Learning to Synthesize*

Yingfei Xiong
Key Laboratory of High Confidence Software Technologies (Peking University), MoE; EECS, Peking University
Beijing, China
xiongyf@pku.edu.cn

Bo Wang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE; EECS, Peking University
Beijing, China
wangbo_15@pku.edu.cn

Guirong Fu
School of Mathematical Sciences, Peking University
Beijing, China
fgr079@126.com

Linfei Zang†
State Information Center
Beijing, China
zanglf126@126.com

ABSTRACT

In many scenarios we need to find the most likely program under a local context, where the local context can be an incomplete program, a partial specification, natural language description, etc. We call such problem *program estimations*. In this paper we propose an abstract framework, *learning to synthesis*, or *L2S* in short, to address this problem. L2S combines four tools to achieve this: *rewriting rules* are used to define the search space and search steps, *constraint solving* is used to prune off invalid candidates at each search step, *machine learning* is used to estimate conditional probabilities for the candidates at each search step, and *search algorithms* are used to find the best possible solution. The main goal of L2S is to lay out the design space to motivate the research on program estimation.

We have performed a preliminary evaluation by instantiating this framework for synthesizing conditions of an automated program repair (APR) system. The training data are from the project itself and related JDK packages. Compared to ACS, a state-of-the-art condition synthesis system for program repair, our approach could deal with a larger search space such that we fixed 4 additional bugs outside the search space of ACS, and relies only the source code of the current projects.

ACM Reference Format:
Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. 2018. Learning to Synthesize. In *GI'18: GI'18:IEEE/ACM 4th International Genetic Improvement Workshop , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194810.3194816

*This work is supported by National Key Research and Development Program under Grant No. 2017YFB1001803, and the National Natural Science Foundation of China under Grant No.61725201, 61672045, 61332010.
†The work was done when Linfei was a master student at Peking University.

GI'18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5753-1/18/06…$15.00
https://doi.org/10.1145/3194810.3194816

1 MOTIVATION

In many tasks we need to synthesize a program automatically. A typical category is genetic improvement [21], where the system automatically searches for a program within a space to meet a specific goal, such as performance improvement or bug fixing. Some problems have clearly defined specification: when the program meets the specification, the program is considered as correct. Traditional program synthesis techniques are mainly designed to deal with such problems [9]. However, many problems do not have such a correctness specification [21]. For example, in test-based program repair [27] and program by examples [7], only a set of tests is available to validate the correctness of the patched program. Other related fields are code completion [19] and program synthesis from natural languages [22], where code is generated based on a partial 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 have revealed repairing only for passing the tests often results in incorrect patches. In the latter two scenarios, there is not even a partial 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, we would like to find a program $prog$ that maximizes the conditional probability $P(prog \mid context)$, where $context$ refers to the local context, including a specification, an incomplete program, and/or natural language description. To ease presentation, we refer this particular subproblem of program synthesis as *program estimation*.

However, it is not easy to solve the program estimation problem. First, we need to estimate the conditional probability $P(Prog \mid Context)$. In particular, we need to ensure the estimated probability is consistent with the partial specification, i.e., programs that do not satisfy the specification should have zero probability. Second, we need to locate the program with the maximum probability, which is not easy because the space of possible programs is usually huge.

In this paper we propose an abstract framework, *learning to synthesis*, or *L2S* in short, for program estimation. The main goal of L2S is to lay out the design space to motivate the research on program estimation. L2S solves the program estimation problem by combining four tools: rewriting rules, constraint solving, machine learning, and search algorithms. Rewriting rules are used to define the solution space and to convert the estimation problem into a search problem.

Constraint solving is used to prune off the infeasible candidates at each search step using constraints from the partial specification and generated partial program. Machine learning is used to estimate conditional probabilities for the candidates at each step, and these probabilities can be combined to estimate the probability of a generated program. Finally, a search algorithm is used to solve the search problem, by using the estimated conditional probabilities as hints.

We have instantiated L2S on a automated program repair (APR) system [6, 16], aiming to synthesis correct conditional expressions as patches. We performed an preliminary evaluation of L2S on two projects from the Defects4J benchmark, with 133 defects in total. Comparing to state-of-art APR systems, ACS [28], L2S could deal with a significantly larger search space, leading to 4 additional bugs to be fixed outside the search space of ACS, and relies only the source code of the current projects.

In the rest of the paper, we shall first describe the framework and its implementation in details, then present our preliminary evaluation, finally discuss related work and conclude the paper.

2 FRAMEWORK

2.1 Overview

We first give an overview of L2S by describing an example of estimating a conditional expression. Conditional expressions are a common source of bugs, and existing work [24, 28] has shown that correctly estimating conditions could help repair a significant number of bugs. Now assuming that in the current local context we have two integer variables, "hours" and "value", and we would like to estimate the conditional expression used in the next "if" statement. For demonstration purpose, we assume the condition expression could be described by the following grammar.

$$\mathtt{E} \rightarrow \mathtt{E}\ \text{“}>12\text{”} \mid \mathtt{E}\ \text{“}>0\text{”} \mid \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E} \mid \text{“}\mathtt{hours}\text{”} \mid \text{“}\mathtt{value}\text{”} \mid \ldots$$

The first step of using L2S is to describe the search space using rewriting rules. The simplest way is to derive directly from the grammar, as follows.

$$\begin{array}{rcl} & \Rightarrow & \mathtt{E}^{\mathtt{D}} \\ \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} & \Rightarrow & \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}>12\text{”} \mid \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}>0\text{”} \\ & \mid & \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}+\text{”}\ \mathtt{E}^{\mathtt{D}} \mid \mathtt{E}_{\mathtt{0}} \rightarrow \text{“}\mathtt{hours}\text{”} \mid \ldots \end{array}$$

The first rule is called a *creation rule*, indicating that we create a root node $\mathtt{E}^{\mathtt{D}}$ first. The superscript $\mathtt{D}$ is to indicate that this node needs further downward expansion. Then at each step of the search, we find a node that matches the left hand side (the symbol before $\Rightarrow$) of a rule and expand it into the tree in the right hand side of the rule. The subscript $\mathtt{0}$ in $\mathtt{E}_{\mathtt{0}}$ indicates that this node will replace the original node matched by the left hand side. The search stops when no rule can be applied.

To reduce the search space, we use constraint solving to prune off infeasible rules at each node. At each step, we generate constraints from the context, the already generated AST, the chosen node to expand, and the chosen rewriting rule. Then we put the constraints into a constraint solver to check their satisfiability. If unsatisfiable, this rule is infeasible. For example, a common class of constraints is the type constraints. Since we are generating a Boolean expression, by using the type constraints we know that only the first two rules for expanding $\mathtt{E}^{\mathtt{D}}$ are feasible to expand the root node.

To distinguish the feasible rules at each node, we use machine learning to calculate the conditional probabilities of each rule. The conditional probability has the form $P(Rule|Context, Prog, Node)$, where $Context$ represents the context for the program generation, $Prog$ represents the currently generated AST, $Node$ represents the node chosen to be expanded, and $Rule$ represents the choice of a rule starting from the symbol of $Node$.

L2S does not enforce any concrete machine learning methods, and the user could choose the methods that fit best to the problem. Furthermore, different machine learning methods could be specified for different symbols for best results. To train the models, L2S requires a training set includes pairs of programs and their contexts, and parses the programs to produce the training set at each node, similar to PHOG [1]. For each node in the parsed AST, the chosen production rule is a positive instance, and all other production rules starting from the same symbol are negative instances.

Given the conditional probability of the rule at each step, the probability of the whole program is their product. Please note that a program can be generated in different ways by choosing different nodes to expand at search steps, but the probabilities calculated from different orders will be the same. A detailed proof will be given later.

Now we can estimate the probability of each program, we need to solve the search problem to select the most-likely program. Please note that we cannot simply select the most probable rule at each node, because local optimal does not necessarily lead to global optimal. For example, let us assume at the root node the learned model estimates the following probabilities.

$$\begin{array}{ll} \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}>12\text{”} & 0.3 \\ \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}>0\text{”} & 0.6 \end{array}$$

Please note other options have been pruned off by type constraints. When choosing $\mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}>0\text{”}$ that has the highest probability, the learned model estimates the following probabilities for the newly added $\mathtt{E}^{\mathtt{D}}$ node.

$$\begin{array}{ll} \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \text{“}\mathtt{hours}\text{”} & 0.1 \\ \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \text{“}\mathtt{value}\text{”} & 0.2 \\ \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}+\text{”}\ \mathtt{E}^{\mathtt{D}} & 0.05 \end{array}$$

Combining the two, the most likely expression is `value > 0` which has a probability of 0.12. However, if we choose the other option at the first step, the learned model predicts the following probabilities for the newly added $\mathtt{E}^{\mathtt{D}}$ node.

$$\begin{array}{ll} \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \text{“}\mathtt{hours}\text{”} & 0.8 \\ \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \text{“}\mathtt{value}\text{”} & 0.1 \\ \mathtt{E}^{\mathtt{D}}_{\mathtt{0}} \Rightarrow \mathtt{E}_{\mathtt{0}} \rightarrow \mathtt{E}^{\mathtt{D}}\ \text{“}+\text{”}\ \mathtt{E}^{\mathtt{D}} & 0.05 \end{array}$$

Thus, the combination `hours > 12` actually has a higher probability of 0.24. If we select only the best candidate at each step, we would not be able to produce this expression.

At each step we need to make two decisions. First select a node marked with D, and then select a rule to expand it. For the former, L2S uses a policy that does not depend on the machine-learned models. We require the policy not to depend on the learned models so that we can also use this policy in preparing the training set. For the latter, L2S uses a search algorithm to find a set of proper rules at all steps to maximize the probability of the synthesized program. L2S does not enforce any particular policy or search algorithm and the user could choose those suitable to the problem. For example, a

policy could be used in our example is to expand the nodes from left to right, and a search algorithm could be used is beam search [15]. The beam search is a greedy algorithm that keeps at most k ASTs that have the highest probability. At each step, the algorithm constructs a new set of ASTs by expanding the next node with the k best rules in each AST, and keeps k new ASTs with the highest probability. In the above example, if we set k to 2, we shall get `hours > 12`.

In the rest of this section we discuss several key issues in L2S.

2.2 Rewriting Rules

In the previous subsection we have seen how to directly map grammar rules into rewriting rules. L2S does not confine a particular way of defining the search space and the users could define in any way they want. For example, directly map grammar rules into rewriting rules expand the program in a top-down order. We may also start from a leaf node, and expand the program upwardly to the root node, by using the following set of rules.

$$\begin{array}{lcll} & \Rightarrow & \text{“value”}^{\mathsf{U}} \mid \text{“hours”}^{\mathsf{U}} & \\ \text{“hours”}^{\mathsf{U}}_{\emptyset} & \Rightarrow & \mathsf{E}^{\mathsf{U}} \rightarrow \text{“hours”}_{\emptyset} & \\ \text{“value”}^{\mathsf{U}}_{\emptyset} & \Rightarrow & \mathsf{E}^{\mathsf{U}} \rightarrow \text{“value”}_{\emptyset} & \\ \mathsf{E}^{\mathsf{U}}_{\emptyset} & \Rightarrow & \mathsf{E}^{\mathsf{U}} \rightarrow \mathsf{E}_{\emptyset} \text{ “ > 12”} \mid \mathsf{E}^{\mathsf{U}} \rightarrow \mathsf{E}_{\emptyset} \text{ “ > 0”} & (1) \\ & \mid & \mathsf{E}^{\mathsf{U}} \rightarrow \mathsf{E}_{\emptyset} \text{ “ + ”} \mathsf{E}^{\mathsf{D}} \mid \mathsf{E}_{\emptyset} & \\ \mathsf{E}^{\mathsf{D}}_{\emptyset} & \Rightarrow & \mathsf{E}_{\emptyset} \rightarrow \mathsf{E}^{\mathsf{D}} \text{ “ > 12”} \mid \mathsf{E}_{\emptyset} \rightarrow \mathsf{E}^{\mathsf{D}} \text{ “ > 0”} & \\ & \mid & \mathsf{E}_{\emptyset} \rightarrow \mathsf{E}^{\mathsf{D}} \text{ “ + ”} \mathsf{E}^{\mathsf{D}} \mid \mathsf{E}_{\emptyset} \rightarrow \text{“hours”} \mid \ldots & \end{array}$$

To mark a node that requires upward expansion, we use another annotation U. This rule set starts by creating a variable node, and then expand upwardly. When node E_{U} is generated, the rule $\mathsf{E}^{\mathsf{U}}_{\emptyset} \Rightarrow \mathsf{E}_{\emptyset}$ could stop the upward expansion. Also, we need the top-down rules to expand the E_{D} nodes generated during the process.

To ensure that we construct one AST, we require only one application of creation rules. We call this “one-tree” expansion. L2S also supports “multi-tree” expansion, where several ASTs could be constructed independently and then connected together. The basic idea is to introduce *connecting rules* that connects different subtrees. For example, the following rule set performs a purely bottom-up construction of the expression.

$$\begin{array}{lcl} & \Rightarrow & \text{“value”}^{\mathsf{U}} \mid \text{“hours”}^{\mathsf{U}} \\ \text{“hours”}^{\mathsf{U}}_{\emptyset} & \Rightarrow & \mathsf{E}^{\mathsf{U}} \rightarrow \text{“hours”}_{\emptyset} \\ \text{“value”}^{\mathsf{U}}_{\emptyset} & \Rightarrow & \mathsf{E}^{\mathsf{U}} \rightarrow \text{“value”}_{\emptyset} \\ \mathsf{E}^{\mathsf{U}}_{\emptyset} & \Rightarrow & \mathsf{E}^{\mathsf{U}} \rightarrow \mathsf{E}_{\emptyset} \text{ “ > 12”} \mid \mathsf{E}^{\mathsf{U}} \rightarrow \mathsf{E}_{\emptyset} \text{ “ > 0”} \mid \mathsf{E}_{\emptyset} \\ (\mathsf{E}^{\mathsf{U}}_{\emptyset}, \mathsf{E}^{\mathsf{U}}_{1}) & \Rightarrow & E^{U} \rightarrow E_0 \text{ “ + ”} E_1 \end{array}$$

The last rule is a connection rule. This rule matches two nodes, each in a different AST tree, and connects them using the trees in the right hand sides. The notes with subscripts on the right hand side will replace the nodes with the same subscripts on the left hand side. Please note that we can only connect nodes in two ASTs but not two nodes in the same AST, as ill-formed tree will be generated.

In the above process we derive a set of rewriting rules from the grammar rules. We may also design the rewriting rules freely without referencing the grammar. For example, in the application of genetic improvement, we may directly use rewriting rules to describe the changes on the program. Based on our experience, using a different set of rewriting rules may significantly affects overall performance. It is up to the user to select a set that are most suitable to the target problem.

In particular, we concern unambiguous set of rules. Given an AST tree and a set of rewriting rules R, in general multiple sequences of rule applications may generate the tree. The rule set R is *unambiguous* if and only if any program can only be expanded from the set of rule applications (which can be applied in different orders). The rule sets we have seen above are all unambiguous. If we add a rule $\mathsf{E}^{\mathsf{U}} \Rightarrow \mathsf{E}^{\mathsf{U}} \rightarrow \mathsf{E}^{\mathsf{D}}$ “ + ” $\mathsf{E}_{\emptyset}$ to the rule set (1), the rule set becomes ambiguous. The property of unambiguousness is important as it allows the calculation for the probability of an AST, as shown later.

2.3 Constraint Generation

L2S introduces a structural way of generating the constraints based on the rewriting rules. Given a set of rule applications, variables are generated from nodes while constraints are generated from the rewriting rules and/or the context. To demonstrate, let us consider the the expression `hours > 12`, where two rewriting rules are used to generate the expression.

$$\mathsf{E}^{\mathsf{D}}_{1} \Rightarrow \mathsf{E}_{1} \rightarrow \mathsf{E}^{\mathsf{D}}_{2} \text{ “ > 12”} \quad (2)$$

$$\mathsf{E}^{\mathsf{D}}_{2} \Rightarrow \mathsf{E}_{2} \rightarrow \text{“hours”} \quad (3)$$

We start from the type constraints. Type constraints can be generated using the standard type inference algorithms. First, each node n in the tree generates a type variable $T[n]$, which is an enumeration of types. Then the following two constraints are generated from the two rewriting rules.

$$\mathsf{T}[\mathsf{E}_2] = \texttt{Int} \ \textit{//generated from rule (2)}$$

$$\mathsf{T}[\mathsf{E}_2] = \mathsf{T}[\text{“hours”}] \ \textit{//generated from rule (3)}$$

And the context gives us the following constraint.

$$\mathsf{T}[\text{“hours”}] = \texttt{Int}, \ \mathsf{T}[\mathsf{E}_1] = \texttt{Boolean}$$

In this case the constraints are satisfiable, so there is no type error. However, if we try to expand E_1 with $E \rightarrow$ “*hours*”, the constraint will be unsatisfiable and we know this expansion is infeasible.

Similarly, in the scenarios of test-based program repair and programming by example, we can generate variables from the nodes to represent their values in test executions, and generate constraints based on program semantics and the test cases. In this way, if a partial program could not satisfy a test case, we could know its infeasibility early.

Another interesting type of constraints is the size constraints. In many usage scenarios, we would like to limit the size of the generated tree to avoid searching a too large space. Let us assume the size is defined as the number of nodes in the tree, we can first statically calculate the lower bound for expanding each symbol using the following formulas. For simplicity, we assume no connecting rules.

$$\begin{array}{lcl} size_s(symbol^A) & = & min(\{size_t(tree) \mid \text{exists rule } symbol^A \Rightarrow tree\}) \\ size_s(symbol) & = & 1 \\ size_n(node) & = & size_s(s_{node}) \text{ where } s_{node} \text{ is the symbol of } node \\ size_t(tree) & = & sum(\{size_n(n) \mid \text{for each node } n \text{ in } tree\}) \end{array}$$

The above formulas contain recursion and cannot be computed directly. We need to use a fixed-point algorithm to find the smallest fixed point. Then given an AST t, we require $size_t(t)$ is smaller than the limit.

Please note that L2S requires that the generated constraints are solvable by a constraint solver. In particular, the constraint solver should support incremental solving, such that we can efficiently check multiple candidate rules.

2.4 Machine Learning

To use L2S, the user should specify, for each left hand side, the machine learning algorithm and the functions to extract features from the generated tree and the context, and provides a training set consisting of programs and their contexts. For each program in the training set, L2S finds the sequence of rule applications to generate the program based on the policy of choosing symbols to expand, and produces a set of training set using the feature extract functions. When the rewriting rules are a direct mapping from the grammar rules, the standard parsing algorithms can be used to find the the sequence of rule applications, otherwise the user needs to specify how to find the sequence of rules.

2.5 Probability of a Program

In this subsection we discuss how to calculate the probability of a program. When the rewriting rules are unambiguous, the probability of the program, $P(Prog \mid Context)$, is the product of the conditional probabilities[1], $P(Rule \mid Context, Prog, Node)$, of each rule choice made along any generation process.

Now we show why the calculation is correct. Since $Context$ is always available as a condition, we ignore $Context$ in the discussion. First, each program $prog$ is decided by the sequence of rule applications to create the program, including the choices of node to expand $n_1, n_2, \ldots, n_m$ and the choice of rule at each node $r_1, r_2, \ldots, r_m$. Thus, $P(prog) = P(n_1, \ldots, n_m, r_1, \ldots, r_m)$. Now let us assume the nodes are expanded in the order $(i_1, i_2, \ldots, i_m)$, and the tree after each node expansion are $(prog_1, \ldots, prog_m)$ where $prog_m = prog$. We have

$$\begin{aligned} P(prog) &= P(n_1, \ldots, n_m, r_1, \ldots, r_m) \\ &= P(n_{i_1})P(r_{i_1} \mid n_{i_1})P(n_{i_2} \mid n_{i_1}, r_{i_1}) \ldots P(r_{i_m} \mid \\ &\quad n_{i_1}, \ldots, n_{i_m}, r_{i_1} \ldots, r_{i_{m-1}}). \end{aligned}$$

We notice the probabilities $P(n_{i_1})$, $P(n_{i_2} \mid n_{i_1}, r_{i_1})$, $\ldots$,$P(n_{i_m} \mid n_{i_1}, \ldots, n_{i_{m-1}}, r_{i_1}, \ldots, r_{i_{m-1}})$ must all be 100% because the corresponding nodes, $n_{i_1} \ldots n_{i_m}$, have been generated and must be expanded in the process. We also notice that

$$P(prog_k) = P(n_{i_1}, \ldots, n_{i_k}, r_{i_1}, \ldots, r_{i_k}),$$

then we have

$$P(prog) = P(r_{i_1} \mid n_{i_1})P(r_{i_2} \mid prog_1, n_{i_2}) \ldots P(r_{i_m} \mid prog_{m-1}, n_{i_m}),$$

which is the product from the probability of each rule choice along the order. Since the order is arbitrarily chosen, any order could lead to the same probability.

Please note that we rely on machine-learned models to estimate the conditional probability. However, many discriminative machine-learned models do not calculate probabilities. In addition, when the rule set is ambiguous, we cannot calculate the probability in this way. In these cases, the user needs to provide a fitness function to tell L2S that how to score programs based on the outputs of the machine-learned models.

[1] To facilitate the presentation, we assume that a creation rule application also expands a pseudo node.

3 ESTIMATING CONDITIONS

To understand the potential of this framework, we instantiate L2S framework for conditional expression synthesis. The instantiated synthesizer is called *L2S-E*. As studied by Victor et al. [24], in Defects4J [11], a widely used real-world Java bug dataset, 42.78% bugs are related to conditional blocks, which are the most prevalent category. Estimating conditional expressions has multiple potential usage scenarios, such as bug fixing, bug detecting, and code completion. In this paper we mainly focus on bug fixing, in which case we need to estimate a new condition to replace the localized buggy condition.

Our instantiation perform cross-project training to complete a missing conditional expression in a project. The input to our implementation is the Java source code of a project, where one conditional expression is missing. Our implementation first uses the source code of the target project to train a set of prediction models, and then based on these models to search for a conditional expression at the target location. For simplicity, we consider only conditional expressions without logic operators ("`&&`", "`||`", and "`!`"). Conditional expressions such as `a && !b` will be treated as two expressions `a` and `b`.

3.1 Syntax

$$\begin{aligned} \mathtt{E} &\rightarrow \tau_1(\mathtt{V}_1, \ldots, \mathtt{V}_{k_1}) \mid \ldots \mid \tau_n(\mathtt{V}_1, \ldots, \mathtt{V}_{k_n}) \\ \mathtt{V} &\rightarrow \mathtt{var}_1 \mid \ldots \mid \mathtt{var}_m \end{aligned}$$

Figure 1: The Meta Syntax of Conditional Expression

$$\begin{aligned} \mathtt{E} &\rightarrow \mathtt{V}\ \text{“}\mathtt{>\ 12}\text{”} \mid \mathtt{V}\ \text{“}\mathtt{>\ 0}\text{”} \mid \mathtt{V}\ \text{“}\mathtt{+}\text{”}\ \mathtt{V}\ \text{“}\mathtt{>\ 0}\text{”} \mid \ldots \\ \mathtt{V} &\rightarrow \text{“}\mathtt{hours}\text{”} \mid \text{“}\mathtt{value}\text{”} \mid \ldots \end{aligned}$$

Figure 2: A Concrete Syntax of Conditional Expression

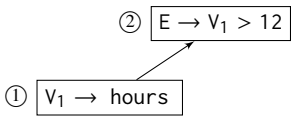
Figure 1 shows the meta grammar for our condition synthesis. The meta grammar will be instantiated using the data in the training set and in the local context. Figure 1 shows an instantiated grammar. More concretely, the non-terminal E expands to an expression with variables represented by non-terminal V. In the right hand side, each τ_i represents a conditional expression used in the training dataset. The non-terminal V expands to a variable in the local context of the conditional expression. Compared with the running example in the previous section, this grammar flattens the hierarchy of E and uses a two-level expansion: the first level expands E to an expression that contains only non-terminal V, and the second level expands each V to a variable. This goal of this flattening is to simplify the construction of machine-learning models.

In our instantiation we use one-tree bottom-up order to complete the conditional expression. More concretely, Figure 1 shows the rewriting rule we derive from the syntax to synthesize the conditional expressions. We first predict the left most variable, which is a leaf of an AST. Then we upward predict an expression as the parent of

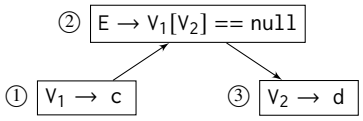
the variable. At last we predict the remaining variable downward. Finally we get an conditional expression. Figure 4 further depicts two examples that we synthesize using this order. In this figure, the circled numbers indicate the order of synthesis while the arrows show the structure of the AST.

$$\begin{array}{rcl} & \Rightarrow & \mathsf{V}^{\mathsf{U}} \rightarrow \mathtt{var}_1 \mid \cdots \mid \mathsf{V}^{\mathsf{U}} \rightarrow \mathtt{var}_m \\ \mathsf{V}^{\mathsf{U}}_1 & \Rightarrow & \mathsf{E} \rightarrow \tau_1(\mathsf{V}_1, \mathsf{V}^{\mathsf{D}}_2 \cdots, \mathsf{V}^{\mathsf{D}}_{k_1}) \mid \cdots \\ & \mid & \mathsf{E} \rightarrow \tau_n(\mathsf{V}_1, \mathsf{V}^{\mathsf{D}}_2, \cdots, \mathsf{V}^{\mathsf{D}}_{k_n}) \\ \mathsf{V}^{\mathsf{D}}_1 & \Rightarrow & \mathsf{V}_1 \rightarrow \mathtt{var}_1 \mid \ldots \mid \mathtt{var}_m \end{array}$$

Figure 3: The Meta Rewriting Rules



(a) *hour* > 12



(b) *c*[*d*] == *null*

Figure 4: AST Examples

3.2 Constrains

In our instantiation we use type constraints to filter out infeasible options. Our implementation reads the type declarations and variable declarations from the source file of the target project, and generates type constraints based on the variable and types used in the expressions. An example of the type constraints has already been given in Section 2.3. Please note that here we do not need size constraint since the size of the conditional expressions have already been confined by the conditional expressions in the training set. We also do not use value constraint as in this scenario, value constraints seldom reduces the size of the search space.

3.3 Machine-Learned Models

There are three set of rewriting rules in Figure 3, each sharing the same left hand side. To assign probabilities to the rules in each set, we need to build three prediction models, one for each set. For the ease of presentation, we call the three sets as creation rules, expression rules, and variable rules, and call the three models as the creation model, the expression model, and the variable model.

In the creation model and the variable model, we need to predict the probabilities of variables. In the expression model, we need to predicate the probabilities of expressions. While the expressions are extracted from the training set, the variables are extracted from the local context and may not be available in the training set. As a result, we need to use different machine learning models to deal with them. For expressions, we consider each expression as a class and train a multiclass classifier to predict the probability of each class. For variables, we consider mapping each variable to a feature vector, and train a binary classifier to predict the probability of this variable.

3.3.1 Feature Engineering. Our implementation uses four set of features, as detailed below.

Context Features Context features are extracted from the context of condition to be synthesized.

Variable Features Variable features are extracted from a variable to represent different aspects of the variable.

Expression Features Expression features are extracted from an expression to represent different aspects of the expression.

Position Feature This feature is mainly used by variable model to indicate the index of the variable being considered.

The features used by the three models are the combinations of the above four sets. The creation model uses the context features to represent the context and the variable features to represent the current chosen variable. The expression model uses the context features to represent the context and the variable features to represent the variable that has been chosen. The variable model uses all the four sets of features, where the context features are used to represent to the context, the variable features are used to represent the previous variable in the expression, the expression features are used to represent the chosen expression, and the position feature is used to indicate the position of the variable being considered.

In many of the features, we need to encode names, such as variable name, method name, type name, etc. One important property we would like to achieve is to let similar names have similar encodings. For example, "length" is often abbreviated into "len", and we would like to give a similar encoding to them. To achieve this, we extract a bag containing 2-gram of characters from the names, and then uses a vector to represent the bag. The vector has $n \times n$ dimensions, where n is the number of characters that can be used in names. Each dimension in this vector represents the occurrence count of a corresponding 2-gram. For example, a variable "len" contains two 2-grams, "le" and "en", and then we have a vector where the dimension of "le" and "en" are 1 and all other dimensions are 0. Finally, to reduce dimensions, we apply principle component analysis [4] on our training set to map the name vectors to vectors with no more than 20 dimensions.

Below we give details about the four sets of features.

Context features Context features include class information, method information and code structure information.

Class information indicates features of the containing Java class, includes basic information such as package name, class name, field names (all field names encoded as one vector) and field types, and complexity features such as inheritance hierarchy depth, class length and method number.

Method information includes the features of the enclosing method, such as its name, return type, modifiers, lines of method body, parameters and etc.

Code structure information captures the syntax structure at a certain line of a Java program, including token vectors [26] before and after conditional expression.

Variable features Variables features mainly consist of four aspects: naming information, type information, definition information, documentation information, definition information and usage information.

Naming information captures the information in the variable name. First, we encode variable names as vectors using the method mentioned above. Second, we argument this vector with the length of name, the last word in the name, etc.

Type information includes features related to the type of the variable. First, the type name is encoded as a vector as mentioned above. Second, we argument the vector with a classification of the types. We classify the type names into integer type (`short, int, lang`), float point type (`float, double`), array type (arrays), collection type (subclasses of `List`, `Set` and `Map`), string related type (`char, String, StringBuffer`) and other type, and use an integer to represent each type. Third, we argument other features, such as a Boolean feature to indicate whether the name of variable is contained in its type, e.g., `map` is contained in `Hashmap`.

Definition information is extracted from the declaration of a variable, including whether it is changeable (`final`), whether it is static (`static`), whether it is loop index/iterator and its initialization value if available, the distance between the variable definition and the conditional expression.

For some well documented projects, their document contains plenty of useful information. We analyze whether this variable is mentioned in specific patterns in the Java document, such as throwing an exception.

Definition information concerns how the value in the variable is defined. We extract at most k places where the variable would be defined (a variable may be defined in multiple places following different paths), each with a feature representing the type of the expression producing the value.

Usages information concerns how the variable is used, including features such as how many times the variable is used in other if conditional expressions in the same method, the same class, or the whole project.

Expression features Expression features are extracted from an expression (possibly containing nonterminals), such as how many variables it contains, the expected type of the first k variables, invoked method name, used comparable operators, arithmetic operators and numbers.

Position feature The last category includes only one feature indicating the index of the variable to be expanded.

3.3.2 Model Training. In our implementation we utilize the Gradient Boosting Tree algorithm to train the three models. Tree-based algorithms are suitable for our task, because they can handle unbalanced data easily and do not need complexly preprocess on training set. We select XGBoost [3], which a widely used implementation of the algorithm.

3.4 Search Algorithms

To solve the search problem, we need to decide the policy for selecting non-terminal and the search algorithm for selecting the rule. For non-terminals, we simple expand the non-terminals from left to right. For rules, we use the beam search algorithm. After the creation rules, we keep the top 5 results, and after expanding the expression, we keep the top 200 results.

When applying our approach to repair conditional expressions, we also use anti-patterns [25] to disable expressions that easily lead to incorrect but plausible repairs. In our implementation one anti-pattern is used: `obj != null`. This pattern is disabled because such expressions usually evaluated as `true`, which can easily generate plausible conditions.

4 PRELIMINARY EVALUATION

4.1 Experimental Setup

We have performed a preliminary evaluation of L2S-E. Our evaluation is based on two projects from the Defects4J [11] benchmark. Defects4J is a benchmark of real-world defects in large software projects. The two projects we used are apache-commons-math and joda-time, both containing a rich set of conditional expressions. The two subjects have 133 bugs in total. Table 1 manifests the details of the subjects.

The evaluation consists of two experiments. In the first experiments, we took two versions of the two projects, math-12 and time-11. The two versions are selected because they are the largest programs in all trained subjects in the second experiment. For each version, we randomly selected 10% conditions as testing set, and the rest of the conditional expressions as training set. Then we trained L2S-E using the train set, and used L2S-E to estimate the conditions in the test set.

In the second experiment, we replaced the condition synthesizer component in ACS [28] with L2S-E to repair the defects in the benchmark. ACS is a program repair system that focuses on repairing incorrect conditions. It either inserts new if statements for dealing with boundary cases, or arguments existing if conditions with additional Boolean expressions. In both repair patterns, conditional expressions need to be synthesized and ACS originally uses a synthesizer that query the whole GitHub repository database to predict the most-likely expression. In our experiments we replace the synthesizer with L2S-E. Furthermore, while in the second case ACS augments an existing if condition, our modified ACS directly replaces the original condition for more readable results. Please note that our space of conditions is also much larger than ACS.

Table 1: Statistics of the Used Subjects of Defects4J

Project	KLoc	Test Cases	Defects	ID
Commons-Math	104	3602	106	Math
Joda-Time	81	4130	27	Time
Total	185	7732	133	-

To save training time, instead of training an estimator for each bug, we used the same trained estimator for a set of bugs that are close in time. More concretely, we trained an estimator using the project source containing the first bug in a calendar year, and fixed the later bugs based on the estimator. Table 2 gives the details of the training versions, the corresponding repaired versions, the covered calendar year and the training time of the version. Please note the larger the bug ID, the earlier the project version. Owing to the increasing scale of the programs, Commons-Math's training time ranges from one

Table 2: Training Versions

Training	Repaired	Year	Training Time
Math12	Math1-Math12	2013	35m
Math37	Math13-Math37	2012	24m
Math59	Math38-Math59	2011	10m
Math75	Math60-Math75	2010	5m
Math94	Math76-Math94	2009	3m
Math102	Math95-Math102	2008	2m
Math104	Math103-Math104	2007	2m
Math106	Math105-Math106	2006	1m
Time11	Time1-Time11	2013	5m
Time17	Time11-Time17	2012	5m
Time24	Time18-Time24	2011	5m
Time27	Time25-Time27	2010	4m

Table 3: Predict Results

Project	Top 1 Precision	Top 10 Precision	Top 50 Precision
Math12	37.8%	62.2%	69.9%
Time11	48.9%	71.2%	75.5%%
Average	43.5%	66.7%	72.7%

Table 4: Overall Comparison with Existing Techniques (Correct / Incorrect)

Technique	Commons-Math	Joda-Time	Total
L2S-E	9 / 12	2 / 4	11 / 16
ACS	12 / 4	1 / 0	13 / 4
Nopol	1 / 20	0 / 1	1 / 21

minute to thirty five minutes, while the Joda-Time's spans from four minutes to five minutes. By the aforementioned strategy, the average training time for each bug is acceptable.

All the experiments ran on a personal computer with Intel Core i7-6700 3.4GHz CPU, 16G memory, Ubuntu 16.04LTS and JDK 1.7. We used 30 minutes as the timeout for each defect, same as the experiments of ACS [28].

4.2 Experimental Results

Table 3 presents the experiment of predicting conditions within a project. The results show that the average precision can be 43.5% for top one, 66.7% for top ten, and 72.7% for top fifty. The result is consistent with existing studies [10] that the code is repetitive and predictable.

We compare our result with ACS [28] and Nopol [13, 29], which are state-of-art techniques focusing on if-statement repair. Table 4 presents the overall comparison results in terms of number of correctly and incorrectly fixed defects of the two subjects. We manually checked each patch of L2S-E, and only considered a patch as *correct* when it is semantically equivalent to the developer-provided patch. According to Table 4, we can find that ACS repaired the most defects correctly with the least incorrectly repaired defects. L2S-E correctly repaired two defects less than ACS while generated more wrong patches. Nopol repaired the least defects with the most wrong patches. Please note that, while ACS requires the whole Github repository as backend, L2S-E and Nopol requires only the source code of the current project.

Table 5 presents the repair results in detail. L2S-E has 4 patches that the others cannot repair. Further studying the four patches we found that all these patches are outside the search space of ACS. In order to minimize the wrong patches, ACS uses a very small search space without method calls or arithmetic operations. On the other hand, the search space of L2S-E is much larger including all predicates appeared in conditional expressions.

Another observation from Table 5 is that L2S-E performs worse when the bug ID grows. This is because the large the bug ID, the earlier the project version is in Defects4J. As a result, the training set becomes much smaller than the bug ID grows. On the other hand, ACS queries the whole Github repository and is not affected by the bug IDs. This finding suggests that incorporating conditions from related projects into the training set may be a potential way to improve the performance of L2S-E in future.

All the patches and analysis of the uniquely fixed patches are available online[2].

5 RELATED WORK

Several studies try to build prediction model for code, either based on probabilistic CFG [1] or based on graph models [18]. Different from us, these approaches focus on predicting the next element in the model rather than a whole AST.

Several studies focus on generating API usage code snippet [2, 22]. Generally, these approaches first try to extract abstract patterns of API usage from code, find the pattern most relevant to the query, and then map the pattern back into code. Different from these approaches, L2S is guided by rewriting rules and uses machine learning to select a rule at each step.

In the domain of program synthesis, recently several attempts [8, 17, 23] have been made to also use machine learning to generate programs under the guidance of a syntax. Compared with those approaches, which are designed to solve a particular problem, L2S is designed as a general framework for laying out a design space of program estimation. This results in two main differences. First, each of these approaches covers only part of the issues discussed in this paper. For example, all these approaches use top-down expansion and do not consider other orders. Second, some of these approaches rely on properties of their target problem and cannot be easily generalized to other problems. For example, in the paper [8], each choice at a search step should be able to transform the input for use in the next step, which does not hold in general.

Program synthesis techniques have been used in different program repair approaches [5, 12, 14, 20]. However, the program synthesis techniques employed are mainly traditional ones and thus may easily generate incorrect patches in program repair, which, as discussed before, is a program estimation problem.

6 CONCLUSION

In this paper we have seen a framework, learning to synthesize, for program estimation. From practical perspective, the framework allows users to design an algorithm to solve a concrete program

[2] https://github.com/wangbo15/L2S-PATCHES-GI

Table 5: Detailed Analysis of Patches

Bug ID	L2S-E	ACS	Nopol
Math2	Incorrect	–	–
Math3	Correct	Correct	–
Math4	Correct	Correct	–
Math5	Correct	Correct	–
Math25	Correct	Correct	–
Math28	Incorrect	Incorrect	–
Math32	**Correct**	–	Incorrect
Math33	**Correct**	–	Incorrect
Math35	Correct	Correct	–
Math40	–	–	Incorrect
Math41	Incorrect	–	–
Math42	–	–	Incorrect
Math46	Incorrect	–	–
Math48	Incorrect	–	–
Math49	–	–	Incorrect
Math50	Incorrect	–	**Correct**
Math57	–	–	Incorrect
Math58	–	–	Incorrect
Math61	Correct	Correct	–
Math63	**Correct**	–	–
Math64	Incorrect	–	–
Math69	–	–	Incorrect
Math71	Incorrect	–	Incorrect
Math78	–	–	Incorrect
Math80	–	–	Incorrect
Math81	–	Incorrect	Incorrect
Math82	–	**Correct**	Incorrect
Math85	Incorrect	**Correct**	Incorrect
Math87	–	–	Incorrect
Math88	–	–	Incorrect
Math89	–	**Correct**	–
Math90	–	**Correct**	–
Math97	–	Incorrect	Incorrect
Math99	Incorrect	**Correct**	–
Math104	–	–	Incorrect
Math105	–	–	Incorrect
Time1	Incorrect	–	–
Time11	–	–	Incorrect
Time15	Correct	Correct	–
Time18	Incorrect	–	–
Time19	**Correct**	–	–

"Correct" means the patch is success. "Incorrect" means the patch is a wrong patch. "–" means there is no patch generated. The bold "Correct" word means it is only fixed by one approach.

estimation problem by instantiating the components in the framework. For academic perspective, L2S lays out a design space of program estimation, which would hopefully facilitate and inspire new research in this area.

REFERENCES

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

[2] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *ICSE*. 782–792.

[3] 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.

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

[5] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. ACM, 85–91.

[6] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* (2017).

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

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

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

[10] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software *(ICSE '12)*. 837–847.

[11] 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.

[12] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604.

[13] 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.

[14] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*.

[15] 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. DOI: http://dx.doi.org/10.1016/0004-3702(77)90026-1

[16] Martin Monperrus. 2017. *Automatic Software Repair: a Bibliography*. Technical Report. 1–24 pages. DOI: http://dx.doi.org/10.1145/3105906

[17] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. *CoRR* abs/1703.05698 (2017). arXiv:1703.05698

[18] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-Based Statistical Language Model for Code. In *ICSE*. 858–868.

[19] 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.

[20] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *ICSE*. 772–781.

[21] Justyna Petke, Saemundur Haraldsson, Mark Harman, William Langdon, David White, and John Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* PP, 99 (2017), 1–1.

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

[23] Rishabh Singh and Pushmeet Kohli. 2017. AP: Artificial Programming. In *SNAPL*. 16:1–16:12.

[24] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. *CoRR* abs/1801.06393 (2018). arXiv:1801.06393

[25] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-Based Program Repair. In *FSE*.

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

[27] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. 364–374.

[28] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*.

[29] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2016).