then we discuss how ASTs and expansion trees can be converted from each other (Section 3.1.3).

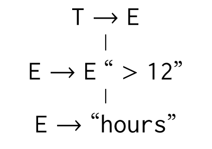
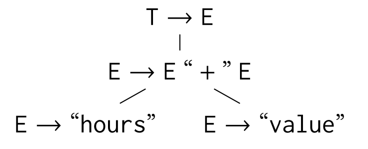
Program	`hours>12`	`hours+value`
AST		

Fig. 4. Example ASTs in L2S

3.1.1 Grammar Rules and ASTs . We start with the definition of grammar rule.

Definition 3.1 (Grammar Rule). Given a set of terminal symbols Σ, a set of non-terminal symbols N, and a root symbol T, where $\Sigma, N, \{T\}$ are disjoint with each other, a *grammar rule r* is a sequence of symbols taking the following form: $n_0 \longrightarrow n_1 \dots n_k$, where $n_0 \in N \cup \{T\}$ is called the *left symbol*, $n_1 \dots n_k \in N \cup \Sigma$ are called *right symbols*. If the left symbol is the root symbol, we call this rule a *start rule*.

We also introduce some notations to access the non-terminals in the grammar rules. We use $g[i]$ to denote the ith non-terminal in the right symbols of a grammar rule g, where i starts from 1. We also use $|g|$ to denote the number of non-terminals in the right symbols of g. If g is not a start rule, $g[0]$ indicates the left symbol, otherwise $g[0]$ is not defined. Symbol $g[i]$ is also not defined for any $i > |g|$. We use $g[i]\downarrow$ to denote that $g[i]$ is defined.

The generation of a program is a series of applications of grammar rules, starting from the root symbol. In each application, a non-terminal generated by a previous rule g' can be further expanded by a rule g whose left symbol matches the non-terminal. To capture this relation between g and g', we introduce the definition of connectivity.

Definition 3.2 (Connectivity of Rules). A grammar rule g is *connectible* to another rule g' at the ith symbol, denoted as $g \multimap\!\!\in g'[i]$, if $0 < i < |g'|$ and $g[0] = g'[i]$.

For example, E $\longrightarrow$ "hours" is connectible to E $\longrightarrow$ E " > 12" at the first right symbol.

Based on the above definitions, we introduce the definition of abstract syntax tree (AST). In L2S, an abstract syntax tree is viewed as a tree of grammar rules, where a grammar rule g_1 can be the ith child of a grammar rule g_2 only if g_1 is connectible to g_2 at i. For example, Figure 4 shows a few ASTs and their corresponding programs.

Definition 3.3 (AST). A *(partial) abstract syntax tree (AST)* is a tuple (V, p, G, l) where V is a set of vertexes, $p : V \longrightarrow ((V \times \mathbb{N}) \cup \{\tau\})$ is an injective parent function (where $\mathbb{N}$ is the set of natural numbers, $p(v) = (v', i)$ indicates that v is the ith child of v', and $p(v) = \tau$ indicates that v is the root), G is a set of grammar rules, and $l : V \longrightarrow G$ is a function labeling each vertex with a grammar rule, such that

- let $q^0(v) = \{v\}, q^i(v) = \left\{v'' \mid \exists j, v' \in q^{i-1}(v) : \ p(v') = (v'', j)\right\}$, and we know that for any $v \in V$, there does not exist $k > 0$, such that $v \in q^k(v)$, i.e., there is no cycle,
- $p(v) = (v', i)$ implies that $l(v) \multimap\!\!\in l(v')[i]$, i.e., a child rule should be connectible to its parent;
- $p(v) = \tau \implies \neg l(v)[0]\downarrow$, i.e., the root vertex is associated with a start rule.

Please note that the injectivity of the parent function ensures that there is at most one root vertex.

Given an AST (V, p, G, l), we use $nb^p(v, i)$ to denote the ith neighbor of vertex $v \in V$, where $i = 0$ indicates the parent vertex, and $i > 0$ indicates the ith child.

If there are unexpanded non-terminals, the partial AST can be further expanded. Formally, any (v, i) such that $0 < i < |l(v)| \wedge \forall v' \in V, p(v') \neq (v, i)$ is called a *(grammatical) expansion position.* An abstract syntax tree is *complete* if there is no expansion position.

3.1.2 Expansion Rules and Expansion Trees .

Based on the definition of grammar rules and AST, we now generalize grammar rules into expansion rules. An expansion rule is a grammar rule plus an index indicating the non-terminal from which the rule is expanded. If the index is τ, the rule is a creation rule and is applied initially to create the first node in the AST.

Definition 3.4 (Expansion Rule). An *expansion rule* r is a pair (g, i), where g is a grammar rule and i indicates the start of the expansion, i.e., $i \in \mathbb{N} \cup \{\tau\}$ such that $i \neq \tau \implies g[i]\downarrow$. When $i = 0$, r is called a *top-down* rule; when $i > 0$, r is called a *bottom-up* rule; when $i = \tau$, r is called a *creation* rule. We use r^g to denote the grammar rule g of r and use r^i to denote the index i of r. We refer to the ith symbol of g as the *start symbol* of expansion rule (g, i) when i is not τ.

Similar to grammar rules, we also define the connectivity of expansion rules. Intuitively, two expansion rules connect if their corresponding grammar rules connect, and the start symbol of one rule is generated by the other.

Definition 3.5 (Connectivity of Expansion Rules). An expansion rule $r = (g, i)$ is *connectible* to another expansion rule $r' = (g', i')$ at the jth non-terminal, denoted as $r \multimap\!\!\in r'[j]$, if $g \multimap\!\!\in g'[j]$, and the expansion directions match, i.e., $(i = 0 \wedge i' \neq j) \vee (i \neq 0 \wedge i' = j)$.

An AST captures the structure of the program according to a set of grammar rules. After we generalize grammar rules into expansion rules, a program is now generated by a set of expansion rules applications. To capture the structure of the program based on a set of expansion rules, we introduce the concept of *expansion tree.* An expansion tree is similar to an AST, except that each vertex is mapped to an expansion rule rather than a grammar rule.

Definition 3.6 (Expansion Tree). A *(partial) expansion tree* is a tuple (V, p, R, ϕ) where V is a set of vertexes, $p : V \longrightarrow ((V \times \mathbb{N}) \cup \{\tau\})$ is an injective parent function that defines the tree structure similarly to the one in AST, R is a set of expansion rules, and $\phi : V \longrightarrow R$ is a function labelling each vertex with an expansion rule, such that

- let $q^0(v) = \{v\}, q^i(v) = \left\{v'' \mid \exists j, v' \in q^{i-1}(v) : \ p(v') = (v'', j)\right\}$, we know that for any $v \in V$, there does not exist $k > 0$, such that $v \in q^k(v)$, i.e., there is no cycle;
- $p(v) = (v', i)$ indicates that $\phi(v) \multimap\!\!\in \phi(v')[i]$, i.e, a child rule should be connectible to its parent;
- $\exists v \in V : \phi(v)^i = \tau$, i.e., there should be a creation rule.

Similarly, given an expansion tree (V, p, R, ϕ), we use $nb^p(v, i)$ to denote the ith neighbor of vertex $v \in V$.

For example, Figure 5 shows expansion trees. The arrows show the order of expansion for facilitating understanding.

Lemma 3.7. *There exists exactly one $v \in V$ such that $\phi(v)^i = \tau$, i.e., exactly one vertex with a creation rule. We call v the initial vertex of the expansion tree.*

Proof. Assume there are two vertexes with creation rules. Since the vertexes form a tree, there is exactly one path between the two vertexes. As a result, there must be a vertex on the path that has two start symbols, which is impossible. □

Similar to an AST, an expansion tree can be further expanded if there exist an *expansion position.* Intuitively, an expansion position is a non-terminal that has been generated by a rule but not expanded by another rule, and is τ if the tree is empty.

Program	`hours>12`	`hours+value`
Expansion Trees	$$\begin{tikzcd} (\mathsf{T} \longrightarrow \mathsf{E}, 1) \\ (\mathsf{E} \longrightarrow \mathsf{E}\text{ “ }>12\text{”}, 1) \arrow[u] \\ (\mathsf{E} \longrightarrow \text{“}\mathtt{hours}\text{”}, \tau) \arrow[u] \end{tikzcd}$$	$$\begin{tikzcd} & (\mathsf{T} \longrightarrow \mathsf{E}, 1) & \\ & (\mathsf{E} \longrightarrow \mathsf{E}\text{ “ }+\text{ ” }\mathsf{E}, 1) \arrow[u] \arrow[dr] & \\ (\mathsf{E} \longrightarrow \text{“}\mathtt{hours}\text{”}, \tau) \arrow[ur] & & (\mathsf{E} \longrightarrow \text{“}\mathtt{value}\text{”}, \mathtt{0}) \end{tikzcd}$$

These expansion trees are based on the expansion rules in listed in Equation (1).

Fig. 5. Example Expansion Trees

Definition 3.8 (Expansion Position). Given an expansion tree $t = (V, p, R, \phi)$, an *expansion position* of t is either

- τ when $V = \emptyset$, i.e., the tree is empty, or
- any (v, i) when $|V| > 0$, such that
  - $\phi(v)^g[i]\downarrow$, i.e., the non-terminal exists at v,
  - $(i = 0 \implies p(v) = \tau)$, i.e., the non-terminal is the left symbol and the vertex has no parent, and
  - $(i \neq 0 \implies \forall v' \in V : p(v') \neq (v, i))$, i.e., the non-terminal is a right symbol and there is no corresponding child.

An expansion tree is *complete* if there is no expansion position.

If a rule can be used to expand an expansion position, the rule is called an *expansion candidate.*

Definition 3.9 (Expansion Candidate). Given an expansion tree $t = (V, p, R, \phi)$ and an expansion position ρ of t, an *expansion candidate* of t and ρ is either

- a creation rule when $\rho = \tau$, or
- an expansion rule $r = (g, j)$ when $\rho = (v, i)$, such that
  - $r \multimap\!\in \phi(v)[i]$ when $i \neq 0$, or
  - $\phi(v) \multimap\!\in r[j]$ when $i = 0$.

Finally, we define how an expansion candidate performs the expansion.

Definition 3.10 (Expanded Tree). Given an expansion tree $t = (V, p, R, \phi)$, an expansion position ρ and an expansion candidate $r = (g, j)$, an expanded tree of t, p and r is a new expansion tree (V', p', R, ϕ') such that

- $V' = V \cup \{v'\}$, where v' is a fresh vertex not in V;
- $\phi' = \phi \cup \{v' \mapsto r\}$.
- when $\rho = \tau$, $p' = \{v' \mapsto \tau\}$
- when $\rho = (v, i)$, $(i \neq 0 \implies p' = p[v' \mapsto (v, i)]) \wedge (i = 0 \implies p' = p[v \mapsto (v', j)][v' \mapsto \tau])$

Here $p[v \mapsto (v', j)]$ is a new function that maps v to (v', j) and maps any other vertex v'' to $p(v'')$.

3.1.3 Conversion between the two trees . Now we have two types of trees: the abstract syntax trees based on grammar rules and the expansion trees based on expansion rules. We show that the two types of trees can be converted from each other. This conversion is important because (1) once we synthesize an expansion tree, we need to convert it into an AST to obtain the program, and (2) during the training phase, we need to convert the ASTs into expansion trees to generate the training set.

We first consider the conversion from an expansion tree to an AST. Since an AST requires to start from the root symbol, not all expansion trees can be converted to AST: only those expansion trees where the root is mapped to an expansion rule with the root symbol can be converted. In other words, those that can be further expanded upward cannot be converted. When an expansion tree can be converted, the conversion is simple: by removing the second component in all expansion rules, we obtain an AST.

Definition 3.11 (From Expansion Tree to AST). An expansion tree (V, p, R, ϕ) *generates* an AST (V, p, G, l) such that $\forall v \in V, \phi(v)^g = l(v)$.

Theorem 3.12. *If there exists* $v \in V, p(v) = \tau$ *and* $\neg\phi(v)^g[0]\downarrow$, (V, p, R, ϕ) *generates* $(V, p, \{r^g \mid r \in R\}, \{v \mapsto \phi(v)^g \mid v \in V\})$.

Proof. Directly from the definition. □

However, the other direction is not that straightforward. First, given an AST (V, p, G, l) and a set of expansion rules R, it is not always possible to convert the AST into an expansion tree. For example, if a vertex in the AST is associated with a grammar rule g such that there is no $r \in R$ where $r^g = g$. Second, there may also be multiple ways to convert the grammar rule into an expansion rules. For example, if the expansion rules contain all top-down rules and all bottom-up rules, the AST can be generated from either top-down and bottom-up.

To enable training, we expect any AST can be converted to an expansion tree. Also, as will be seen later, to facilitate probability calculation, we need to require that only one expansion tree can be converted from an AST. Therefore, it is desirable for the expansion rules to have the following two properties, *completeness* and *uniqueness*.

Definition 3.13 (Completeness). A set of expansion rules R is said to be *complete* with respect to a set of grammar rule G, if for any AST (V, p, G, l), there exists an expansion tree (V, p, R, ϕ) that generates (V, p, G, l).

Definition 3.14 (Uniqueness). A set of expansion rule R is said to be *unique* with respect to and a set of grammar rule G, if for any AST (V, p, G, l), there exists at most one expansion tree (V, p, R, ϕ) that generates (V, p, G, l).

In the following we show a sufficient condition for a complete and unique rule set. Intuitively, such a rule set must include creation rules to create the initial vetex, and a set of bottom-up rules to expand the tree upward until we reach the root, and include all top-down rules to expand the remaining non-terminals generated during the process. We call such an expansion rule set a *regular rule set.*

Definition 3.15 (Regular Rule Set). A *regular (expansion) rule set* with respect to a grammar rule set G is a minimal set R of expansion rules satisfying the following conditions.

- $g \in G \wedge g[0]\downarrow \implies (g, 0) \in R$,
- $\forall root, child \in G : \neg(root[0]\downarrow) \wedge (child, root) \in ChildRules^* \implies isOK(child)$, where
  - $ChildRules = \{(child, parent) \mid \exists i : child \multimap\!\!\in parent[i] \wedge (parent, i) \in R\}$
  - $isOK(g) ::= |BURules(g)| = 1 \vee (|BURules(g)| = 0 \wedge (g, \tau) \in R)$.
  - $BURules(g) ::= \{(g, i) \in R \mid i \neq 0\}$,

The first condition ensures that a top-down rule is included for each grammar rule that is not a start rule. The second condition ensures that, for each AST, a unique vertex will be constructed by a creation rule, and there is a unique path from the vertex with the creation rule to the root that can be constructed by bottom-up rules. This second condition is checked inversely. First, for each start grammar rule, we check the condition *isOK*—if there is either a unique bottom-up expansion rule associated with it, or a creation rule, i.e., there exists one unique way to convert this start rule into an expansion rule. If there is an associated bottom-up expansion rule with a start symbol at i, we perform this check again for any grammar rule that can connect to this start rule at i. We recursively apply this check until we reach a creation rule. In the formal notations, this recursive check is captured by the reflexive transitive closure of the relation $ChildRule$, denoted as $ChildRule*$.

For example, a rule set containing all top-down rules (i.e., $\{(g, 0) \mid g \in G \wedge g[0]\downarrow\} \cup \{(g, \tau) \mid g \in G \wedge \neg(g[0]\downarrow)\}$), is a regular rule set. Also, the rule set in Formula (1) is a regular rule set.

Next we show that a regular set is complete and unique. To show this, we need the following lemma, which shows that the position of the creation rule determines the expansion tree.

Lemma 3.16. *Suppose both* (V, p, R, ϕ) *and* (V, p, R, ϕ') *generates* (V, p, G, l)*. If* $\exists v \in V, g \in G : \phi(v) = \phi'(v) = (g, \tau)$*, we know that* $\phi = \phi'$.

Proof. Since the vertexes form a tree, there is exactly one path between a vertex v and the vertex with the start symbol. As a result, the start symbol of v must be the one associated with the path, i.e., there is a unique choice for the expansion rule of v. □

With the lemma, we have the following theorem.

Theorem 3.17. *A regular rule set* R *with respect to* G *is complete and unique with respect to* G.

Proof. We show the completeness of R by constructing an expansion tree (V, p, R, ϕ) from an arbitrary AST (V, p, G, l). First let v be the root vertex. Based on the second condition of the regular rule set, we know that there must exist either a creation rule $(l(v), \tau)$ or a bottom-up rule $(l(v), i)$. If the latter, let v be the ith child of v and repeat the process. Since the height of the tree is finite and there is no bottom-up rule for leaf vertexes (since there is no non-terminal in the right symbols), the process must end on a vertex v' where a creation rule $(l(v'), \tau)$ exists. Then we choose the creation/bottom-up rule for each vertex visited during the process, and choose the top-down rule for all other vertexes. It is easy to see these choices form an expansion tree.

Then we show the uniqueness. Because the root vertex cannot be mapped to a top-down rule, the above process determines a unique possible vertex for the creation rule. Further because of Lemma 3.16, we know that the expansion tree is unique. □

Definition 3.15 also gives us a method to create a regular expansion rule set from a set of grammar rules. Algorithm 1 demonstrates this method. First we add a top-down rule for each grammar rule that is not a start rule. Then for each start rule g, we decide whether to add a creation rule or a bottom-up rule. If we choose to add a bottom-up rule (g, i), we need to find all grammar rules that connect to g at i, and decide for each such grammar rule whether we add a creation rule or a bottom-up rule. We repeat this process until no more bottom-up rules left.

Finally, we present a dynamic programming algorithm to convert an expansion tree into an AST, as shown in Algorithm 2. This algorithm has the time complexity of $O(nm^2)$ time, where n is the number of vertexes in the AST and m is the maximum number of non-terminals in the right hand side of a grammar rule. As m is usually very small and does not scale with respect to the size of the AST, this algorithm is effectively linear.

The core insight of the algorithm is that the expansion rule to which a vertex can be mapped is determined by whether its children can be constructed upward and/or downward. The algorithm visits the tree in the post order, and decides whether a vertex can be mapped to a top-down rule or a bottom-up/creation rule based on the status of the children. The 2-dimensional array `Feasible` is used to record this information, as well as the concrete rules for producing the results.

```
Input: $G$: a set of grammar rules
Output: $R$: a set of expansion rules
foreach $g \in G$ such that $g[0]\downarrow$ do
    $R \leftarrow R \cup \{g, 0\}$
end
foreach $g \in G$ such that $\neg g[0]\downarrow$ do
    AddRule($g$)
end
Procedure AddRule($g$)
    if $|g| > 0$ then
        Decide to add a bottom-up rule or a creation rule.
        if bottom-up is chosen then
            Decide the index $i$ of the start symbol, $1 \leq i \leq |g|$.
            $R \leftarrow R \cup \{g, i\}$
            foreach $g \in G$ such that $g' \dashv\!\in g[i]$ do
                AddRule($g$)
            end
            return
        end
    end
    $R \leftarrow R \cup \{g, \tau\}$
end
```

Algorithm 1: Convert a set of grammar rules into a regular expansion rule set

3.2 Probability

In this subsection we show the probability of an expansion tree is the product of the probabilities of choosing expansion candidates during an expansion process, and the probabilities of choosing expansion positions can be omitted. To show this, we first introduce the concept of *expansion sequence* to represent an expansion process.

Definition 3.18 (Expansion Sequence). An expansion sequence is a sequence of triples $(\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}\rangle)_{\mathtt{i=1}}^{\mathtt{n}}$, where $\mathtt{prog_1} = (\emptyset, \emptyset, R, \emptyset)$, and for any $1 \leq i \leq n$, $\mathtt{pos_i}$ is an expansion position of $\mathtt{prog_i}$, $\mathtt{rule_i}$ is an expansion candidate of $\mathtt{prog_i}$ and $\mathtt{pos_i}$, and $\mathtt{prog_{i+1}}$ is an expanded tree based on $\mathtt{prog_i}, \mathtt{pos_i}$, and $\mathtt{rule_i}$. We also use $\mathtt{prog_{n+1}}$ to denote the expanded tree based on $\mathtt{prog_n}, \mathtt{pos_n}$ and $\mathtt{rule_n}$.

We say an expansion tree $\mathtt{prog'}$ can be *expanded* from another expansion tree $\mathtt{prog}$, if there exists an expansion sequence $(\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}\rangle)_{\mathtt{i=1}}^{\mathtt{n}}$ such that $\mathtt{prog} = \mathtt{prog_j}$ and $\mathtt{prog'} = \mathtt{prog_k}$ and $0 \leq j < k \leq n + 1$.

We assume there exists a policy to select an expansion point when multiple expansion points exist. Formally, a policy is a function that maps an incomplete expansion tree to an expansion point. It is natural to assume that (1) the probability of a program is independent of the choice of a policy, and (2) the probability of choosing an expansion candidate for an expansion position is independent of the choice of a policy.

Now we present our theorem of probability and its proof. To prove this theorem, we need to first prove a lemma about the uniqueness of expansion sequences.

Lemma 3.19. *Given a unique set R of expansion rules, a policy* $\mathtt{policy}$*, an AST* $\mathtt{prog}$*, there exists at most one expansion sequence* $(\langle \mathtt{prog_i}, \mathtt{position_i}, \mathtt{rule_i}\rangle)_{\mathtt{i=1}}^{\mathtt{n}}$ *based on R such that* $\mathtt{policy}(\mathtt{prog_i}) = \mathtt{position_i}$ *for any $1 \leq i \leq n$, and* $\mathtt{prog_{n+1}}$ *generates t.*

```
Input: $(V, p, G, l)$: an AST
Input: $R$: a set of expansion rules
Output: $\phi$: A map from a vertex to an expansion rule
Data: Feasible: A map from a vertex and a Boolean to a expansion rule, or Nil to denote infeasible.
        The Boolean indicates whether the direction is top-down
// Initialize Feasible
foreach $v \in V$ do
    Feasible[$v$, $true$] ← Nil
    Feasible[$v$, $false$] ← Nil
end
// Calculate a solution in the postorder
foreach $v$ in the postorder of $V$ do
    foreach $r \in R$ such that $r^g = l(v)$ do
        if feasible($r, v$) then Feasible[$v, i = 0$] ← $r$ ;
    end
    if ¬Feasible[$v$, $true$] ∧ ¬Feasible[$v$, $false$] then return Nil;
end
// Recover the solution in the preorder
$root$ ← the root node
$\phi[root]$ ← Feasible[$root$, $true$] ≠ Nil ? Feasible[$root$, $true$] : Feasible[$root$, $false$]
foreach $v$ in the preorder of $V$ such that $v \neq root$ do
    $(v_p, i) \leftarrow p(v)$
    $(g, j) \leftarrow \phi[v_p]$
    $\phi[v]$ ← Feasible[$v, i \neq j$]
end
return $\phi$
Function feasible($r, v$) : Boolean
    // Determine if $v$ can be mapped to $(g, i)$.
    $(g, i) \leftarrow r$
    if $i \neq 0$ ∧ ¬Feasible[$nb^p(v, i)$, $false$] then return $false$;
    forall $j$ such that $0 < j < |g| \land j \neq i$ do
        if ¬Feasible[$nb^p(v, j)$, $true$] then return $false$;
    end
    return $true$
end
```

Algorithm 2: Converting an AST to an expansion tree

Proof. This is a direct result from the definition of unique expansion rule set. □

Theorem 3.20. *Given a unique set R of expansion rules, an AST* prog*, a context* context*, an expansion sequence* $(\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i} \rangle)_{i=1}^{n}$ *such that* $\mathtt{prog_{n+1}}$ *generates* prog*, we know that*

$$\mathtt{Pr(prog} \mid \mathtt{context)} = \prod_{i} \mathtt{Pr(rule_i} \mid \mathtt{context, prog_i, pos_i)}$$

Proof. In the following, we ignore context in all conditional probabilities as it is always available as a condition. Let policy be a policy that generates the above expansion sequence. Since the choice of a policy is independent of programs, we have that Pr(prog) = Pr(prog | policy).

Because of Lemma 3.19, we have that

$$\begin{aligned}
& \mathtt{Pr(prog)} \\
= \; & \mathtt{Pr(prog \mid policy)} \\
= \; & \mathtt{Pr((\langle prog_i, pos_i, rule_i\rangle)_{i=1}^{n} \mid policy)} \\
= \; & \mathtt{Pr(prog_1 \mid policy)Pr(pos_1 \mid policy, prog_1)Pr(rule_1 \mid policy, prog_1, pos_1)} \\
& \mathtt{Pr(prog_2 \mid policy, prog_1, pos_1, rule_1)\ldots} \\
& \mathtt{Pr(prog_{n+1} \mid policy, (prog_i)_{i=1}^{n}, (pos_i)_{i=1}^{n}, (rule_i)_{i=1}^{n})}
\end{aligned}$$

Further considering that $\mathtt{prog_i}, \mathtt{pos_i}$ and $\mathtt{rule_i}$ determines $\mathtt{prog_{i+1}}$, $\mathtt{policy}$ determines $\mathtt{pos_i}$, and R determines $\mathtt{prog_1}$, we know that $\mathtt{Pr(prog_{i+1} \mid prog_i, pos_i, rule_i, \ldots)} = 1, \mathtt{Pr(pos_i \mid policy, \ldots)} = 1, \mathtt{Pr(prog_1 \mid \ldots)} = 1$, and thus we have that

$$\begin{aligned}
\mathtt{Pr(prog)} \; = \; & \textstyle\prod_i \mathtt{Pr(rule_i \mid policy, (\langle prog_j, pos_j, rule_j\rangle)_{j=1}^{i-1}, prog_i, pos_i)} \\
= \; & \textstyle\prod_i \mathtt{Pr(rule_i \mid policy, prog_i, pos_i)} \\
= \; & \textstyle\prod_i \mathtt{Pr(rule_i \mid prog_i, pos_i)}
\end{aligned}$$

□

3.3 Path Finding Problem

In this section we formally define on how to convert the program estimation problem into a path finding problem based on expansion rules. Given the following items,

- a set of grammar rules G,
- a set of expansion rules R based on G that is unique and complete,
- a context `context`,
- a specification `spec` that is a predicate on complete ASTs defined on G,
- a probability estimation function pr that takes an expansion tree `prog`, an expansion point `pos`, and an expansion candidate `rule` as input, and returns an estimated value of the probability $\mathtt{Pr(rule \mid prog, pos, context)}$,

we define a path finding problem as the follows.

- A graph (V, E) where
  - V is the (possibly infinite) set that contains all expansion trees defined on R, i.e., all expansion trees taking the form $(\_, \_, R, \_)$, and
  - $(\mathtt{prog_1}, \mathtt{prog_2}) \in \mathtt{E}$ iff $\mathtt{prog_1}$ and $\mathtt{prog_2}$ are expansion trees, and there exists an expansion point `pos` of $\mathtt{prog_1}$ and an expansion candidate `rule`, such that $\mathtt{prog_2}$ is an expanded tree of $\mathtt{prog_1}$, `pos`, and `rule`
- A gain function *gain* defines the gain of a path on graph and is defined as follows

$$gain(e_1 e_2 \ldots e_n) = \prod_{i=1}^{n} pr(\mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i})$$

  where $e_1 e_2 \ldots e_n$ is n edges forming a path on graph, $\mathtt{prog_i}$ is the start vertex of e_i, $\mathtt{pos_i}$ is the expansion point of e_i, and $rule_i$ is the expansion candidate of e_i.
- A start vertex $(\{\}, \{\}, R, \{\}) \in V$;
- A set of goal vertexes $\{v \in V \mid$ there exists an AST t, s.t. t is complete, v generates t and $\mathtt{spec}(t)\}$.

The goal of the path finding problem is to find a path, whose gain is as large as possible, from the start vertex to a goal vertex. Since R is complete, all valid ASTs have been taken into consideration. Since R is unique, based on Theorem 3.20, we are looking for an AST whose estimated probability is as large as possible.

As mentioned, L2S does not confine a particular algorithm for solving the path finding problem and the user can choose any algorithm that best fits the target domain.

$cs(\mathsf{T} \rightarrow \mathsf{E}, 1) = \{\texttt{true}\}$ $cs(\mathsf{E} \rightarrow \mathsf{E}\ \text{“} > \texttt{12}\text{”}, 1)(E_0) = \{e_1 \mid e_0 \in E_0 \wedge$ $e_0 = \texttt{true} \rightarrow e_1 > 12 \wedge e_0 = \texttt{false} \rightarrow e_1 \leq 12\}$ $cs(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)(E_0) = \begin{cases} \mathit{true} & \text{if } 13 \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases}$	$$\begin{tikzcd} \llbracket(\mathsf{T} \rightarrow \mathsf{E}, 1)\rrbracket = \{\mathit{true}\} \\ \llbracket(\mathsf{E} \rightarrow \mathsf{E}\ \text{“} > \texttt{12}\text{”}, 1)\rrbracket = \{v \mid v > 12\} \arrow[u] \\ \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)\rrbracket = \mathit{true} \arrow[u] \end{tikzcd}$$

(a) For input-output example `hours=13`, `ret=true` and program `hours > 12`.

$cs(\mathsf{T} \rightarrow \mathsf{E}, 1) = \{10\}$ $cs(\mathsf{E} \rightarrow \mathsf{E}\ \text{“} + \text{”}\ \mathsf{E}, 1)(E_0, E_2)$ $= \{e_1 \mid e_0 \in E_0 \wedge e_2 \in E_2 \wedge e_0 = e_1 + e_2\}$ $cs(\mathsf{E} \rightarrow \text{“}\texttt{value}\text{”}, 0) = \{1\}$ $cs(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)(E_0) =$ $\begin{cases} \mathit{true} & \text{if } 13 \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases}$	$$\begin{tikzcd} & \llbracket(\mathsf{T} \rightarrow \mathsf{E}, 1)\rrbracket = \{10\} & \\ & \llbracket(\mathsf{E} \rightarrow \mathsf{E}\ \text{“} + \text{”}\ \mathsf{E}, 1)\rrbracket = 9 \arrow[u] \arrow[dr] & \\ \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)\rrbracket = \mathit{false} \arrow[ur] & & \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{value}\text{”}, \texttt{0})\rrbracket = \{1\} \end{tikzcd}$$

(b) For input-output example `hours=13`, `value=1`, `ret=10` and program `hours + value`.
D is the set of values that each sub expression could have.

Fig. 6. Example semantic constraint function and the concrete attributes

3.4 Pruning off Infeasible Partial Programs

In this section, we introduce how we prune off the infeasible partial programs based on static analysis on expansion rules. Infeasible partial programs are the programs that cannot lead to a complete program satisfying the specification. To achieve this, L2S assumes the specification can be represented as functions over expansion rules, called *constraint functions*. Each constraint function produces an output at the start symbol based on values collected from other non-terminals. As a result, given an expansion tree, the constraint function produces a value at each vertex, called a *vertex attribute*. The vertex attribute at the initial vertex determines whether the expansion tree satisfies the specification or not.

Definition 3.21 ((Concrete) Constraint Function). A (concrete) constraint function cs based on a set D maps an expansion rule (g, i) to a function, where the input of the function is a sequence of subsets of D corresponding to each non-terminal in the rule except for the ith non-terminal, and the output of the function is a subset of D corresponding to the ith non-terminal. When i is τ, the output of the function is Boolean, indicating whether the specification is satisfied or not.

Common specifications can be represented in constraint functions. For example, a common type of semantic specification is input-output examples. The left column of Figure 6 shows two specifications with different input-output examples. In the figures, we only show the expansion rules that are used in the expansion trees in the right column, but in real applications, semantic functions need to be defined for all expansion rules. Please note that a concrete constraint function is not necessarily computable and is used to specify the semantics of the constraint. Later we will introduce abstract constraint functions that perform the actual checks of the partial expansion trees.

Besides semantic specification, other types of specifications can also be represented as constraint functions. A basic specification for a program is that it should be typable. The left column of Figure 7 shows an example constraint function for type check. The basic idea is to compute all possible types a non-terminal can have. Another specification that is often used in program synthesis is size limit: the size of the synthesized program is limited to avoid searching for too large programs.

$$\begin{aligned} &cs(\mathsf{T} \rightarrow \mathsf{E}, 1) = \{\texttt{Boolean}\} \\ &cs(\mathsf{E} \rightarrow \mathsf{E} \text{ “ + ” } \mathsf{E}, 1)(E_0, E_2) = \\ &\quad E_0 \cap E_2 \cap \{\texttt{Int}, \texttt{Float}\} \\ &cs(\mathsf{E} \rightarrow \text{“}\texttt{value}\text{”}, 0) = \{\texttt{Int}\} \\ &cs(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)(E_0) = \\ &\quad \begin{cases} \mathit{true} & \text{if } \texttt{Int} \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{tikzcd} & \llbracket(\mathsf{T} \rightarrow \mathsf{E}, 1)\rrbracket = \{\texttt{Int}\} & \\ & \llbracket(\mathsf{E} \rightarrow \mathsf{E} \text{ “ + ” } \mathsf{E}, 1)\rrbracket = \{\texttt{Int}\} \arrow[u] \arrow[dr] & \\ \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)\rrbracket = \mathit{true} \arrow[ur] & & \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{value}\text{”}, \texttt{0})\rrbracket = \{\texttt{Int}\} \end{tikzcd}$$

D is the set of type that each sub expression could have.

Fig. 7. Example type constraint function and the concrete attributes

$$\begin{aligned} &cs(\mathsf{T} \rightarrow \mathsf{E}, 1) = 1 \\ &cs(\mathsf{E} \rightarrow \mathsf{E} \text{ “ + ” } \mathsf{E}, 1)(E_0, E_2) = E_0 + E_2 + 1 \\ &cs(\mathsf{E} \rightarrow \text{“}\texttt{value}\text{”}, 0) = 1 \\ &cs(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)(E_0) = \\ &\quad \begin{cases} \mathit{true} & \text{if } E_0 + 1 \le 3 \\ \mathit{false} & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{tikzcd} & \llbracket(\mathsf{T} \rightarrow \mathsf{E}, 1)\rrbracket = \{0, 1, 2\} & \\ & \llbracket(\mathsf{E} \rightarrow \mathsf{E} \text{ “ + ” } \mathsf{E}, 1)\rrbracket = \{0\} \arrow[u] \arrow[dr] & \\ \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{hours}\text{”}, \tau)\rrbracket = \mathit{false} \arrow[ur] & & \llbracket(\mathsf{E} \rightarrow \text{“}\texttt{value}\text{”}, \texttt{0})\rrbracket = \{1\} \end{tikzcd}$$

Assume the maximum size of the AST is 3
D is the set of natural numbers.

Fig. 8. Example size constraint function and the concrete attributes

The left column of Figure 8 shows an example constraint function for checking the size limit. This constraint function calculates the number of vertexes that is expanded from the current the vertex plus the current one.

Based on the constraint functions, we give the definition of concrete vertex attributes.

Definition 3.22 ((Concrete) Vertex Attribute). Given a complete expansion tree (V, p, R, ϕ) and a constraint function cs, the (concrete) attribute of vertex $v \in V$, denoted as $\llbracket v \rrbracket$, is defined as follows.

$$\llbracket v \rrbracket = cs(\phi(v)) \left((\llbracket nb^p(v, j) \rrbracket)_{j=0}^{\phi(v)^i - 1}, (\llbracket nb^p(v, j) \rrbracket)_{j=\phi(v)^i + 1}^{|\phi(v)|} \right)$$

To check whether a complete expansion tree satisfies a constraint, we just need to check whether $\llbracket v_0 \rrbracket$ is true, where v_0 is the initial vertex. For example, the right columns of Figure 6, Figure 7, and Figure 8 show the vertex attributes on four example expansion trees based on the previous example constraint functions.

Concrete constraint function defines the semantics of the specification on complete programs. To check whether a partial program is feasible, L2S utilizes a user-defined abstract domain based on the abstract interpretation framework [12]. The abstract domain is used for two computations. First, an offline static analysis is performed on expansion rules to calculate the *abstract upward/downward attribute* for each non-terminal, which is an abstract value covers all possible concrete vertex attributes that a vertex expanded upward/downward from this non-terminal could have. Second, an online analysis is performed on the partial expansion tree to calculate the *abstract vertex attributes*, which cover concrete vertex attributes calculated from all possible complete trees. This second analysis relies on the abstract upward/downward attributes calculated from the first analysis.

The user-defined abstract domain should satisfy some basic requirements. The abstract interpretation framework captures such requirements by the Galois connection.

Definition 3.23 (Galois Connection). A concrete domain D, a finite abstract domain A, a partial order $\sqsubseteq$ over A, and two functions $\alpha : 2^D \rightarrow A$ and $\gamma : A \rightarrow 2^D$ form a Galois connection $(2^D, \subseteq) \leftrightarrows^{\gamma}_{\alpha} (A, \sqsubseteq)$ if and only if $\forall d \in 2^D, a \in A : \ \alpha(d) \sqsubseteq a \Leftrightarrow d \subseteq \gamma(a)$.

For simplifying presentation, below we assume the existence of a Galois connection $(2^D, \subseteq) \leftrightarrows^{\gamma}_{\alpha} (A, \sqsubseteq)$. We also assume that there exists a safe union operator $\sqcup$ on the abstract domain A that is consistent with $\sqsubseteq$, i.e., $a_1 \sqsubseteq a_2 \Leftrightarrow a_1 \sqcup a_2 = a_2 \wedge \alpha(\gamma(a_1) \cup \gamma(a_2)) \sqsubseteq a_1 \sqcup a_2$.

The computation over the abstract domain is captured by an *abstract constraint function*, which is a function abstract of the concrete constraint function. Please note that for creation rules, an abstract domain for the result is directly provided rather than user-defined: returning `true` indicates that the concrete constraint function may return `true`, while returning `false` indicates that the concrete constraint function never returns `true`.

Definition 3.24 (Abstract Constraint Function). A constraint function acs based on A abstracts a constraint function cs based on D iff for any expansion rule $r = (g, i)$ where $i \neq \tau$, $acs(r)$ is monotone and is a safe function abstraction of $cs(r)$, i.e., $\alpha \circ cs(r)(\gamma(a_1), \dots, \gamma(a_n)) \sqsubseteq acs(r)(a_1, \dots, a_n)$, and for any expansion rule $r = (g, \tau)$, we have $\exists d \subseteq \gamma(a) : cs(r)(d) \implies acs(r)(a)$.

For example, the left columns of Figure 9, Figure 10, and Figure 11 show the abstract constraint functions for the previous example concrete constraint functions. For input-output examples, an element in abstract domain is an interval [12]. For type checking, an element in the abstract domain is a set of types. For size limit, an element in the abstract domain is a lowerbound of the size.

Given a computable constraint function acs that abstracts a constraint function cs, we can calculate the two types of abstract attributes mentioned before.

Definition 3.25 (Abstract Upward/Downward Attributes). An abstract upward/downward attribute of a non-terminals in R is a minimal value in A satisfying one of the following equations, where $\llfloor N \rrfloor$ denotes the abstract upward attribute of non-terminal N, $\llceil N \rrceil$ denotes the abstract downward attribute of non-terminal N.

For each N where $\exists g : (g, 0) \in R \wedge g[0] = N$:

$$\llceil N \rrceil = \bigsqcup_{r=(g,0)\in R \wedge g[0]=N} acs\,(r) \left((\llceil g[j] \rrceil)_{j=1}^{|g|} \right)$$

For each N where $\exists g : (g, i) \in R \wedge g[i] = N \wedge i > 0$:

$$\llfloor N \rrfloor = \bigsqcup_{r=(g,i)\in R \wedge g[i]=N \wedge i>0} acs(r) \left(\llfloor g[0] \rrfloor, (\llceil g[j] \rrceil)_{j=1}^{i-1}, (\llceil g[j] \rrceil)_{j=i+1}^{|g|} \right)$$

Since the abstract domain is finite and the functions produced by acs are monotone, these equations can be solved by a standard fixed-point algorithm [2].

Definition 3.26 (Abstract Vertex Attribute). Given a partial expansion tree (V, p, R, ϕ), the abstract attribute of vertex $v \in V$, denoted as $\|v\|$, is defined as follows.

$$\|v\| = acs(\phi(v)) \left((attr(v, j))_{j=0}^{\phi(v)^i - 1}, (attr(v, j))_{j=\phi(v)^i+1}^{|\phi(v)|} \right)$$

where

$$attr(v, j) = \begin{cases} \|nb^p(v, j)\| & \text{if } nb^p(v, j) \text{ exists} \\ \llfloor \phi(v)[0] \rrfloor & \text{elif } j = 0 \\ \llceil \phi(v)[j] \rrceil & \text{else} \end{cases}$$

$$
\begin{aligned}
&acs(\mathsf{T} \rightarrow \mathsf{E}, 1) = \{\mathtt{true}\}\\
&acs(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } > 12\text{”}, 1)(E_0) =\\
&\quad\begin{cases} [-\infty, +\infty] & \text{if } \{\mathtt{true}, \mathtt{false}\} \sqsubseteq \mathtt{E}_0\\ [12, +\infty] & \text{elif } \{\mathtt{true}\} \sqsubseteq \mathtt{E}_0\\ [-\infty, 12] & \text{elif } \{\mathtt{false}\} \sqsubseteq \mathtt{E}_0\\ \emptyset & \text{else}\end{cases}\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)(E_0) = \begin{cases} true & \text{if } [13, 13] \sqsubseteq E_0\\ false & \text{otherwise}\end{cases}
\end{aligned}
$$

$$\begin{tikzcd}
\|(\mathsf{T} \rightarrow \mathsf{E}, 1)\| = \{true\}\\
\|(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } > 12\text{”}, 1)\| = [12, +\infty] \arrow[u]\\
\|(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)\| = true \arrow[u]
\end{tikzcd}$$

(a) For input-output example hours=13, ret=true and program hours > 12.

$$
\begin{aligned}
&acs(\mathsf{T} \rightarrow \mathsf{E}, 1) = [10, 10]\\
&acs(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } + \text{ ” }\mathsf{E}, 1)(E_0, E_2) =\\
&\quad (E_0 - E_2)\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{value}\text{”}, 0)(E_0) = [1, 1]\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)(E_0) =\\
&\quad\begin{cases} true & \text{if } [13, 13] \sqsubseteq E_0\\ false & \text{otherwise}\end{cases}
\end{aligned}
$$

$$\begin{tikzcd}
& \|(\mathsf{T} \rightarrow \mathsf{E}, 1)\| = [10, 10] &\\
& \|(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } + \text{ ” }\mathsf{E}, 1)\| = [-\infty, 9] \arrow[u] \arrow[dr] &\\
\|(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)\| = false \arrow[ur] & & \lceil\!\lceil \mathsf{E} \rceil\!\rceil = [1, +\infty]
\end{tikzcd}$$

(b) For input-output example hours=13, value=1, ret=10 and partial program hours + E.

Fig. 9. Example Abstract Semantic Constraint Function and the abstract attributes

$$
\begin{aligned}
&acs(\mathsf{T} \rightarrow \mathsf{E}, 1) = \{\mathtt{Boolean}\}\\
&acs(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } + \text{ ” }\mathsf{E}, 1)(E_0, E_2) =\\
&\quad E_0 \cap E_2 \cap \{\mathtt{Int}, \mathtt{Float}\}\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{value}\text{”}, 0) = \{\mathtt{Int}\}\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)(E_0) =\\
&\quad\begin{cases} true & \text{if } \mathtt{Int} \in E_0\\ false & \text{otherwise}\end{cases}
\end{aligned}
$$

$$\begin{tikzcd}
& \|(\mathsf{T} \rightarrow \mathsf{E}, 1)\| = \{\mathtt{Int}\} &\\
& \|(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } + \text{ ” }\mathsf{E}, 1)\| = \{\mathtt{Int}\} \arrow[u] \arrow[dr] &\\
\|(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)\| = true \arrow[ur] & & \lceil\!\lceil \mathsf{E} \rceil\!\rceil = \{\mathtt{Int}, \mathtt{Float}, \mathtt{Boolean}\}
\end{tikzcd}$$

Fig. 10. Example Abstract Type Constraint Function and the abstract attributes

$$
\begin{aligned}
&acs(\mathsf{T} \rightarrow \mathsf{E}, 1) = [1, +\infty]\\
&acs(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } + \text{ ” }\mathsf{E}, 1)(E_0, E_2) = E_0 + E_2 + [1, +\infty]\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{value}\text{”}, 0)(E_0) = [1, +\infty]\\
&acs(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)(E_0) =\\
&\quad\begin{cases} true & \text{if } 3 \in (E_0 + [1, +\infty])\\ false & \text{otherwise}\end{cases}
\end{aligned}
$$

$$\begin{tikzcd}
& \|(\mathsf{T} \rightarrow \mathsf{E}, 1)\| = [0, 2] &\\
& \|(\mathsf{E} \rightarrow \mathsf{E}\text{ “ } + \text{ ” }\mathsf{E}, 1)\| = [0, 0] \arrow[u] \arrow[dr] &\\
\|(\mathsf{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)\| = false \arrow[ur] & & \lceil\!\lceil \mathsf{E} \rceil\!\rceil = [1, +\infty]
\end{tikzcd}$$

Fig. 11. Example Abstract Size Constraint Function and the abstract attributes

For example, the right columns of Figure 9, Figure 10, and Figure 11 give examples of the abstract upward/downward attributes and the abstract vertex attributes calculated for some partial expansion trees.

We prune off a partial expansion tree when the abstract vertex attribute of the initial vertex is `false`, i.e., this partial tree can never be expanded to a complete tree that satisfies the specification. Now we give a theorem to show that this pruning is safe.

Theorem 3.27. *Given a partial expansion tree* prog $= (V, p, R, \phi)$*, for any complete expansion tree* prog$' = (V', p', R, \phi')$ *that can be expanded from* prog*, we have* $[\![v_0]\!] \implies \|v_0\|$*, where v_0 is the initial vertex.*

Proof. We show the theorem holds in three steps: (1) we show that for any $v \in V'$, $i \neq 0 \implies [\![v]\!] \subseteq \gamma(\lfloor\!\lfloor N \rfloor\!\rfloor)$ and $i = 0 \implies [\![v]\!] \subseteq \gamma(\lceil\!\lceil N \rceil\!\rceil)$, where $i = \phi(v)^i$ and $N = \phi(v)^g[\phi(v)^i]$, (2) $[\![v]\!] \subseteq \gamma(\|v\|)$ for any $v \in V$ for any $v \in V$ that is not the initial vertex, and (3) the original theorem holds.

(1) According to the definition of concrete attribute, any concrete attribute is calculated by applications of the functions $cs(\phi(v))$. We show this property holds by induction over k, where k indicates the maximum number of applications. When $k = 1$ and $i = 0$, we have

$$\begin{aligned}
[\![v]\!] &= cs(\phi(v))() && \textit{the definition of concrete vertex attribute}\\
&\subseteq \gamma(acs(\phi(v))()) && \textit{safety of asbtract constraint function}\\
&\subseteq \gamma(\lceil\!\lceil N \rceil\!\rceil) && \textit{safety of union}
\end{aligned}$$

Similarly we can show the case for $k = 1$ and $i \neq 0$.

Now assume that the property holds for all $k < k'$ and we consider the case $k = k'$. We first consider the case where $i = 0$. Let $v_i = nb^{p'}(v, i)$ and $n = |\phi(v)|$, we have

$$\begin{aligned}
[\![v]\!] &= cs(\phi(v))([\![v_1]\!], \dots, [\![v_n]\!]) && \textit{the definition of concrete vertex attribute}\\
&\subseteq \gamma\left(acs(\phi(v))(\alpha(\{[\![v_1]\!]\}), \dots, \alpha\{[\![v_n]\!]\})\right) && \textit{safety of abstract constraint function}\\
&\subseteq \gamma(acs(\phi(v))(\lceil\!\lceil \phi(v)^g[1] \rceil\!\rceil, \dots, \lceil\!\lceil \phi(v)^g[n] \rceil\!\rceil)) && \textit{induction assumption and the monotonicity of } \alpha, \gamma, \textit{ and } acs(\phi(v))\\
&\subseteq \gamma(\lceil\!\lceil N \rceil\!\rceil) && \text{safety of union}
\end{aligned}$$

Similarly we can prove the case for $i \neq 0$. Putting together, we prove the first step.

(2) Proving the second step is similar. We perform an induction over k, where k is the number of abstract vertex attributes that are calculated during the calculation of $\|v\|$. Initially we consider the case where $k = 1$ and $i = 0$. Since only one abstract vertex attribute is calculated, we know that v has no children, i.e., $nb^p(v, j)$ does not exist for any $j > 0$. Then we have

$$\begin{aligned}
&[\![v]\!] = cs(\phi(v))([\![v_1]\!], \dots, [\![v_n]\!]) && \textit{the definition of concrete vertex attribute}\\
&\subseteq \gamma\left(acs(\phi(v))(\alpha(\{[\![v_1]\!]\}), \dots, \alpha\{[\![v_n]\!]\})\right) && \textit{safety of abstract constraint function}\\
&\subseteq \gamma(acs(\phi(v))(\lceil\!\lceil \phi(v)^g[1] \rceil\!\rceil, \dots, \lceil\!\lceil \phi(v)^g[n] \rceil\!\rceil)) && \textit{the result of (1) and the monotonicity of } \alpha, \gamma, \textit{ and } acs(\phi(v))\\
&= \gamma(\|v\|) && \text{the definition of abstract vertex attribute}
\end{aligned}$$

Similarly, we can prove for the case where $k = 1$ and $i \neq 0$.

Now assume that the property holds for all $k < k'$. We first consider the case $k = k'$ and $i = 0$. We show that $[\![v]\!] \in \gamma(attr(v, j))$ holds for any $1 \leq j \leq |\phi(v)^g|$: $attr(v, j)$ can only be either $\|nb^p(v, j)\|$ or $\lceil\!\lceil \phi(v)[j] \rceil\!\rceil$. Because of the induction assumption, the first case holds. Because of the results of (1), the second case holds. Then $[\![v]\!] \in \gamma(attr(v, j))$ holds. Based on this property, we have

$$\begin{aligned}
&[\![v]\!] = cs(\phi(v))([\![v_1]\!], \dots, [\![v_n]\!]) && \textit{by definition}\\
&\subseteq \gamma\left(acs(\phi(v))(\alpha(\{[\![v_1]\!]\}), \dots, \alpha\{[\![v_n]\!]\})\right) && \textit{safety of abstract constraint function}\\
&\subseteq \gamma(acs(\phi(v))(attr(v, 1), \dots, attr(v, n))) && \textit{by the above discussion and the monotonicity of } \alpha, \gamma, \textit{ and } acs(\phi(v))\\
&= \gamma(\|v\|) && \text{the definition of abstract vertex attribute}
\end{aligned}$$

Similarly, we can prove for the case where $k = k'$ and $i \neq 0$.

(3) Based on the result of (2), the original theorem can be obtained from the definition of abstract constraint function. □

4 L2S-COND

In this section we describe how we implement L2S-Cond, an instantiation of L2S for synthesizing Java conditional expressions where the surrounding code as context and the typing rules as specification. To instantiate L2S, we need to design four sub-modules: (1) an expansion rule set to define the search space, (2) machine learning models to estimate the conditional probability, (3) a search algorithm to solve the path finding problem, and (4) abstract constraint functions to prune off infeasible paths early. In the rest of this section we describe how we design the four modules.

In particular, the design of each module involves multiple design choices, and we decide some of the choices empirically. We first supply several design options for each sub-module, then we perform a set of controlled experiments to systematically explore the influence of the options on the performance of L2S-Cond, and select the best overall design based on the experimental results.

4.1 Expansion rules

4.1.1 Grammar Rules. Expansion rules are derived from grammar rules. To define expansion rules, we need to first give the grammar rules of the conditional expressions. One problem of defining the grammar rules is that conditional expressions at different locations have different sets of accessible variables and methods, and thus it is difficult to define one unique set of grammar rules for all locations. To deal with this problem, we use two strategies, respectively for variables and methods. For variables, since variables accessible at different locations are largely different, we use different sets of variables for different locations. Compared with variables, the sets of methods accessible at different locations are only slightly different. Furthermore, we observe that in conditional expressions the methods are usually combined in fixed ways: for example, `a.length > 0` is often used rather than `a.length + b.length > c.length` for some arrays `a`, `b`, and `c`. We call such a fixed combination of Boolean type a *test*. We mine all tests from the training set and use them as part of the grammar. As a result, our grammar is partially fixed, the fixed part includes the logical operators and the tests, and the unfixed part includes the variables extracted from current local context.

Another design consideration is that designing grammar rules is a trade-off between the number of non-terminals (depth) and the number of choices (width). For example, the following two rule sets describe the same language, but set (2) has more non-terminals than set (3), while each non-terminal has less choices.

$$\mathtt{E} \rightarrow \mathtt{V}\ \text{“}>\text{”}\ \mathtt{L},\quad \mathtt{V} \rightarrow \text{“}\mathtt{a}\text{”} \mid \text{“}\mathtt{b}\text{”},\quad \mathtt{L} \rightarrow \text{“}\mathtt{0}\text{”} \mid \text{“}\mathtt{1}\text{”} \tag{2}$$

$$\mathtt{E} \rightarrow \text{“}\mathtt{a} > \mathtt{0}\text{”} \mid \text{“}\mathtt{b} > \mathtt{0}\text{”} \mid \text{“}\mathtt{a} > \mathtt{1}\text{”} \mid \text{“}\mathtt{b} > \mathtt{1}\text{”} \tag{3}$$

To understand how the number of non-terminals and the number of choices affect the performance of L2S-Cond, we design two sets of grammar rules, shown as Figure 12. Figure 12(a) have more non-terminals, where only atomic tests are mined from the training set and a non-terminal E recursively defines how these atomic tests are combined by logical operators. On the other hand, Figure 12(b) defines a flattened set, where the tests containing logical operators are directly mined from the training set and no more attempt is considered to combine them.

4.1.2 Expansion Rules. Expansion rules are derived from the grammar rules based on Algorithm 1. The main decision consideration is where to put the creation rule to start search. To understand the influence of this design choice, we use two strategies to generate an expansion rule set from a grammar rule set. The first one is top-down, where we always decide to add a creation rule at line 9 of Algorithm 1, i.e., the creation rule is at the root vertex and the tree is expanded top-down. The second one is bottom-up, where we always decide to add a bottom-up rule at line 9 of Algorithm 1

```
T    →      E
E    →      L                          | L “&&” E
     |      L “||” E                   | “!” E
L    →      “FastMath.abs(”V“) > V”    | V“.isEmpty()”
     |      V “ > ” V                  | V “ == 0”
     |      V “! = 0”                  | V “ ∗ ” V “ == 0” | ...
V    →      “overflow”                 | “a0”
     | “u”  | “v”                      | ...
```

(a) The recursive grammar rule set

```
T    →      E
E    →      “FastMath.abs(”V“) > ”V    | V“.isEmpty()”
     |      V “ > ” V                  | V “ == 0”
     |      V “ == 0 ||” V “ == 0”     | V “ ∗ ” V “ == 0” | ...
V    →      “overflow”                 | “a0”
     | “u”  | “v”                      | ...
```

(b) The flat grammar rule set

Fig. 12. Two grammar rule sets in L2S-Cond
E: expression. L: boolean literal. V: variable.

and choose 1 at line 11 of Algorithm 1, i.e., the creation rule is at the leftmost leaf vertex and the tree is first expanded bottom-up. We apply the top-down strategy to the recursive grammar set, and apply the both strategies to the flat grammar set, and obtain three expansion rule sets as shown in Figure 13. Later we will empirically evaluate the performance of the three sets to understand their influences to the performance of L2S-Cond.

Please note that technically more strategies can be defined. For example, we may randomly choose to add a bottom-up rule or a top-down rule at line 9 of Algorithm 1. However, such a strategy may lead a rule set conceptually difficult to implement and debug. We leave such an exploration as future work.

4.2 Machine-Learning Methods

For each non-terminal n and each expansion direction d (top-down or bottom-up), we need to train a machine-learning model to estimate the probabilities of the expansion rules starting from n along d. For each case where $n = \mathsf{V}$ and d is top-down, we train a binary classifier to predict the probability of a rule as the variables expanded from V differ from location to location. For other cases, we train a multi-class classification model where each class corresponds to an expansion rule whose start symbol is n and its direction is d. Given a set of condition expressions for training, we first parse the expressions into expansion trees, and each vertex in an expansion tree forms a training data sample for the corresponding classifier.

4.2.1 Feature Engineering. Applying machine learning approaches requires feature engineering. A challenge of feature engineering over programs is how to encode strings. Programs contain a lot of strings, such as variable names, method names, type names, etc. Although these strings can be encoded using standard methods such as label encoder, which maps different strings to different integers, the relationship between the identifiers would be lost. In particular, a common substring of strings often suggests some common attributes. For example, the identifiers “length”, “len”, “xLen” have a common part “len” and are all related to length. Similarly, identifiers “mutationRate”, “crosoverRate”, “elistimRate” have a common part “rate” and all related to rate.

$$\begin{array}{llll} \langle T \rightarrow E, \quad \tau \rangle & \langle E \rightarrow L \text{ “||” } E, \quad 0 \rangle & \langle L \rightarrow V \text{ “ > ” } V, \quad 0 \rangle & \langle V \rightarrow \text{“}u\text{”}, \quad 0 \rangle \\ & \langle E \rightarrow L, \quad 0 \rangle & \langle L \rightarrow V \text{“ == 0”}, \quad 0 \rangle & \langle V \rightarrow \text{“}v\text{”}, \quad 0 \rangle \\ & \ldots & \ldots & \ldots \end{array} \tag{4}$$

(a) The recursive expansion rule set derived by the grammar rule set of Figure 12(a)

$$\begin{array}{lll} \langle T \rightarrow E, \quad \tau \rangle & \langle E \rightarrow V \text{ “ > ” } V, \quad 0 \rangle & \langle V \rightarrow \text{“}u\text{”}, \quad 0 \rangle \\ & \langle E \rightarrow V \text{“ == 0 ||” } V \text{“ == 0”}, \quad 0 \rangle & \langle V \rightarrow \text{“}v\text{”}, \quad 0 \rangle \\ & \ldots & \ldots \end{array} \tag{5}$$

(b) The top-down expansion rule set derived by the grammar rule set of Figure 12(b)

$$\begin{array}{lll} \langle T \rightarrow E, \quad 1 \rangle & \langle E \rightarrow V \text{ “ > ” } V, \quad 1 \rangle & \langle V \rightarrow \text{“}u\text{”}, \quad \tau \rangle \\ & \langle E \rightarrow V \text{“ == 0 ||” } V \text{“ == 0”}, \quad 1 \rangle & \langle V \rightarrow \text{“}u\text{”}, \quad 0 \rangle \\ & \ldots & \langle V \rightarrow \text{“}v\text{”}, \quad \tau \rangle \\ & & \langle V \rightarrow \text{“}v\text{”}, \quad 0 \rangle \\ & & \ldots \end{array} \tag{6}$$

(c) The bottom-up expansion rule set derived by the grammar rule set of Figure 12(b)

E: expression. L: boolean literal. V: variable.

Fig. 13. The expansions rule sets of Figure 12

To capture this relationship, we encode a string into a Boolean model containing bi-gram of characters. The vector has n × n dimensions, where n is the length of the alphabet containing all possible characters. If a bi-gram is presented, the corresponding dimension is set of 1, otherwise is set to 0. For example, encoding `len` we would get a vector where the elements of “le" and “en" are 1 while all other elements are 0. Since the vector is large and sparse, we perform PCA analysis [13] to reduce the dimensions to no more than 20. Here we only consider bi-gram but not N-gram with a higher N, because the vector for bi-gram is already very large.

This encoding also allows us to encode a set of strings, where a bi-gram presented in any of the string is considered to be presented in the set.

Based on the encoding, we design features for machine learning. As previously illustrated, the conditional probability of a choice is `Pr(rule | context, prog, pos)`. To train a model, we need to extract features from the four parts, `context`, `prog`, and `pos`. For the binary classifiers dealing with variables, we need to also extract features from the variables, i.e., `rule`. Here we give a high-level description of the features extracted from the four parts.

Context features. Context features describe the surrounding context information of a code location, including class information, method information and flow information. Class information is extracted from the surrounding class, such as package name, class name, field names and field types. Method information are extracted from the containing method, such as its signature, lines of code, parameter names, etc. Flow information includes the local variable names and the structural information, such as the types of closest control-flow statements (for, if, ...) [62].

Program features. Program features are extracted from the (partial) conditional expression already generated, including features related to variables and operators in the expression. Variable features include name, type, and how many lines the last assignment is away from the condition. Operator features include whether certain operators are present or not.

Position features. Position features are extracted based on the position of the node to be expanded, such as its index among the siblings, the direct parent symbol, the sibling symbols, and whether the vertex to be expanded is an argument of a method call, and the index of a argument.

Rule features. The rule features are only available when expanding V downward, as in other cases we have multi-class models. When expanding V downward, the choices are variables, and we extract features about the variable including name, type, how many times it is used in other conditional expressions, how many times it is used in the body of the conditional statement, how many lines the the last assignment of the variable is away from the condition, etc.

4.2.2 Machine Learning Algorithm Options. To estimate the probabilities of expansion rule choices, we employ a statistical machine-learning algorithm. A candidate machine-learning algorithm should fulfill the following requirements: (1) it is able to output probabilities; (2) it is able to handle unbalanced training data; (3) its outputs should be as precise as possible.

Based on the requirements, we consider three machine-learning algorithms, namely Naive Bayes , Support Vector Machine (SVM), and XGBoost [9]. Naive Bayes is a classic statistical learning method based on Bayes' theorem, which assumes that the features are independent. SVM finds a hyperplane to distinctly separates the data points. XGBoost is a tree-based learning algorithm which is reasonably fast among modern machine learning algorithm. All algorithms directly produce probabilities as output. Later we empirically evaluate the influences of these algorithms to the performance of L2S-Cond.

4.3 Search Algorithm

To understand the performance of different path-finding algorithms, we implement two algorithms: an exact algorithm—the Dijkstra's algorithm and an approximation algorithm—the beam search algorithm [44]. The Dijkstra's algorithm maintains a list of visited vertexes and the highest gains to reach these vertexes. In each iteration, it picks the vertex with the highest gain and add all its neighbors to the list. The process continues until we reach a goal vertex. The beam search algorithm is similar to Dijkstra's search except that it maintains a list of only the top K vertexes with the highest gains. The parameter K is called the *beam width*. Later we empirically evaluate the influences of the two algorithms to the performance of L2S-Cond.

4.4 Abstract Constraint Functions

We implement two abstract constraint functions in L2S-Cond: a type constraint function and a size constraint function. The type constraint function is similar to those in Figure 10 and filters out infeasible partial expansion trees based on the typing rules of Java. The size constraint function filters out partial expansion trees whose any complete form has a depth higher than a threshold N, and are similar to those in Figure 11. In our current implementation, we set N to 4. Later we empirically evaluate the influences of the type constraint function to the performance of L2S-Cond. We do not evaluate the influence of the size constraint function as it is only effective for the recursive expansion rule set, which, as the evaluation will show later, is not the most effective rule set.

4.5 Evaluation of L2S-Cond

We have implemented L2S-Cond in Java based on Eclipse-JDT [4].

4.5.1 Research Questions. We aim to understand the overall performance of L2S-Cond as well as the influences of the sub-modules. Specifically, we answer the following research questions:

[4]https://www.eclipse.org/jdt/

- **RQ1**: What is the overall performance of each configuration?
- **RQ2**: What are the influences to the L2S-Cond performance of the three expansion rule sets, *TD*, *BU* and *RECUR*?
- **RQ3**: What are the influences to the L2S-Cond performance of the path finding algorithms, Dijkstra's algorithm and beam search?
- **RQ4**: What are the influences to the L2S-Cond performance of the statistical learning algorithms, XGBoost and Naive Bayes?
- **RQ5**: What is the influence to the L2S-Cond performance of the type constraint function?
- **RQ6**: Does the randomness introduced by selecting the training set affect the performance?

Table 1. Statistics of the subject projects

Project	**Bug ID**	**KLoc***	**# If**	**Validation Set Size**	**Avg. # Token**
Apache Commons Math	106	9	635	60	6.34
Apache Commons Lang	65	28	1880	183	5.60
Joda Time	27	50	1879	186	5.08
JFree Chart	26	37	5261	523	4.74
Total	-	131	9655	952	5.09

The **KLoc** is reported by *cloc*.

4.5.2 Dataset. We select four Java projects from the benchmark Defects4J [29] v1.0.0, namely Apache-Commons-Math [5], Apache-Commons-Lang [6], Joda-Time [7] and JFree-Chart [8]. We select these four projects because they would also be used in our evaluation for L2S-Hanabi in the next section. For each project, we select the version associated with the biggest bug ID. We extract all the conditional expressions inside `if` statements and their corresponding contexts as our dataset. For each project, we randomly select 90% of the `if` statements as training set, and train models over the training set. Then we equip L2S-Cond with the models and validate it over the remaining 10% `if` statements. As shown in Table 1, in total we collect a validation set consisting of 961 `if` statement conditional expressions to evaluate L2S-Cond. To measure the size of the expressions, we count the average numbers of the tokens per expression, as shown in the last column. In this experiment each expression contains 5.09 tokens on average.

To mitigate the potential bias of the above random selection of the training set and the validation set, we repeat the experiment three times with three random division of the training set and the validation set, and report the average results. We also analyze the derivation among the three executions in RQ6.

4.5.3 Setup. To answer the latter four research questions, we need to explore the influence of each design option for each component. However, since the combinatorial space of all options is very large, we design a set of single-factor experiments. First, we conducted a pilot study on the project involving the smallest number of 64 conditional expressions, Apache-Common-Math, and decided a potentially effective configuration of all the options, as shown in the first line of Table 2. Then, we use the pre-selected configuration as the default configuration, each time we change the option for one sub-module, and observe the performance changes. The set of all configurations we use in our experiment is shown in Table 2, and we use the following steps to answer the research questions.

- We use the first, default configuration to answer the RQ1.

[5]http://commons.apache.org/proper/commons-math/
[6]https://commons.apache.org/proper/commons-lang/
[7]https://www.joda.org/joda-time/
[8]https://www.jfree.org/jfreechart/

- By comparing the configurations **1-3**, we answer RQ2, the influences of the three expansion rule sets.
- By comparing the configurations **1, 4, 5 and 6**, we answer RQ3, the influences of the search algorithms and the effectiveness of beam width K.
- By comparing the configurations **1 7 and 8**, we answer RQ4, the influences of the machine learning methods.
- By comparing the configurations **5 and 9**, we answer RQ5, the influence of the type constraint function.
- By comparing the results of three independent runs against the configurations **1-8**, we answer RQ6, the randomness introduced by the selection of training data.

All the experiments in this section are performed in parallel in three VMware virtual machines, each of which is equipped with four cores of an Intel Core i7-9870H processor and 8GB memory.

Table 2. Configurations

ID	Expansion Rules	Learning Algorithms	Path Finding Algorithms	Abstract Constraint Function
1	BU	XGBoost	Beam 400	Type Constraint
2	TD	XGBoost	Beam 400	Type Constraint
3	RECUR	XGBoost	Beam 400	Type Constraint
4	BU	XGBoost	Beam 100	Type Constraint
5	BU	XGBoost	Beam 25	Type Constraint
6	BU	XGBoost	Dijkstra	Type Constraint
7	BU	Naive Bayes	Beam 400	Type Constraint
8	BU	SVM	Beam 400	Type Constraint
9	BU	XGBoost	Beam 400	None

"Beam 400" indicates the beam search algorithm with K=400. "None" indicates that the type constraint function is not used.

4.5.4 Metrics. We use two metrics to measure the accuracy and the efficiency of each configuration. For accuracy, we use the "TOP N" metric, which indicates the number of conditional expressions where the correct expression is within the top N predicted candidates. We consider an expression as correct if it is token-by-token the same as the ground-truth. For efficiency, we calculate the time needed for producing the top N candidates for the largest N.

Table 3. Performance of the configuration **1** on the validation set

ID	TOP K	Chart 26	Lang 65	Math 106	Time 27	SUM	PCT.
1 (BU-XGB-Bm400-Tp)	**TOP 5**	318	104	27	115	564	58.7%
	TOP 25	357	124	35	134	650	67.7%
	TOP 100	367	144	37	141	689	71.7%
	TOP 200	370	150	39	142	701	72.9%
	TIME(S)	4989	1929	970	2815	10704	

"BU-XGB-Bm400-Tp" is the abbreviation of the option "the *BU* expansion rule set, XGBoost method, the beam search algorithm with K=400 and type constraint function used". The line **TIME(S)** indicates the total execution time for generating the top 200 candidates for all conditional statements in our validation set. The column **PCT.** indicates the percentage of correctly synthesized expressions on the whole validation set.

4.5.5 RQ1: The overall performance of L2S-Cond. The overall performance of the configuration **1** is shown in Table 3. For the total 961 conditional expressions, L2S-Cond synthesizes *722 (75.1%)*

expressions within *Top 200*, *710 (73.9%)* expressions within *Top 100*, *684 (71.2%)* expressions within *Top 25*, and *593 (61.7%)* expressions within *Top 5*. L2S-Cond in total costs 9681 seconds for 961 conditional expressions, and on average it costs 10 seconds for synthesizing 200 candidate expressions for one conditional statement. These performance data indicate that L2S-Hanabi has the potential to be applied in different downstream applications, such as code completion and program repair.

Table 4. Performance of the expansion rule sets, *BU*, *TD* and *RECUR*

ID	TOP N	Chart 26	Lang 65	Math 106	Time 27	SUM
1 (BU-XGB-Bm400-Tp)	TOP 5	318	104	27	115	564
	TOP 25	357	124	35	134	650
	TOP 100	367	144	37	141	689
	TOP 200	370	150	39	142	701
	TIME(S)	4989	1929	970	2815	10704
2 (TD-XGB-Bm400-Tp)	TOP 5	301	102	21	98	522
	TOP 25	378	133	32	133	676
	TOP 100	402	147	37	143	729
	TOP 200	404	150	38	143	736
	TIME(S)	22831	7207	1779	8813	40630
3 (RC-XGB-Bm400-Tp)	TOP 5	281	97	20	93	491
	TOP 25	340	119	28	123	610
	TOP 100	357	128	33	130	649
	TOP 200	362	132	33	131	657
	TIME(S)	29166	7909	2645	9001	48721

"RC" indicates the *RECUR* expansion rule set.

4.5.6 RQ2: The influence of the three expansion rule sets. Table 4 shows the comparison of the three expansion rule sets. In terms of accuracy, we can see that the performances of *BU* and *TD* are close, especially over *TOP 100* and *TOP 200*, and *BU* performs better at *TOP 5* and *TOP 25*. On the other hand, the performance of *RECUR* significantly lags behind *BU* and *TD*. In terms of efficiency, *BU* is significantly faster than the other rule sets, where *RECUR* costs 4.17 times as much as *BU* and *TD* costs 4.10 times as much as *BU*.

We conjecture the reason for the high efficiency of *BU* is that, as we discussed in Section 2.5, the variables often have large probability differences in a conditional expression, and thus *BU* converges to the probable expressions quickly.

To understand the accuracy difference between the three rule sets, we further analyze the overlaps between the correctly generated expressions by each rule set. We compare the results based on the second round experiment, because in this round the three rule sets have relatively even sets of uniquely correctly generated expressions. In total, these rules generate 808 if conditions, and TD/BU/Recur generates 789/693/698 conditions, respectively. Figure 14 shows the overlaps among the three rules by a Venn diagram, where the red/green/blue circle represents TD/BU/Recur, respectively. As we can see, there is a large overlap (602 conditions) among the three rule sets and each rule set uniquely correctly synthesizes only a small part of the expressions (22, 5, 11, respectively). Among the uniquely correctly synthesized expressions, we found that TD is good at handling expressions with `instanceof` and expressions with many variables, BU is good at handling expressions with `||`, while Recur is good at handling expressions without logical operators, i.e., &&, `||` and `!`.

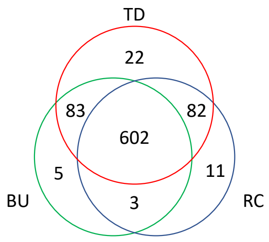


Fig. 14. The overlaps among the rules

Table 5. Performance of the Dijkstra Algorithm and beam search with different widths

ID	TOP N	Chart 26	Lang 65	Math 106	Time 27	SUM
1 (BU-XGB-Bm400-Tp)	TOP 5	318	104	27	115	564
	TOP 25	357	124	35	134	650
	TOP 100	367	144	37	141	689
	TOP 200	370	150	39	142	701
	TIME(S)	4989	1929	970	2815	10704
4 (BU-XGB-Bm100-Tp)	TOP 5	318	104	27	115	564
	TOP 25	357	124	35	134	650
	TOP 100	365	140	37	139	681
	TOP 200	365	140	37	139	681
	TIME(S)	3215	975	601	1637	6428
5 (BU-XGB-Bm25-Tp)	TOP 5	318	104	27	115	564
	TOP 25	356	123	34	131	644
	TOP 100	356	123	34	131	644
	TOP 200	356	123	34	131	644
	TIME(S)	1554	422	251	685	2913
6 (BU-XGB-DJ-Tp)	TOP 5	318	104	27	115	564
	TOP 25	357	124	35	134	650
	TOP 100	367	144	37	141	689
	TOP 200	370	150	39	142	701
	TIME(S)	5138	1829	976	2509	10451

"DJ" is short for Dijkstra's algorithm. "Bm*K*" indicates beam search method with beam width *K*.

4.5.7 RQ3: The influence of search algorithms. Table 5 shows the comparison of the configuration **1**, **4**, **5** and **6**. As we can see, with the increase of beam width *K*, the accuracy and the inference time both increase. When $K = 400$, the accuracy and the inference time are both the same as or very close to those of Dijkstra's algorithm. Also, we observe that a small beam width already gives good accuracy in the first few candidates. For example, with $K = 25$, the performance of beam search at TOP 5 is already the same as Dijkstra' algorithm. This suggests that we should choose the beam width based on how many candidates we need. In the next section we shall construct a repair tool L2S-Hanabi based on L2S-Cond. Since in program repair the inference time of around 10 seconds is not critical, and hundreds of candidate expressions could be tested before the correct one is found, we still choose for L2S-Hanabi the option in the default configuration, i.e., beam search with $K = 400$.

Table 6. The performance of different machine learning methods

ID	TOP N	Chart 26	Lang 65	Math 106	Time 27	SUM
1 (BU-XGB-Bm400-Tp)	TOP 5	318	104	27	115	564
	TOP 25	357	124	35	134	650
	TOP 100	367	144	37	141	689
	TOP 200	370	150	39	142	701
	TIME(S)	4989	1929	970	2815	10704
7 (BU-NB-Bm400-Tp)	TOP 5	150	43	8	63	264
	TOP 25	193	52	11	76	332
	TOP 100	195	52	11	76	334
	TOP 200	195	52	11	76	334
	TIME(S)	1077	236	139	268	1720
8 (BU-SVM-Bm400-Tp)	TOP 5	84	58	11	45	198
	TOP 25	97	79	15	53	244
	TOP 100	101	85	18	56	260
	TOP 200	101	89	19	56	265
	TIME(S)	3781	1117	715	892	6505

"NB" is short for Naive Bayes.

4.5.8 RQ4: The influences of the machine learning methods. Table 6 shows the comparison of the three machine learning methods. From the table we can find that *Naive Bayes* and *SVM* are significantly behind *XGBoost* in terms of accuracy. *Naive Bayes* is the fastest as its algorithm is relatively simple, *SVM* is much slower, and *XGBoost* is the slowest due to its complexity. As discussed previously, since the inference time of around 10 seconds is not critical, in the construction of L2S-Hanabi in the next section we will still choose *XGBoost*, as the default configuration.

4.5.9 RQ5: The influence of the type constraint function. The goal of the type constraint function is to speed up the synthesis by pruning off infeasible partial programs early and avoid validating the incorrect complete programs. To measure this speedup, we add a validation procedure to our experiment. For each synthesized expression, we invoke the compiler to check if it is well-typed. Since the validation requires a significant amount of time, we stop when we get 25 well-typed expressions, or we have consumed all programs within the beam width.

Table 7. The influence of the type constraint function

ID	TOP K	Chart 26	Lang 65	Math 106	Time 27	SUM
1 (BU-XGB-Bm400-Tp)	**Cmpl.**	62566	11305	2898	12660	89430
	TIME(S)	30290	6147	1219	6411	44068
8 (BU-XGB-Bm400-NONE)	**Cmpl.**	105300	28207	9064	31931	174503
	TIME(S)	50235	12901	3462	15777	82374

"Cmpl." is the total number of times to invoke the compiler.

Table 7 shows the comparison between the configuration with or without the type constraint function. We observe that, to get the same amount of well-typed expression, the one with the type constraint function requires only 51.9% compiler invocations, and 50.4% of the total time. The results confirm the effectiveness of the type constraint function, and we still retain it in L2S-Hanabi.

4.5.10 RQ6: The effects of the randomness in training data selection. The previous research questions were answered with the average results of three times experiments. To understand the derivation between the three divisions of training set and validation set, we further analyze the standard derivation, as shown in Table 8. The columns **1st**, **2nd** and **3rd** show the results of the three divisions, respectively. The column **AVG** shows the arithmetic averages and the column **SD** shows the standard deviation. As we can see, the derivations are all small and do not affect the findings of the previous research questions.

Table 8. Comparison of the results of 3 times evaluation

ID		1st	2nd	3rd	AVG	SD
1 (BU-XGB-Bm400-Tp)	**TOP 25**	684	636	631	650.3	23.9
	TOP 200	722	693	688	701.0	15.0
2 (TD-XGB-Bm400-Tp)	**TOP 25**	649	747	632	676.0	50.7
	TOP 200	721	789	697	735.7	39.0
3 (RC-XGB-Bm400-Tp)	**TOP 25**	608	650	573	610.3	31.5
	TOP 200	661	698	611	656.7	35.6
4 (BU-XGB-Bm100-Tp)	**TOP 25**	684	636	631	650.3	23.9
	TOP 200	705	664	673	680.7	17.6
5 (BU-XGB-Bm25-Tp)	**TOP 25**	678	626	627	643.7	24.3
	TOP 200	678	626	627	643.7	24.3
6 (BU-XGB-DJ-Tp)	**TOP 25**	684	636	631	650.3	23.9
	TOP 200	722	693	688	701.0	15.0
7 (BU-NB-Bm400-Tp)	**TOP 25**	336	320	339	331.7	8.3
	TOP 200	337	323	342	334.0	8.0
8 (BU-SVM-Bm400-Tp)	**TOP 25**	242	258	233	244.3	10.3
	TOP 200	253	283	258	264.7	13.1

5 L2S-HANABI

Based on L2S-Cond, we build a program repair approach, L2S-Hanabi, for repairing buggy conditional statements. In this section, we present the implementation details of L2S-Hanabi and evaluate it on 272 real-world defects.

5.1 Workflow

L2S-Hanabi repairs buggy conditional statements based on L2S-Cond. L2S-Hanabi takes as input a buggy program and a set of tests where at least one test fails, and produces a patch that repairs a buggy conditional expression in the program. The design of L2S-Hanabi follows the state-of-the-art approach to condition repair, ACS [68], and uses L2S-Cond for condition synthesis. L2S-Hanabi consists of three main components, including fault localization, patch generation and patch validation.

5.2 Fault Localization

This step takes the buggy program and its test suite which contains at least one failing test as input, and returns a list of suspicious locations to be fixed. If the location is a conditional expression, it is a candidate to be replaced with a new expression. If the location is not a conditional expression, it is a candidate for inserting a new conditional statement. Following ACS, we use a popular spectrum-based fault localization technique, Ochiai [1], to localize the suspicious locations. When Ochiai localizes to a conditional expression, we further apply predicate switching [73] to verify if it is indeed buggy. That is, we invert the returned Boolean value of a conditional expression, and then check whether it can pass the failing test.

5.3 Patch Generation

For each a suspicious location, L2S-Hanabi attempts to apply one of the following templates to repair the bug.

- **Template 1: Repairing buggy conditional expression.**
  `if` (c_o) $\Rightarrow$ `if` (c)
  If the suspicious location is a conditional expression c_0, L2S-Hanabi synthesizes a new condition c using L2S-Cond to replace the original one.
- **Template 2: Inserting a missing boundary check.**
  $\Rightarrow$ `if` (c) `return` v
  $\Rightarrow$ `if` (c) `throw` e
  If the suspicious location is not a conditional expression, L2S-Hanabi attempts to use this template to insert an `if` statement before the suspicious location. First, L2S-Hanabi analyzes the failed test and check if the test is expecting an error code v or an exception e. An expected output v is an error code if the output is written as a const static field of a primitive type in the test. If the test expects an error code or an exception, L2S-Hanabi assumes a boundary check may be missing, synthesizes a new condition c using L2S-Cond and inserts the `if` statement as shown in the template, otherwise L2S-Hanabi skips this suspicious location.
- **Template 3: Repairing return or throw statement.**
  `return` v_o $\Rightarrow$ `return` v
  `throw` e_o $\Rightarrow$ `throw` e
  If the fault localization identifies a return or throw statement as faulty, we directly change the returned value or the exception to the expected one. The expected values are extracted from the oracle of the failed test.

The first two templates are inherited from ACS, which rely on L2S-Cond to synthesize a conditional expression. Moreover, we found that 27% (4/15) of the patches of the first template in ACS add a redundant check, as follows.

```
    if(c){
+     if(c){return v1;}
      return v0;}
```

To generate more natural patches and avoid the duplicated check, we add the third template to directly modify the return or throw statements. We apply the templates in the order of $3 \longrightarrow 2 \longrightarrow 1$. For the first two templates, we use L2S-Cond to synthesize M conditional expressions to form M patches, and validate them one by one. If none of the M conditional expressions passes the tests, we proceed to the next suspicious location. Currently we set M to 200 based on a pilot study on a small set of subjects.

5.4 Patch Validation

For each generated patch, we validate it by the test suite in the project. Since existing work has revealed that prioritizing the tests could significantly accelerate the validation [48], we prioritize the tests using the following order:

(1) Execute the failing test method(s).
(2) Execute the test methods from the same test cases as the failing test methods.
(3) Execute the tests from the same package as the failing test methods.
(4) Execute all remaining tests.

In Defects4J there are cases where under the same bug ID there are a multi-hunk patch and multiple failing tests, where each hunk repairs a bug revealed by a failing test. In other words, there are actually multiple bugs under one bug ID. Following the existing practice [27, 68], we consider that a patch passes the validation if the number of failing tests decreases. In this way, we enable the fix of these multi-bug cases.

5.5 Evaluation of L2S-Hanabi

5.5.1 Research Questions. We evaluate L2S-Hanabi by answering the following three research questions:

- **RQ7**: How does L2S-Hanabi perform over real-world defects?
- **RQ8**: How does the performance of L2S-Hanabi compare with existing APR approaches?
- **RQ9**: What are the contributions of the three patch generation templates?

Table 9. Statistics of the used projects

Project	**#Bugs**	**Dataset**	**KLoc***	**Test KLoc***	**#Test Classes**
Apache Commons Math	106	Defects4J	84	86	497
Apache Commons Lang	65	Defects4J	22	38	128
Joda Time	27	Defects4J	28	53	155
JFree Chart	26	Defects4J	96	49	389
Apache Accumulo	16	Bugs.jar	154	20	147
Apache Camel	32	Bugs.jar	116	130	2,277
Total	272	-	500	376	3,593

KLoc and **#Test Cases** are taken from the most recent version, as reported by *cloc*.

Table 10. Training versions for repair Defects4J

Training	# of Bugs	Year	Training	# of Bugs	Year
Math 12	12	2013	Lang 5	5	2013
Math 37	25	2012	Lang 14	9	2012
Math 59	22	2011	Lang 24	10	2011
Math 75	16	2010	Lang 35	11	2010
Math 94	19	2009	Lang 43	8	2009
Math 102	8	2008	Lang 48	5	2008
Math 104	2	2007	Lang 55	7	2007
Math 106	2	2006	Lang 65	10	2006
Time 11	11	2013	Chart 4	4	2009
Time 17	6	2012	Chart 16	12	2008
Time 24	7	2011	Chart 26	10	2007
Time 27	3	2010			
ACCUMULO-4098	2	2016	CAMEL-8584	6	2015
ACCUMULO-3746	4	2015	CAMEL-7130	14	2014
ACCUMULO-1661	7	2014	CAMEL-6987	2	2013
ACCUMULO-1544	1	2013	CAMEL-5770	4	2012
ACCUMULO-907	1	2012	CAMEL-3960	5	2011
ACCUMULO-151	1	2011	CAMEL-3388	1	2010

5.5.2 Dataset. Our dataset is extracted from the Defects4J [29] benchmark and the Bugs.jar [54] benchmark, which are real-world Java defects benchmarks. Table 9 shows the statistics about our dataset. Because the four subjects from Defects4J are libraries, we further add two non-library projects from Bugs.jar to avoid bias. Since the number of bugs in Bugs.jar is large and different bugs often require different setups, we follow a common practice in existing studies [32, 55] to only evaluate on bugs that require single-hunk patches. In the end, we have 272 defects from six diverse projects.

5.5.3 Training L2S-Hanabi. As aforementioned, L2S-Hanabi relies on L2S-Cond to synthesize conditional expressions. As L2S-Cond is driven by data, We need to prepare a training set. we train L2S-Cond using the conditional expressions extracted from the same project to get the best performance. We assume in a real-world usage, the team may periodically re-train the program repair tool using the newest code. To simulate this scenario, we train L2S-Cond using the first buggy project of a calendar year, and repair the bugs occurred in the same year based on the trained models. Table 10 gives the details of the training versions, the number of corresponding bugs to repair and the covered calendar year. Since some early versions do not have enough expressions to train an effective synthesizer, we also added the source code of JDK as an additional training set.

5.5.4 RQ7: Repair performance of L2S-Hanabi.

Setup. To answer RQ6, we evaluated L2S-Hanabi over all the 272 bugs of our dataset. We set a time budget for each bug as 180 minutes. Following existing studies [27, 55, 66, 68], we configured L2S-Hanabi to terminate at the first patch that passes all the tests.

We determine a patch as *plausible* if it passes all the tests, and as *correct* if it is semantically equivalent to the developer-provided patch. We manually checked the correctness of all the patches, which is a standard practice in previous work [27, 43, 55, 64, 66, 68]. To reduce the possible errors that may be introduced in this manual process, following the recommendation of Le et.al. [33], we further evaluate the patches labeled as correct against independent test suites. If a patch is not textually equal to the developer's patch, we invoke EVOSUITE [16] to generate a new test suite based on the fixed version and validate the patch with this test suite. We label the patch as incorrect if the patch is unable to pass the enhanced tests. Furthermore, we release all the patches for public judgement in our project repository. We measure the performances of the approaches by precision and recall, where precision is defined as the portion of correct patches among plausible patches, and recall is defined as the portion of correctly repaired bugs among all bugs.

Table 11. Performance of repair on our dataset

Project	**#Bugs**	**Correct**	**Incorrect**	**Precision**	**Recall**
Math	106	19	3	86%	18%
Lang	65	4	0	100%	6%
Time	27	2	0	100%	7%
Chart	26	3	2	60%	12%
Results on Defects4J	224	28	5	85%	13%
Accumulo	16	1	0	100%	6%
Camel	32	3	1	75%	9%
Total Results	272	32	6	84%	12%

Precision=Correct/(Correct+Incorrect)×100%. **Recall**=Correct/#Bugs×100%.

Results. Table 11 shows the overall repair results of L2S-Hanabi on our dataset. Our approach generates 38 plausible patches in total, 32 of which are correct, achieving a high precision of 84% (32/38) and a recall of 12% (32/272).

If we group the results by benchmark, L2S-Hanabi has a precision of 85% (28/33) and a recall of 13% (28/224) on Defects4J and a precision of 80% (4/5) and a recall of 9% (4/45) on Bugs.jar. The results show that the performance of L2S-Hanabi is stable across different types of projects and different benchmarks.

5.5.5 RQ8: Comparison with existing approaches on Defects4J.

Setup. We compare L2S-Hanabi with a set of existing repair approaches as baselines. We select the baseline approaches by the following criteria: (1) the approach was published after 2017; (2) the approach was evaluated on Defects4J; (3) all the generated patches have been released. As a result, we compare L2S-Hanabi with nine state-of-the-art APR approaches: TBar [37], SimFix [27], CapGen [64], SketchFix [24], ELIXIR [55], JAID [7], ssFix [66], ACS [68] and Nopol [70]. Among these approaches, ACS and Nopol are designed for repairing conditional statement bugs, which are the same as L2S-Hanabi, while the others are aiming to fix more general defects. Some approaches generate more than one patch for a bug, and we only consider the first reported patch for a fair comparison. L2S-Hanabi and all the listed techniques have been evaluated on the same 224 defects from the four projects of Defects4J (i.e. Math, Lang, Time and Chart), enabling us to compare their performance directly.

We compare from two aspects. First, as L2S-Hanabi is designed for bugs in conditional statements only, we compared the performance of these APR techniques in terms of fixing bugs in conditional statements. We selected a subset of the 144 conditional statement bugs out of the 224 bugs, and compare the nine techniques in terms of precision and recall. This subset was selected by Sobreira et al. [58] and is publicly available[9]. The selection criterion is whether the bug is related to conditional statements. Second, we compared the performance of the APR techniques on all of the 224 bugs, to get overall comparison results.

[9]http://program-repair.org/defects4j-dissection

Table 12. Detailed comparison on all the 143 conditional bugs

Proj.	**Total**	**Hanabi**	**ACS**	**Nopol**	**TBar**	**SimFix**	**SketchFix**	**CapGen**	**ELIXIR**	**JAID**	**ssFix**
Chart	16	3 (1)	2 (0)	1 (3)	5 (2)	1 (3)	2 (1)	1 (0)	2 (1)	1 (2)	1 (2)
Math	62	15 (4)	8 (4)	1 (11)	2 (12)	3 (9)	4 (1)	4 (2)	4 (3)	0 (5)	3 (10)
Lang	47	2 (0)	1 (1)	3 (1)	3 (7)	7 (4)	1 (0)	0 (0)	1 (3)	1 (3)	1 (6)
Time	19	2 (0)	1 (0)	0 (0)	0 (1)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (2)
Total	144	22 (5)	12 (5)	5 (15)	10 (22)	11 (16)	7 (2)	5 (2)	8 (7)	2 (10)	5 (20)
Precision	-	**81%**	71%	25%	31%	41%	78%	71%	53%	17%	20%
Recall	-	**15%**	8%	3%	7%	8%	5%	3%	6%	1%	3%

Table 13. Detailed comparison on all the 224 bugs of Defects4J

Proj.	**Total**	**Hanabi**	**ACS**	**Nopol**	**TBar**	**SimFix**	**SketchFix**	**CapGen**	**ELIXIR**	**JAID**	**ssFix**
Chart	26	3 (2)	2 (0)	1 (5)	9 (5)	4 (4)	6 (2)	4 (0)	4 (3)	2 (2)	3 (4)
Math	106	19 (3)	12 (4)	1 (20)	19 (17)	14 (12)	7 (1)	12 (4)	12 (7)	1 (7)	10 (16)
Lang	65	4 (0)	2 (2)	3 (4)	5 (9)	9 (3)	3 (1)	5 (0)	8 (4)	1 (7)	5 (7)
Time	27	2 (0)	1 (0)	0 (1)	1 (2)	1 (0)	0 (1)	0 (0)	2 (1)	0 (0)	0 (4)
Total	224	28 (5)	17 (6)	5 (30)	34 (33)	28 (19)	16 (5)	21 (4)	26 (15)	4 (16)	18 (31)
Precision	-	**85%**	74%	14%	51%	60%	76%	84%	63%	20%	37%
Recall	-	13%	8%	2%	**15%**	13%	7%	9%	12%	2%	8%

A bold number represents it achieves highest performance. **X (Y)**: X is the number of correct patches, Y is the number of incorrect plausible patches. L2S-Hanabi, ACS and Nopol are listed first because they are designed for conditional statement bugs. The remaining APR techniques are sort by their publication year.

Comparison on Conditional Statement Bugs. Table 12 exhibits the repair results of L2S-Hanabi and other nine APR techniques. Compared with these APR techniques, L2S-Hanabi fixes 22 out of the 144 defects, which achieves the largest number of bugs fixed and the highest recall. Moreover, within all the 27 plausible patches, L2S-Hanabi only generates five incorrect patches, achieving the highest precision. Compared with the two techniques aiming to fix conditional statement bugs, L2S-Hanabi outperforms ACS and Nopol both in recall and precision. We can conclude that in terms of fixing conditional statement bugs, L2S-Hanabi outperforms the existing APR techniques.

Comparison on All Bugs. Table 13 shows the results from different techniques on all the 224 defects of Defects4J. Note that sometimes the developers' patch does not modify or insert a conditional statement, but is equivalent to a patch that uses an `if` statement. For example, in Math-35, the developers' patch changes the target of a method call to a method that contains an addition boundary check, and an APR technique can create a patch that directly inserts the boundary check. Thus, in Table 13, both L2S-Hanabi and ACS fix more bugs than Table 12.

From the table, we can see that though L2S-Hanabi is not designed for repairing general types of bugs, L2S-Hanabi still achieves the highest precision and the second highest recall. L2S-Hanabi generates six patches less than TBar, losing 2% of recall, but its precision is 30% higher than that of TBar. The results shows L2S-Hanabi keeps a high precision and a relative high recall compared with the existing approaches.

Moreover, we compare the sets of correctly repaired bugs by different approaches, and find that **six** bugs repaired by L2S-Hanabi are not repaired by any existing technique.

5.5.6 RQ9: Influence of each repair template. To analyze the contribution of the three templates, we classified all the generated patches by repair template, shown in Table 14. As we can see, Template 1, Template 2 and Template 3 fix 46.9%, 37.5% and 15.6% of the bugs, respectively. Please note that the evaluation of ACS, Template 2 only repairs 2 bugs (11.8% of all repaired bugs), which are

Table 14. Influence of each repair template

Template	**Correct**	**Incorrect**	**Total**
Template 1	15 (46.9%)	0 (0%)	15 (39.5%)
Template 2	12 (37.5%)	6 (100%)	18 (47.4%)
Template 3	5 (15.6%)	0 (0%)	5 (13.2%)
Total	32	6	38

X (Y): X is the number of patches, Y is its percentage in the same column.

drastically fewer than those repaired by L2S-Hanabi. The result suggests modifying an existing expression is probably more difficult than generating a missed boundary check, and requires an advanced model to handle.

Please recall that Template 3 is newly added and does not exist in ACS. To compare only the synthesis component with ACS, we further calculate how many bugs the first two templates repairs on the 143 conditional bugs and the 224 bugs in Defects4J. The two templates repair 17 bugs with 5 false positives on the 143 conditional bugs (precision: 77%, recall: 12%), and repair 23 bugs with 5 false positives on the 224 bugs of Defects4J (precision: 82%, recall: 10%), significantly outperforming ACS on both the precision and the recall.

Moreover, referring to the results of TBar and SimFix in Table 12 and Table 13, we can find that although the two approaches achieve high recall on the general bugs of Defects4J, their recall and precision sharply degrade on fixing conditional bugs. Especially the precision of the two approaches is reduced by nearly 20%. This suggests that repairing conditional statement is more error-prone and is hard to achieve a high precision while keeping a high recall.

5.6 Discussion on the Repair Capability of L2S-Hanabi

To understand why L2S-Hanabi fails to repair some conditional statement bugs, we randomly sampled 30 conditional bugs that L2S-Hanabi fails to repair, and analyze the reason why L2S-Hanabi fails to repair the bug. There are mainly four reasons.

- *Failure to localize*: The fault localization technique fails to localize the faulty location, or fail to rank the faulty location to a high position. Bugs in this category require better fault localization approaches to repair. 1 (3%) out of 30 bugs belongs to this case.
- *Unique conditional expressions*: Some bugs require to synthesize an expression that is beyond the space of expressions mined from the training set, e.g., calling a rare API that has never been used in the training set. Bugs in this category may be repaired by some forms of abstraction, e.g., generating only the types of the API call rather than a concrete API call. 5 (17%) out of 30 bugs belong to this case.
- *Beyond the repair templates*: Though some bugs require changing only a conditional statement, the change is beyond what our current template supports. For example, inserting an `if` statement to wrap some other statements, or inserting an `if` statement in a loop with a `break` statement. Bugs in this category may be repaired by introducing more templates. 8 (27%) out of 30 bugs belong to this case.
- *Beyond conditional statements*: Some bugs require not only changing a conditional statement, but also some other statements. Such bugs are beyond the scope of L2S-Hanabi and require other mechanism to repair. 10 (33%) out of 30 bugs belong to this case.
- *Beyond the repair attempts limitation*: Although the patches of some bugs are inside the search space, L2S-Hanabi cannot generate the correct expression within top 200. These bugs require better probability estimation to repair. 4 (13%) out of 30 bugs belong to this case.

6 RELATED WORK

6.1 Program Generation from Context

Many approaches have been proposed to generate different kinds of programs, such as imperative programs [5, 15, 23, 36, 50, 52, 57, 72], lambda expressions [15, 50, 69], string manipulations [14, 45], and regular expressions [31], from various contexts, such as natural language description [15, 23, 31, 36, 50, 57, 69, 72], input-output examples [14, 45], and surrounding code [5, 52].

The main difference among L2S and these approaches is that these approaches do not take a specification as input, and only aim to find the most probable program, while L2S aims to find the most probable program that meets a specification, and uses static analysis to prune off programs that are not type-safe or semantically correct. Besides, there are other differences: (1) these approaches follow natural orders such as left to right or top-down grammar expansion, while L2S supports to use expansion rules to define different expansion order; (2) while existing approaches use a fixed search algorithm, usually beam search, L2S allows using different path-finding algorithms by treating the synthesis problem as a path-finding problem; (3) our paper also gives the proof that the probability can be computed from the probabilities of choosing production rules and the probabilities of choosing expansion points can be ignored.

6.2 Classic Program Synthesis

Classic program synthesis [20] takes a specification as input, and returns a program that meets the specification. Different from classic program synthesis, L2S solves the program estimation problem, which aims to find the most probable program that meets the specification.

Essentially, L2S can be viewed as an enumerative program synthesis, where the probabilities are used to guide the order of the program to be enumerated. It is common for enumerative synthesis to use lightweight deductive analysis to prune off program space [8, 20]. The static analysis for pruning off infeasible partial programs in L2S is a general form of lightweight deductive pruning that is compatible with the probability and the expansion rules.

In particular, Lee et al. [34] proposed an approach that uses a probabilistic model to accelerate program synthesis in parallel with our work [67]. Their approach uses A* search by performing a static analysis on the probabilistic model to produce a heuristic function computing the probability upper-bound of the remaining program. Though their goal is different from us, this work is complementary to L2S as it provides an approach for automatically generating a heuristic function for A* search. On the other hand, L2S provides expansion rules to control the expansion order and uses a static analysis on grammar rules to prune off infeasible partial programs. However, their approach of generating a heuristic function does not directly apply to L2S-Hanabi because the static analysis on the probabilistic model requires to enumerate all possible contexts and only works on simple models. It remains as future work to understand how to combine the two approaches.

Another related work, MaxFlash [25], uses a probabilistic model to accelerate FlashFill-style deductive program synthesis [47]. Different from enumerative program synthesis, FlashFill-style deductive program synthesis requires witness functions to decompose a problem into subproblems, and is only applicable to problems where witness functions are available. Furthermore, since the goal of MaxFlash is to accelerate program synthesis rather than finding the most probable program under a partial specification, it allows only a limited form of probabilistic model, the top-down prediction model. In a top-down prediction model, the choice of a rule can only depend on its ancestors but not siblings. In constrast, in L2S the rule can depend on the whole generated partial program, including both its ancestors and siblings.

6.3 Approaches to Program Estimation

In parallel with our work [67], some researchers also studied the program estimation problem [8, 10, 30], which is known as maximal multi-layer specification synthesis [10] or multi-model synthesis [8] in respective literatures. Compared with these approaches, our framework is the first to propose the theory of expansion rules and use expansion rules to control the order of search. Besides this major difference, we also compare other differences in details as follows.

Kalyan et al. [30] proposed to combine dynamic programming with a neural network, where the neural network predicts the most probable subproblems to explore based on the specification of the current problem, such that the result program is more likely to be desirable. Compared with L2S, (1) their approach requires the probability calculation has the structure of dynamic programming: the most probable program can be composed from the most probable subprograms, and thus is difficult to apply to problems where the context is not a decomposable specification, e.g., a natural language description or surrounding code; (2) their approach cannot guarantee to find the most optimal program, where L2S with a suitable search algorithm could guarantee that; (3) their approach uses witness functions to divide a problem into subproblems, which on the one hand requires the users to provide the witness functions, and on the other hand could potentially achieve faster speed by reusing the solutions to subproblems.

Chen et al. [10] view the problem of program estimation as a weighted MAXSMT problem, where the specification of the problem gives hard constraints and a statistical model gives weighted soft constraints about the desired combination of the symbols. For example, a soft constraint can be "'+' is a child of '-' with a weight 100". A MAXSMT solver tries to find a program that satisfies all hard constraints and the subset of soft constraints that gives the maximal weight. Compared with L2S, it is not clear how to interpret the weights of soft constraints probabilistically, and thus this approach does not ensure the returned program has the highest probability, while L2S can ensure this by using an exact algorithm.

Chen et al. [8] proposed an approach that generates a regular expression from both input-output examples and a natural language description. Their approach first uses a statistical model to predict a likely skeleton of a program based on the natural language description, and then uses program synthesis to fill the missing parts in the skeleton to form a complete program. Compared with L2S, the statistical prediction and the program synthesis are completely separated in their approach, and thus their approach does not ensure that the returned program has the highest probability, either.

6.4 Statistical Program Repair

Typical program repair approaches [17] first localize a faulty code snippet with a fault localization approach, then outline a patch space by grammar rules or edit templates, and finally synthesize a new code snippet to replace the buggy one. The patch synthesis component is a key to repair performance. Since tests are partial specifications [49], existing program repair techniques have employed statistical models to help synthesize patches with higher probabilities. Early approaches [41, 56] use binary models that returns a probability for a patch, apply the model to each patch in the space, and sort these patches based on their probabilities. Since the probability of each patch needs to be calculated, this method only allows very simple models. For example, Elixir [56] employs the logistic regression algorithm to ensure fast inference.

ACS [68] is a program repair approach for repairing buggy conditional expressions. It first localizes to a potentially faulty conditional expressions or a potentially missing boundary check, and then synthesizes a new condition to replace the old one or insert a boundary check. In particular, its synthesis component synthesizes a probable condition by two ranking methods: ranking the variables to be checked and ranking the predicates to be used on the variable. As a result, ACS achieves significantly better precision than existing approaches. We are inspired by the fact that ACS first chooses a variable and then a predicate, and propose expansion rules to generalize different orders for expanding a program. L2S-Hanabi also follows the ACS framework and adopts the two main templates used in ACS. The main difference is the replacement of the synthesis component with L2S, with a minor addition of a new template inspired by ACS. On the other hand, L2S is significantly different from the synthesis component of ACS. L2S is a general framework for the program estimation problem and has been applied to both program repair (this paper) and program generation [59, 60], while the synthesis component of ACS is only designed for conditional expressions. L2S uses expansions rules to describe possible ways of decomposing programs into choices and L2S-Hanabi uses a bottom-up rule set that decomposes a condition into two or more choices. On the other hand, ACS only decomposes a condition into two choices. L2S ensures to find the most probable conditions when an exact search algorithm is used while ACS concerns only the ranks of variables and predicates and does not guarantee the probability of the synthesized condition. L2S allows using different search algorithms while ACS uses a greedy algorithm. Finally, L2S introduces an efficient pruning method based on abstract interpretation while ACS does not perform any pruning.

Recently, a series of research attempts [11, 21, 35, 42, 61, 65] have employed neural networks to predict a patch. These approaches takes a localized suspicious line and its surrounding code as input, and predicts a new line to replace the original line as a patch. Compared with L2S-Hanabi, there are several major differences: (1) these approaches first predict a patch and then validates its feasibility by compilation and testing, and do not try to prune off type-incorrect patches as L2S-Hanabi does; (2) these approaches either treat the patches as a sequence of tokens [11, 21, 42, 61, 65] or treat the patches as a sequence of grammar rules [35], while L2S-Hanabi uses expansion rules to control the order of generation; (3) these approaches are trained over a set of existing patches while L2S-Hanabi is trained over existing source code. As our evaluation shows, L2S-Hanabi outperforms these existing approaches in repairing conditional statement bugs. Nevertheless, neural network could learn rich features to better characterize the conditional probability, and may be integrated in L2S in future research.

Finally, several research attempts [4, 26, 38, 53] try to cluster existing patches and abstract a repair template from each cluster. These studies are orthogonal to our work as the extracted repair template could be integrated into the grammar to better guide the generation of the patches.

7 THREATS TO VALIDITY

External validity. The evaluation of L2S-Cond demonstrates that the components in L2S all have significant influence to the overall performance. This conclusion is drawn from one application and thus should be interpreted as “the components are important for some applications” but not “all applications”. For example, for a weakly typed language the type constraint function may not be very useful. Also, though L2S-Hanabi shows consistent performance on two datasets, Defects4J and Bugs.jar, evaluating on more dataset could further evaluate the generalizability of L2S-Hanabi.

Internal validity. The main threat to internal validity is that the implementation of L2S-Cond and L2S-Hanabi may be wrong. However, since it is unlikely accidental fault would lead to correctly synthesized programs, the performance of L2S-Cond and L2S-Hanabi could only be higher if the implementation were wrong.

Construct validity. The main threat to construct validity is that we manually assess the correctness of a patch following the common practice of existing works [7, 24, 27, 43, 55, 64, 66, 68], but the manual analysis may introduce errors [33]. To mitigate this threat, we followed the advice of the existing studies [33, 71], and further evaluated the correctness of the patches on the enhanced test suites generated by EVOSUITE [16]. Furthermore, all the results are open for publicly judgement.

8 CONCLUSION

In this paper we have presented a framework, L2S, for the program estimation problem. L2S treats the program estimation problem as a path-finding problem, uses expansion rules to lay out the search space, prunes infeasible paths early based on abstract interpretation, estimates the probabilities of the program by estimating the probabilities of each choice, and adopts existing search algorithms to solve the problem. The evaluation demonstrates that the components in L2S are all important in solving the program estimation problem, and instantiating L2S could form a useful program repair approach. So far we only instantiate L2S for repairing conditional statements as well as program generation from natural language description [59, 60]. Future work could apply L2S to more applications such as repairing more types of bugs or programming by example.

ACKNOWLEDGMENTS

We thank Rui Wang from Beijing Jiaotong University for her careful proofreading. This work is sponsored in part by the National Key Research and Development Program of China No. 2019YFE0198100, National Natural Science Foundation of China under Grant No. 61922003.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization *(PRDC)*. IEEE Computer Society, Washington, DC, USA, 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools, 2nd Edition.*

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[5] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *ICML*. 2933–2942.

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[7] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *ASE*. https://doi.org/10.1109/ASE.2017.8115674

[8] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *PLDI*.

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[10] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 602–612.

[11] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[13] Hans Peter Deutsch. 2002. *Principle Component Analysis.* Palgrave Macmillan UK.

[14] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 990–998. http://proceedings.mlr.press/v70/devlin17a.html

[15] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 33–43.

[16] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*. ACM, 416–419.

[17] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2755013

[18] Sumit Gulwani. 2016. Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In *IJCAR*. 9–14.

[19] Sumit Gulwani and Prateek Jain. 2017. Programming by Examples: PL meets ML. In *APLAS*.

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

[21] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*.

[22] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[23] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 925–930. https://doi.org/10.18653/v1/d18-1111

[24] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-demand Candidate Generation. In *ICSE*.

[25] Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding dynamic programing via structural probability for accelerating programming by example. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 224:1–224:29. https://doi.org/10.1145/3428292

[26] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.

[27] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *ISSTA*.

[28] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *ICSE*.

[29] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. 437–440.

[30] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rywDjg-RW

[31] Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 826–836.

[32] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*. 213–224. https://doi.org/10.1109/SANER.2016.76

[33] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina S. Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 524–535. https://doi.org/10.1109/ICSE.2019.00064

[34] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 436–449. https://doi.org/10.1145/3192366.3192410

[35] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. DLfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[36] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[37] 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.

[38] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *ESEC/FSE*. 727–739. https://doi.org/10.1145/3106237.3106253

[39] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *ESEC/FSE*.

[40] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *ICSE*. 702–713. https://doi.org/10.1145/2884781.2884872

[41] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312. https://doi.org/10.1145/2837614.2837617

[42] 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.

[43] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (01 Aug 2017), 1936–1964.

[44] Mark F. Medress, Franklin S. Cooper, James W. Forgie, C. C. Green, Dennis H. Klatt, Michael H. O'Malley, Edward P. Neuburg, Allen Newell, Raj Reddy, H. Barry Ritea, J. E. Shoup-Hummel, Donald E. Walker, and William A. Woods. 1977. Speech Understanding Systems. *Artif. Intell.* 9, 3 (1977), 307–316. https://doi.org/10.1016/0004-3702(77)90026-1

[45] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings)*, Vol. 28. JMLR.org, 187–195. http://proceedings.mlr.press/v28/menon13.html

[46] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*. 69–79.

[47] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 107–126. https://doi.org/10.1145/2814270.2814310

[48] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *ICSE*. 254–265. https://doi.org/10.1145/2568225.2568254

[49] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems *(ISSTA)*. 24–36.

[50] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1139–1149.

[51] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis. In *ICSE*. 357–367.

[52] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 419–428. https://doi.org/10.1145/2594291.2594321

[53] 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 *ICSE*. 404–415.

[54] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 10–13.

[55] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *ASE*. IEEE Press. http://dl.acm.org/citation.cfm?id=3155562.3155643

[56] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 648–659.

[57] Chengxun Shu and Hongyu Zhang. 2017. Neural Programming by Example. In *AAAI*. AAAI Press, 1539–1545.

[58] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *SANER*.

[59] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *Thirty-Third AAAI Conference on Artificial Intelligence*.

[60] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2019. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *Thirty-Third AAAI Conference on Artificial Intelligence*.

[61] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[62] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE*. 297–308.

[63] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[64] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *ICSE*.

[65] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 479–490.

[66] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair *(ASE)*. http://dl.acm.org/citation.cfm?id=3155562.3155644

[67] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. 2018. Learning to Synthesize. In *International Genetic Improvement Workshop*. https://doi.org/10.1145/3194810.3194816

[68] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*. https://doi.org/10.1109/ICSE.2017.45

[69] Kun Xu, Lingfei Wu, Zhiguo Wang, Mo Yu, Liwei Chen, and Vadim Sheinin. 2018. Exploiting Rich Syntactic Information for Semantic Parsing with Graph-to-Sequence Model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 918–924. https://doi.org/10.18653/v1/d18-1110

[70] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *TSE* (2017).

[71] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 1–38.

[72] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

[73] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *ICSE*. 272–281.

APPENDIX: A LIST OF THE EXTRACTED FEATURES

Table 15. Context features

Name	Type	Description
filename	String	The enclosing file name
tdname	String	The enclosing class name
mtdname	String	The enclosing method name
mtdmod	Numerical	The modifier information of the enclosing method
mtdln	Numerical	The number of lines of the enclosing method
locnum	Numerical	The number of accessible local variables
paranum	Numerical	The number of the parameters of the method
fldnum	Numerical	The number of fields of the class
allloc	String	The string of concatenating the name of every accessible variable
allloctp	String	The string of concatenating the type of every accessible variable
locintnm	Numerical	The number of integer variables
locfltnm	Numerical	The number of float point variables
locarrnm	Numerical	The number of array variables
allfld	String	The string of concatenating the name of every field
allfldtp	String	The string of concatenating the type of every field
inloop	Boolean	Whether the location is inside a loop
bodyctl	Label	The control flow information inside the block of current location
befsyn	String	The string of concatenating control flow points before current location
bdsyn	String	The string of concatenating control flow points inside the block body of current location
afsyn	String	The string of concatenating control flow points inside the block body of current location
bes N	Label	The N'th control flow point before current location ($0 \leq N \leq 5$).
bds N	Label	The N'th control flow point inside the block of current location ($0 \leq N \leq 3$).
afs N	Label	The N'th control flow point after the block of current location ($0 \leq N \leq 3$).
lv N	Label	The N'th nearest variable from current location ($0 \leq N \leq 3$).
pstmt N	String	The N'th nearest line before current location ($0 \leq N \leq 1$).
nstmt N	String	The N'th nearest line after current location ($0 \leq N \leq 1$).
befcd	String	The nearest if condition before current location.
befpred	String	The E literal of the nearest if condition before current location.

Table 16. Program features for V

Name	Type	Description
varname	String	The name of the variable.
vartype	String	The type of the variable.
vnmlen	Numerical	The length of the name of the variable.
shortvn	Boolean	Whether the variable name is a short name having no meaning.
vnmwds	Numerical	The number of words divided by camel-case in the name.
ltt N	Label	The last N'th letter in the name ($0 \leq N \leq 2$).
wd N	Label	The N'th word by camel-case in the name ($0 \leq N \leq 2$).
isint	Boolean	Whether the variable type is integer.
isflt	Boolean	Whether the variable type is float point.
isarr	Boolean	Whether the variable type is array.
iscoll	Boolean	Whether the variable type is a sub-type of Java Collection.
ispmtarr	Boolean	Whether the variable type is primitive.
prmtandspl	Boolean	Whether the features *shortvn* and *ispmtarr* are both true.
twdl	Label	The last word divided by camel-case in the variable type.
lastassign	Label	The type of the nearest assignment of the variable.
assop	Label	The operation in the nearest assignment of the variable.
assmtd	Label	The method invocation in the nearest assignment of the variable.
assnm	Label	The variable name in the expression of the nearest assignment of the variable.
assnum	Numerical	The assignment time of the variable before the location.
dis	Numerical	The distance between the assignment and the location.
disl10	Boolean	Whether the assignment is near, i.e., $dis \leq 10$.
disl20	Boolean	Whether the assignment is not far, i.e., $dis \leq 20$.
disg20	Boolean	Whether the assignment is far, i.e., $dis \geq 20$.
preassnum	Numerical	The time to be assigned of the variable before the location.
isparam	Boolean	Whether the variable is a parameter of the method.
isfld	Boolean	Whether the variable is a field of the class.
isfnl	Boolean	Whether the variable has the `final` modifier.
isidxer	Boolean	Whether the variable is an array index.
bodyuse	Boolean	Whether the variable is used inside the body of the current if statement.
casted	Boolean	Whether the variable type-casted before current location.
castedtp	Label	The type that the variable to be casted before current location.
outuse	Boolean	Whether the variable is used outside the body of the current if statement.
incondnum	Numerical	The time the variable used in condition in the current method.
filecondnum	Numerical	The time the variable used in condition in the current file.
totcondnum	Numerical	The time the variable used in condition in the project.
lastpre	String	The nearest predicate using the variable.
docexcp	Label	The exception associated with the variable in Javadoc.
docop	Label	The exception operation associated with the variable in Javadoc.
doczero	Boolean	Whether the exception predicate in Javadoc has 0.
docone	Boolean	Whether the exception predicate in Javadoc has 1.
docnon	Boolean	Whether the exception predicate in Javadoc has `null`.
docrange	Boolean	Whether the exception predicate in Javadoc limits a range.
docincode	Boolean	Whether the variable appears in the code fragment of Javadoc.

Table 17. Position features for V

Name	Type	Description
argused	Boolean	Whether the variable position is an argument of a method invocation.
tpfit	Boolean	Whether the variable type fit the current position of the *test*.
occpostime	Numerical	The occurrence time of the variable in current partial program.
used	Boolean	Whether the variable is used in current partial program.

Table 18. Program features for E

Name	Type	Description
pred	String	The string literal of an E.
posnum	Numerical	The number of positions to be expanded.
roottp	Label	The AST node type of the root of an E.
ariop	Label	The last arithmetic operator in the preorder traversal of an AST of an E.
hight	Numerical	The AST height of an E.
mtd	Label	The name of the last method invoked in the preorder traversal of an E AST.
instcof	Boolean	Whether the E contains `instanceof`.
num N	Numerical	The N'th number appears in an E ($0 \leq N \leq 1$).
hasnull	Boolean	Whether the E contains `null`.

Table 19. Position features for L

Name	Type	Description
isroot	Boolean	Whether the L is the AST root of a partial program.
parenttp	Label	The AST node type of the root.
siblingtp	Label	The AST node type of the sibling node.
location	Label	The location of the node in the parent node.
depth	Numerical	The distance from the node to the root.