KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair

Nan Jiang
Purdue University
West Lafayette, USA
jiang719@purdue.edu

Thibaud Lutellier
University of Alberta[1]
Alberta, Canada
lutellie@ualberta.ca

Yiling Lou
Fudan University[2]
Shanghai, China
yilinglou@fudan.edu.cn

Lin Tan
Purdue University
West Lafayette, USA
lintan@purdue.edu

Dan Goldwasser
Purdue University
West Lafayette, USA
dgoldwas@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00111

***Abstract*—Automated Program Repair (APR) improves software reliability by generating patches for a buggy program automatically. Recent APR techniques leverage deep learning (DL) to build models to learn to generate patches from existing patches and code corpora. While promising, DL-based APR techniques suffer from the abundant syntactically or semantically incorrect patches in the patch space. These patches often disobey the syntactic and semantic *domain knowledge* of source code and thus cannot be the correct patches to fix a bug.**

We propose a DL-based APR approach *KNOD*, which incorporates domain knowledge to guide patch generation in a *direct and comprehensive* way. KNOD has two major novelties, including (1) a novel *three-stage tree decoder*, which *directly* generates Abstract Syntax Trees of patched code according to the inherent tree structure, and (2) a novel *domain-rule distillation*, which leverages syntactic and semantic rules and teacher-student distributions to explicitly inject the domain knowledge into the decoding procedure during *both the training and inference phases*.

We evaluate KNOD on three widely-used benchmarks. KNOD fixes 72 bugs on the Defects4J v1.2, 25 bugs on the QuixBugs, and 50 bugs on the additional Defects4J v2.0 benchmarks, outperforming all existing APR tools.

***Index Terms*—Automated Program Repair, Abstract Syntax Tree, Deep Learning**

I. Introduction

Since developers spend nearly half of their time fixing bugs (49%±39%) [1], support to help developers fix bugs is in high demand. Automated program repair [2]–[4] exactly provides such support, which generates patches for buggy programs with little manual effort to improve software reliability and reduce software development costs. With the rapid development of deep learning, recent learning-based APR techniques [5]–[12] leverage advanced DL techniques to generate patches by learning from existing code corpora. DL-based APR often formulates APR as the translation from the given buggy code to the correct one and adopts neural machine translation (NMT) techniques. They typically follow an encoder-decoder architecture, where the encoder first embeds the buggy code, while the decoder generates the patched code iteratively. The generated patches are then validated against test cases.

One major challenge of DL-based APR is that invalid (i.e., syntactically or semantically incorrect) patches dominate in the patch space [5], [7], [8], [10], which hurts the effectiveness and efficiency of patch generation. This challenge is exacerbated because traditional DL encoders and decoders are designed and built for input such as images and text instead of source code. Different from images and text, source code is a formal language with its own syntax and semantics. Thus, patches disobeying such syntactic and semantic *domain knowledge* cannot be correct patches to fix a bug.

Ideally, given source code and patches as training data, one expects traditional DL models to learn code syntax and semantics well. However, in practice, such models generate a large portion of uncompilable and incorrect patches [6]–[8], [10], wasting a daunting amount of computing power and, in many cases preventing correct patches from being generated with bounded resources. Thus, it is crucial to enforce syntactic and semantic rules directly on DL models. In addition, such rules must be enforced during the training phase, as opposed to during inference only (e.g., filtering out generated patches that cannot be parsed or type-checked), because enforcing syntax and semantics during inference still causes a large portion (as high as 91%) of uncompilable and incorrect patches [8], [10].

To leverage domain knowledge to guide DL-based APR in a *direct and comprehensive* way, we propose a novel DL-based APR approach—**KNOD**, consisting of a novel three-stage decoder with domain-rule distillation. KNOD has two major novelties. First, different from previous work that generates sequences or production rules [6]–[8], [10], our **three-stage decoder** *directly* generates ASTs of patched code from root to children according to the AST tree structure with *three decoders*: a parent decoder, an edge decoder, and a node decoder. Such a three-stage design enforces the model to naturally capture the tree structure in the ASTs, helping the model learn AST syntax and semantics.

[1]This work is done when Thibaud was at University of Waterloo.
[2]This work is done when Yiling was at Purdue University.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00111

Second, the decoder incorporates a **domain-rule distillation** component to explicitly inject the domain knowledge into the decoding procedure. Specifically, the domain-rule distillation component first represents syntax and semantics as rules expressed in first-order logic (FOL). It then uses these logic rules to refine the teacher-student probability distributions to guide our model to learn to follow these syntactic and semantic rules. Different from existing work, our domain-rule distillation uses logic rules to *explicitly* modify the optimization function in the decoding procedure in *both the training and inference phases*, which thus should have a stronger capability of utilizing domain knowledge.

In summary, this paper makes the following contributions.

- **A three-stage tree decoder** to directly generate Abstract Syntax Trees (ASTs) in three stages to capture tree structures, syntax, and semantics,
- **A domain-rule distillation during training and inference** to explicitly modify the optimization function to guide the three-stage tree decoder to follow code syntax and semantics,
- **An APR technique KNOD** based on the proposed domain-knowledge-distilled tree-decoder architecture,
- **An evaluation** of KNOD on three widely-used benchmarks, Defects4J v1.2, Defects4J v2.0 [13], and QuixBugs [14]. KNOD outperforms all existing non-DL and DL-based approaches by fixing 72 bugs on Defects4J v1.2, fixing 8 and 19 more bugs than the most effective DL-based and non-DL-based APR techniques, respectively. KNOD also fixes the most bugs, 25 and 50, on the QuixBugs and Defects4J v2.0 benchmarks, which shows KNOD's generalizability.

II. Approach

This section presents our proposed approach, KNOD. Section II-A gives an overview of KNOD. Section II-B shows how KNOD represents code; Sections II-C to II-E describes the DL models; Section II-F presents training and inference; Section II-G describes patches' generation and validation.

A. Overview

KNOD consists of two phases: the training phase and the inference phase. During the training phase, KNOD takes buggy code and its patches as input and trains a model (an encoder and a decoder) to learn how to generate patches to fix bugs automatically. During the inference phase, KNOD takes an unseen buggy project, including the location of buggy lines as input, which are standard input that existing APR techniques take [5]–[8], [10]. KNOD's trained decoder generates ASTs, which are then reconstructed into patched code. KNOD then automatically validates these code patches with test cases to generate candidate patches for developers to review.

Figure 1 illustrates how KNOD fixes the Closure-123 bug in the widely-used bug benchmark Defects4J. The Closure-123 bug is uniquely fixed by KNOD. Given the buggy function and the bug location (step ①, where the bug is in a yellow background), KNOD normalizes the buggy function by replacing uncommon identifiers with normalized textual representations (step ②, Section II-B) and then builds the Abstract Syntax Graph (ASG) of the normalized buggy function (step ③, Section II-B). The ASG is the input to KNOD's APR model, which is trained to decode the AST of a normalized patch (steps ④ and ⑤, Sections II-C ∼ II-F). KNOD transforms the AST into a patch in its normal form (step ⑥, Section II-G), which is eventually instantiated into a real code patch by replacing the abstract tokens with the concrete identifiers (step ⑦, Section II-G).

Figure 2 presents the overview of KNOD's **domain-knowledge-distilled tree-decoder architecture** which has two main novelties.

Novelty 1: three-stage tree decoder. First, different from existing APR work that generates sequences or production rules, our *three-stage tree decoder* generates bug fixes in an AST format directly (Section II-D), which is then automatically converted to source code (Section II-G). Specifically, our three-stage tree decoder includes three decoders: (i) a parent decoder, which selects the parent node among generated nodes to work on at each step, (ii) an edge decoder, which generates an edge for the parent node that was selected by the parent decoder, and generates the label of the edge (differ from standard ASTs, our ASTs have labels for edges to be leveraged by models as detailed in Section II-B "ASTs and Abstract Syntax Graphs"), and (iii) a node decoder, which generates a new node connected to the edge generated by the edge decoder. *Different from existing APR decoders that generate patches as token sequences [6]–[8], our decoder generates patches in ASTs with explicit structure. As such, our model is forced to learn explicit code structure. Compared to those generating patches as grammar production rules, our decoder emits AST edges with explicit labels, distinguishing various edge labels. In addition, tree generation is incremental, with one edge and one node emitted at a time. Such a design enables fine-grained control over the quality of generated trees (e.g., enforcing syntactic and semantic validity during generation).*

Using Figure 1④ as an example, KNOD's decoder starts from the root node *BlockStmt*, which is always the same as the root node of the buggy AST (③). In step 1, our decoder selects *BlockStmt* as the parent node for this step, generates an edge with label *statements* for this parent, and then generates the next node *LocalVarDecl*. In step 2, our decoder selects node *LocalVarDecl* as the parent, generates an edge of label *type*, and the next node *RefType*. The resulting AST is in Figure 1⑤. Section IV-B shows that with a three-stage tree decoder, KNOD fixes more bugs.

Novelty 2: domain-rule distillation during training and inference. Second, a straightforward design is to first use KNOD's decoder to generate many candidate patch ASTs and then use the parser, type checker, and test suite to rule out the invalid ones, similar to existing APR techniques [6]–[8], [10]. However, without training the decoder to generate syntactically and semantically valid patches, most of the generated ASTs are invalid, which incurs substantial validation overhead and

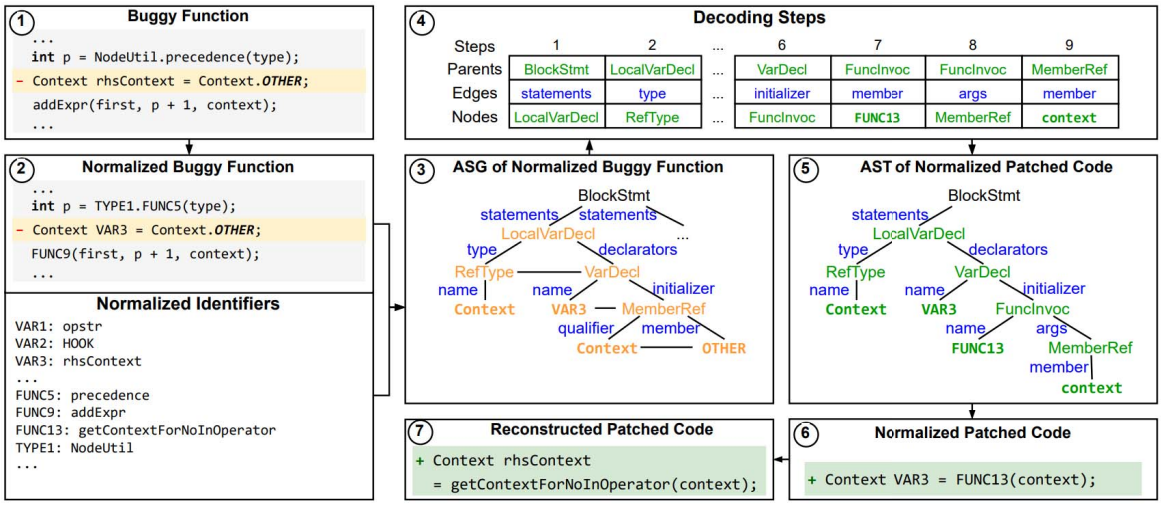


Fig. 1: An example of KNOD fixing bug Closure-123 in Defects4J. Stmt stands for Statement, Var is Variable, Decl is Declaration, Ref is Reference, Func is Function, and Invoc is Invocation.

prevents correct fixes from being generated with bounded efforts.

Therefore, a critical design choice of KNOD is *domain-rule distillation*, which enforces KNOD's decoder to generate valid patches during training in addition to the inference phase. Specifically, during training, we universally encode the grammar and type-checking rules as first-order logic formulas. We create a teacher distribution using these logic rules to modify the student distribution, which is the distribution from our encoder-decoder model. *We create a loss between the student distribution (model distribution) and the teacher distribution (model distribution with syntactic and semantic rules), and add this loss to penalize ASTs that violate the grammatical or type rules.* This loss is added to the traditional loss used by APR techniques, which is between the student distribution (model distribution) and the ground-truth labels (correct developer patches in training data). *The teacher-student loss enables domain knowledge transfer to the models' weights at each iteration* [15]. In summary, our domain-rule distillation enables our APR models to learn from both the training data and the logic rules. During inference, the model automatically distills the invalid generations and emits the likely valid ones.

For the Closure-123 bug in Figure 1, an example semantic rule is that when the model generates a member (e.g., `context`) as an argument of a function invocation (e.g., `FUNC13(context)`), only nodes whose types are compatible with that function's argument type should be generated. Figure 1② shows that `FUNC13` was normalized from `getContextForNoInOperator`, whose argument type is `Context`. Thus, only identifiers whose types are compatible with type `Context` are valid. Our domain-rule distillation sets the probabilities of all identifiers with incompatible types to 0. As a result, the decoder can generate the correct node `context` instead of `p` (of type `int`), which has the highest probability without domain-rule distillation.

An example syntactic rule is that node *MemberRef* must have only one edge of label *member* (for member reference, there must be one and only one member). Since our domain-rule distillation modifies the distributions during training (similar to the example above to set the probabilities of invalid edge/node labels to 0), which teaches the model (by modifying model weights) to give the edge of label *member* the highest probability for the parent node *MemberRef* during inference, KNOD generates a valid AST. Section IV-B shows that with domain-rule distillation, KNOD fixes more bugs.

B. Data Preprocessing and Extraction

KNOD normalizes a given buggy function and its patch and parses them into Abstract Syntax Graphs, which are directional graphs with edges added to connect sibling nodes in an AST. Such additional edges enable closer distance between sibling nodes to help models learn syntaxes and semantics effectively. While the edges between siblings nodes are useful for the encoder to facilitate learning, they are not part of source code and do not need to be generated by our decoder, because it is trivial to add such edges to a generated AST. Thus, we only need an AST decoder.

Code Normalization. Due to the potentially unlimited number of unique identifiers in source code, NMT-based APR models [6]–[8] usually suffer from the large vocabulary size issue and the out-of-vocabulary problem. Therefore, to address these issues, KNOD first performs code normalization by applying src2abs [16], [17] to transform identifiers (e.g., data types, function names, variable names, and literal values) to normalized textual representation. For example, the buggy function for bug Closure-123 (Figure 1①) is transformed to normalized code (Figure 1②), where the mappings from normalized identifiers to concrete identifiers are kept for reconstruction later. In addition, we denote identifiers that appear only in the patch but not in the buggy function as unknown identifiers, which are normalized to special placeholders (e.g., "`TYPE-UNK`", "`FUNC-UNK`", and "`VAR-UNK`").

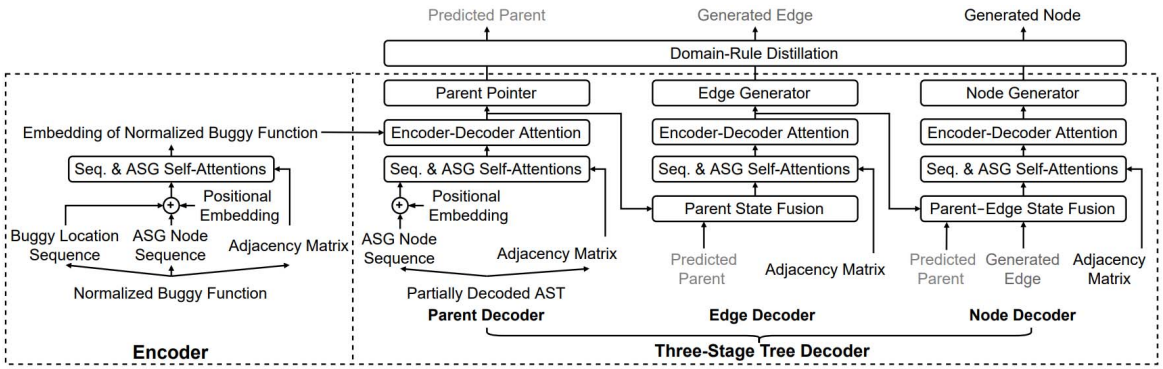


Fig. 2: Architecture of KNOD's model, including a graph-transformer encoder, a three-stage tree decoder (parent, edge, and node decoder), and a domain-rule distillation module.

ASTs and Abstract Syntax Graphs. After parsing the normalized code of the buggy function and the patched line into ASTs, KNOD then represents these ASTs as *Abstract Syntax Graphs*. ASG is a directional graph whose vertices are the AST nodes, while edges are (i) the original edges between AST nodes, and (ii) the new edges between each AST node and its siblings, which make them graphs instead of trees. For example, Figure 1③ illustrates the ASG of the normalized buggy function of the Closure-123 bug. In classic ASTs, edges are introduced between a nonterminal symbol on the left-hand side of a production rule (i.e., the parent) to the symbols, terminal or nonterminal, on the right-hand side of the rule (i.e., the children). While these edges are often not labeled, they have different semantics, which can be leveraged in model training. Thus, we label these edges explicitly. For example, in Figure 1③, node *LocalVarDecl* has two child nodes—node *RefType* denoting a reference type (i.e., `Context`), and node *VarDecl* denoting a variable declaration. The two edges $LocalVarDecl \rightarrow RefType$ and $LocalVarDecl \rightarrow VarDecl$ have different meanings. Therefore, we introduce explicit edge labels, such as *type* and *declarators* for the aforementioned two edges. These labels provide different granularities of information compared to node labels. For instance, a *type* edge may lead to various child nodes such as *RefType* and *BasicType*. We use the additional information carried by edge labels to improve the model. For example, given the parent *LocalVarDecl* node, without the label *type*, a decoder could generate invalid nodes such as `FUNC5`. In contrast, with the label *type*, a decoder could learn to focus on generating type nodes such as *RefType*. We use the `javalang` parser to directly extract and create these edge and node labels from code. Our design is unique compared to the state-of-the-art [9], [10], which do not use explicit edge labels.

C. Encoder

During both training and inference, our encoder takes ASGs of normalized buggy functions as input and output embeddings of these functions to be used by our decoder. We chose graph-transformer [18], [19] as the architecture for our encoder, given its superior scalability and learning capability on graph-structured data. To improve the encoder's effectiveness, we apply *sequence self-attention* and *ASG self-attention* to the encoder to highlight dependencies between every node pair and every adjacent ASG node pair.

Following existing work [10], [18]–[20], we use two vectors/matrices ("ASG Node Sequence" and "Adjacency Matrix" in Figure 2) to represent each ASG, which are the input formats that a deep learning encoder can take. The ASG Node Sequence, notated by $\{n_i\} = (n_1, n_2, \ldots, n_L)$, is a sequence of node names following a pre-order traverse of ASG (e.g., [*BlockStmt*, *LocalVarDecl*, ...]). The adjacency matrix $A = \{a_{ij}\}$ keeps the edge information where $a_{ij} = 0$ means there is no edge between node n_i and n_j, while $a_{ij} = k$ means nodes n_i and n_j are linked by an edge whose label is k (e.g., *statements*). The remaining input to the encoder is "Buggy Location Sequence" (Figure 2). To let the encoder know which nodes in the ASG node sequence belong to the buggy line, KNOD locates the ASG nodes belonging to the buggy line, and generates them as a "Buggy Location Sequence" $\{t_i\} = (t_1, t_2, \ldots, t_L)$ that follows the same order as n_i, where $t_i = 1$ means node n_i belongs to the buggy lines and $t_i = 2$ means node n_i belongs to non-buggy lines.

The ASG node sequence, adjacency matrix, and buggy location sequence are converted to vectorized embeddings for later computation. Respectively, $\{\boldsymbol{n_i}\}$ is the sequence of ASG node embedding, $\boldsymbol{A} = \{\boldsymbol{a_{ij}}\}$ is the embedding of edges in the adjacency matrix, and $\{\boldsymbol{t_i}\}$ is the sequence of buggy location embedding (in this paper, we use light letters for names, labels, etc., use **bolded** letters for vectors, embeddings, etc., and use $\{\}$ for sequence). To let the model learn positional and order information, a positional embedding [21], [22] $\{\boldsymbol{p_i}\}$ is used to encode the absolute index of each node (1, 2, etc.) in the ASG node sequence. All the embeddings are learned by the encoder, and each node n_i is represented by a vector $\boldsymbol{e_i} = \boldsymbol{n_i} + \boldsymbol{t_i} + \boldsymbol{p_i}$.

The encoded vectors $\{\boldsymbol{e_i}\}$ are then fed to a stack of encoder blocks (the number of encoder blocks is configurable), and each encoder block contains a *sequence self-attention module* and an *ASG self-attention module*.

Attention in Encoder. The self-attention module follows the standard architecture in existing architectures [18], [19], [21].

The sequence self-attention captures the dependencies between embeddings of every node pair in the ASG node sequence, which is widely used in sequential transformer-based neural networks [21]. The ASG self-attention explicitly highlights the dependencies between adjacent nodes. Different from traditional tree self-attention, which considers node features only, our ASG self-attention considers both node and edge features. The attention-based hidden states later are fed to a normalization layer [21], [23] and a feed-forward layer [21] to get the final encoder output $\{\boldsymbol{h_i^e}\}$, which are the hidden states for each node in ASG for the input buggy function.

D. Novelty 1: Three-Stage Tree Decoder

Decoding ASTs (more generally, decoding trees) for modern languages such as Java is challenging and less studied. Since modern programming languages such as Java are not context-free, a straightforward approach to decoding trees defined by a context-free grammar that constructs a tree from its root to leaves based on production rules does not work well for Java [24]. Besides, ASTs are domain-specific trees with labels on edges, e.g., *declarators* and *initializer* in Figure 1. Thus, a decoding method needs to decode the edge labels properly. Thus, we propose a novel three-stage tree decoder that decodes AST nodes and edges iteratively. Specifically, our three-stage tree decoder includes three sub-decoders: (i) a parent decoder, which selects the parent node among generated nodes to work on at each step, (ii) an edge decoder, which generates an edge for the parent node selected by the parent decoder, and generates the label of the edge, and (iii) a node decoder, which generates a new node connected to the edge generated by the edge decoder.

1) Parent Decoder: The parent decoder, as the first part of the three-stage tree decoder, takes three inputs, including (i) the encoder's output $\{\boldsymbol{h_i^e}\}$, (ii) the node vectors $\boldsymbol{e_i^d}$ (i.e., $\boldsymbol{e_i^d} = \boldsymbol{n_i^d} + \boldsymbol{p_i^d}$, the sum of the embedding of the node sequence $\{\boldsymbol{n_i^d}\}$ and the positional embedding $\boldsymbol{p_i^d}$), and (iii) the adjacency matrix $\{\boldsymbol{a_{ij}^d}\}$ of the partially generated AST (i.e., the partial AST that the decoder has generated from the beginning to the current decoding iteration). As shown in Figure 2, the parent decoder contains a sequence self-attention, an ASG self-attention, encoder-decoder attention, and a parent pointer.

Attention in Parent Decoder. The parent decoder leverages the same sequence self-attention and AST self-attention as the ones used in the encoder. However, the attention in the decoder is computed among the nodes in the AST of the patch code. Besides, the encoder-decoder attention computes the attention weights from the nodes in the patch AST to the nodes in the input buggy AST, which could capture the dependencies between the patch code and the buggy code. The outputs are then normalized by a normalization layer and projected by a feed-forward layer. In this way, for each node in the patch AST, we get its hidden states $\{\boldsymbol{h_i^{d(p)}}\}$ output by the parent decoder (superscript (p) means the hidden states' output by the *parent* decoder).

Parent Pointer. The parent pointer component leverages the hidden states' output by the last attention module to locate the parent node, for which a new child node would be generated. In particular, the parent point [25] selects an existing node in the partial patch AST as the parent node. $\mathrm{P}_i^{(p)}(j)$, the probability distribution of the j-th nodes being the parent at the i-th decoding iteration is

$$\mathrm{P}_i^{(p)}(j) = \mathrm{softmax}\Big(\frac{W_q \boldsymbol{h_i^{d(p)}} \cdot W_k \boldsymbol{h_j^{d(p)}}}{\sqrt{d}}\Big) \tag{1}$$

where W_q and W_k are trainable weights, and d is the dimension of hidden states. Intuitively, the parent node with the index $p_i^d = \mathrm{argmax}_j \mathrm{P}_i^{(p)}(j)$ at the decoding iteration i, should be the node with the highest attention weight to the i-th node in the AST of normalized patched code.

2) Edge Decoder: After locating the parent node $\{p_i^d\}$ in the current AST, the edge decoder component predicts the label of the new edge that connects the parent node and its child node to be generated. The edge decoder is supposed to work based on the prediction results of the previous parent decoder, since different parent nodes should have different predictions of edge labels. For example, if the parent node is an *IfStmt*, the edge decoder should predict an edge label of *condition*; if the parent node is a *FuncInvoc*, the edge decoder should predict an edge label of *member*. Therefore, the edge decoder takes two inputs, including (i) the parent decoder's output hidden states $\{\boldsymbol{h_i^{d(p)}}\}$, and (ii) the predicted parent index $\{p_i^d\}$. As shown in Figure 2, the edge decoder contains a *parent states fusion layer*, followed by three attention modules, and an edge generation component. The attention modules (i.e., sequence self-attention, ASG self-attention, and encoder-decoder attention) in the edge decoder are similar to those in the parent decoder. Therefore, we focus on describing the novel parent state fusion component and the edge generator as follows.

Parent State Fusion. KNOD incorporates a parent state fusion layer to combine the hidden states of each node $\{\boldsymbol{h_i^{d(p)}}\}$ and the hidden states of its predicted parent node p_i^d: $\boldsymbol{h_i^{d(e)}} = W_f\big(\boldsymbol{h_i^{d(p)}} + \boldsymbol{h_{p_i^d}^{d(p)}}\big)$, where W_f is a trainable parameter for parent states fusion.

Edge Generator. Given the hidden states $\{\boldsymbol{h_i^{d(e)}}\}$ output by the last attention module, the edge generator projects each hidden state to a probability distribution over all the possible edge labels: $\mathrm{P}_i^{(e)}(j) = \mathrm{softmax}(W_e \boldsymbol{h_i^{d(e)}})$, where W_e are the trainable parameters to map the hidden states to a probability distribution over all the edge labels, and $\mathrm{P}_i^{(e)}(j)$ is the probability of the edge label j. The one with the highest probability, i.e., $e_i^d = \mathrm{argmax}_j \mathrm{P}_i^{(e)}(j)$, is predicted as the label of the edge that connects the located parent node and its child node to be generated.

3) Node Decoder: The last step in each decoding iteration is to decode a new child node for the parent node p_i^d with the edge label e_i^d. Therefore, the node decoder takes three inputs, including (i) the hidden states output by the previous edge decoder, (ii) the previously-predicted parent node, and

Nodes	Edge Labels			
	required	**required+**	**optional**	**optional***
MemberRef	member	-	qualifier	selectors
FuncInvoc	member	-	qualifier	args
LocalVarDecl	type	declarators	-	-
Syntax rules for selecting parents and generating edges				

Rule1: $\forall p \in P, \exists e \in E_{req}(p) \cup E_{req+}(p)(F(p,e) = 0)$
$\rightarrow p \in P_{must} \wedge e \in E_{must}(p)$

Rule2: $\forall p \in P, \exists e \in E_{opt}(p) \cup E_{opt*}(p)(F(p,e) = 0)$
$\rightarrow p \in P_{might} \wedge e \in E_{might}(p)$

Rule3: $\forall p \in P, \forall e \in E_{req}(p)(F(p,e) = 1) \wedge E_{req+}(p) = \varnothing \wedge$
$\forall e \in E_{opt}(p)(F(p,e) = 1) \wedge E_{opt*}(p) = \varnothing$
$\rightarrow p \in P_{invalid}$

Rule4: $\forall p \in P, \forall e \in E(e \notin E_{req} \cup E_{req+} \cup E_{opt} \cup E_{opt*}(p))$
$\rightarrow e \in E_{invalid}(p)$

TABLE I: Syntax rules defined in `javalang`, based on which KNOD designs FOL rules for decoding parents and edges. "required" means the node must have one and only one edge with the given label, "required+" means the node must have one or more edges with the given label, "optional" means the node could have zero or one edge with the given label, and "optional*" means the node could have zero, one or multiple edges with the given label.

(iii) the previously-predicted edge label. Figure 2 shows that the node decoder contains a parent-edge state fusion layer, followed by three attention modules (which are similar as those in parent/edge decoders), and a node generator.

Parent-Edge State Fusion. KNOD incorporates a parent-edge state fusion to combine the hidden states of each node $\{\boldsymbol{h_i^{d(e)}}\}$ with the hidden states of its parent node p_i^d and the embedding of the edge e_i^d generated by the edge decoder: $\boldsymbol{h_i^{d(n)}} = W_f\big(\boldsymbol{h_i^{d(e)}} + \boldsymbol{h_{p_i^d}^{d(e)}} + \boldsymbol{e_i^d}\big)$, where $\{\boldsymbol{e_i^d}\}$ is the embedding of generated edges $\{e_i^d\}$.

Node Generator. Similar to the edge generator, the node generator takes the hidden states output by the attention module to compute a probability distribution among all possible nodes: $\mathrm{P}_i^{(n)}(j) = \mathrm{softmax}(W_n \boldsymbol{h_i^{d(n)}})$, where W_n is trainable parameters, and $\mathrm{P}_i^{(n)}(j)$ is the probability of node j being the generated node. The node with the highest probability is generated, i.e., $n_i^d = \mathrm{argmax}_j \mathrm{P}_i^{(n)}(j)$.

E. Novelty 2: Domain-Rule Distillation

We propose domain-rule distillation to force the decoder to generate syntactically and semantically valid patches, which (i) represents *syntax and semantics* as logic rules, (ii) uses logic rules to create *teacher-student distributions*, and (iii) formulates *a new loss function* to guide the models to learn from teacher-student distributions (i.e., to update models' weights) to generate ASTs to follow these logic rules during *both the training and inference phases*.

(i1) Syntactic Rules. Table I shows examples of domain knowledge for a valid AST in `javalang`, where each AST node must have edges with only a small subset of labels. For example, node *MemberRef* must have one and only one edge of label *member*, meaning that to reference a member, there must be one and only one member (e.g., member *context* in Figure 3 ⑤). Such domain knowledge can guide the three-stage tree decoder to predict correct parents and generate correct edge labels. For example, in a partially generated AST, if node *MemberRef* has no edge of label *member*, the decoder should predict *MemberRef* as parent and generate a *member* edge for it; otherwise, the AST is syntactically wrong.

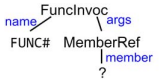
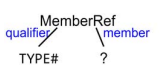
Preconditions	Semantic rules for generating nodes
	**Rule 5:** $\forall n \in N, type(n) \npreceq type(\texttt{FUNC.args}) \rightarrow n \in N_{invalid}$ **Rule 6:** $\forall n \in N, type(n) \preceq type(\texttt{FUNC.args}) \rightarrow n \in N_{might}$
	**Rule 7:** $\forall n \in N, n \notin \texttt{TYPE.fields} \rightarrow n \in N_{invalid}$ **Rule 8:** $\forall n \in N, n \in \texttt{TYPE\#.fields} \rightarrow n \in N_{might}$
	**Rule 9:** $\forall n \in N, type(n) \npreceq type(\texttt{VAR\#}) \rightarrow n \in N_{invalid}$ **Rule 10:** $\forall n \in N, type(n) \preceq type(\texttt{VAR\#}) \rightarrow n \in N_{might}$

TABLE II: Examples of KNOD's semantic rules designed in domain-rule distillation module, and preconditions of applying them. Semantic rules are used during generating nodes (? refers to the node to be generated).

To leverage such syntax, we create logic rules, shown in Table I, where p refers to parent node, P is the collection of nodes that might be predicted as a parent by the model, e is edge label, and $E_{req}(p)$, $E_{req+}(p)$, $E_{opt}(p)$, and $E_{opt*}(p)$ are the sets of edge labels that are required or optional for node p (e.g., $E_{req}(\textit{MemberRef}) = \{\textit{member}\}$, which is the edge label in the same row as *MemberRef* and under column *required*). $F(p,e)$ is a function that returns the number of edges that link to parent p with label e. P_{must}, P_{might}, and $P_{invalid}$ are the sets of nodes that must be predicted as a parent, might be predicted as a parent, or impossible as a parent in the future decoding iterations. Respectively, $E_{must}(p)$, $E_{might}(p)$, and $E_{invalid}(p)$ are the sets of edges that must, might, or must not be generated for parent node p.

The intuition of syntactic rules is that any edge labels required by a node must be generated, any edge labels optional for a node could be generated, and if a node has all potential edges generated, it cannot generate more edges. For example, **Rule 1** states that for a node with any required edge not generated yet, it must be predicted as parent and the required edge must be generated.

During every decoding iteration, domain-rule distillation applies the syntax rules to find parent nodes and corresponding edges belonging to each category (i.e., P_{must}, P_{might}, etc.), which is used later (in the (ii) Teacher-Student Distributions step to modify the parent decoder and edge decoder's output probability distribution ($\mathrm{P}_i^{(p)}$ and $\mathrm{P}_i^{(e)}$).

(i2) Semantic Rules. Another important type of domain knowledge—code semantics, i.e., type matching, of which KNOD further takes advantages, is used to train the model to avoid generating semantically-invalid ASTs. Specifically, KNOD uses `JavaParser` to statically analyze the entire

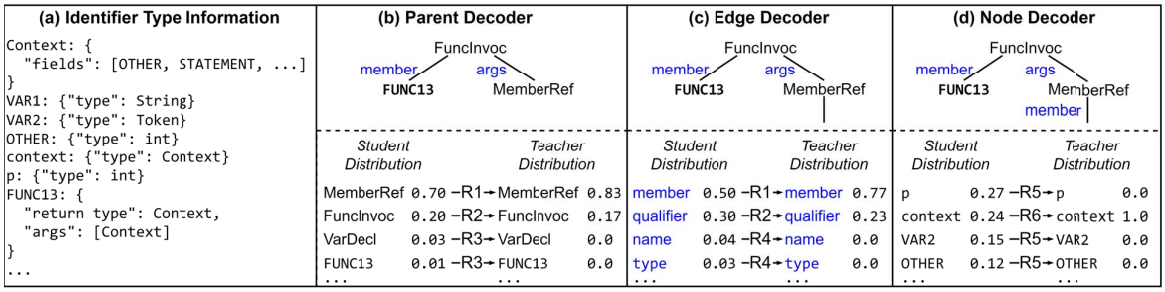


Fig. 3: Process of domain-rule distillation modifying the probability distribution and KNOD generating node `context` for Defects4J's Closure-123. R1 denotes Rule 1.

buggy program (not just the buggy function) to collect type information of each accessible identifier. For example, the left of Figure 3 shows the following type information for the Closure-123 bug: (i) the data type for each variable, (ii) the return type and the arguments for each function, and (iii) the fields and the declared functions for each class. With such type information, we design semantic rules for the decoder to generate nodes (Table II). A rule is applied when its precondition matches the partially generated AST. n refers to leaf node labels, N is the collection of all the accessible identifiers' names, $type(n)$ returns the data type of n, and $\preceq$ means type compatibility. N_{might} and $N_{invalid}$ are the collections of nodes that can or cannot be generated. Semantic rules ensure type compatibility during node generation, for example, **Rule 5** states when the model generates a member as an argument of a function invocation, only the nodes whose types are compatible (i.e., same or subtype) with the argument type defined in the function's signature are possible to be generated.

During every decoding iteration, semantic rules are used to find nodes that are possible or invalid to be generated (i.e., nodes belonging to N_{might} and $N_{invalid}$), which is used later (in the (ii) Teacher-Student Distributions step) to modify the node decoder's output probability distribution ($\mathrm{P}_i^{(n)}$).

(ii) Teacher-Student Distributions. To let the APR model learn syntactic and semantic rules, we adapt the teacher-student architecture [15]. Specifically, the rules are the teacher's knowledge, which we want the student (the APR model) to learn. The terms teacher and student follow prior work [15], as the teacher "teaches" the model to generate output satisfying the domain knowledge rules that the teacher knows.

The *student distribution* is the probability distribution output by the three-stage tree decoder, i.e., $\mathrm{P}^{(p)}$, $\mathrm{P}^{(e)}$ and $\mathrm{P}^{(n)}$. We create the *teacher distribution* $\hat{\mathrm{P}}^{(p)}$ by modifying the student distribution as

$$\hat{\mathrm{P}}_i^{(p)} = \begin{cases} 1 & p_i \in P_{must} \\ \mathrm{P}_i^{(p)} & p_i \in P_{might} \\ 0 & p_i \in P_{invalid} \end{cases} \tag{2}$$

and re-normalize it, while distributions $\hat{\mathrm{P}}^{(e)}$ and $\hat{\mathrm{P}}^{(n)}$ are created similarly. The teacher distributions satisfy the rules, as the probability of parents, edges, and nodes that must be generated are the highest, and the probability of invalid ones are 0.

For example, in Figure 3 (b), by applying **Rule 1**, *MemberRef* has a required edge *member* not generated and thus belongs to P_{must}. The probability of *MemberRef* is modified to 1, and then re-normalized to 0.83. In Figure 3(c), after selecting *MemberRef* as the parent, by applying **Rules 1, 2 and 4**, *member* is required, *qualifier* is optional, and the rest edges are invalid for *MemberRef*. Thus, the probability of *member* is set to 1 (re-normalized to 0.77) and the probability of *qualifier* is kept as 0.3 (re-normalized to 0.23). In Figure 3(d), domain-rule distillation applies **Rules 5 and 6** as the partially generated AST matches their preconditions. Thus, we keep only identifiers (e.g., `context`) that are types compatible with the argument type of `FUNC13` (i.e., type `Context` is valid). Our domain-rule distillation sets the probabilities of all other identifiers with incompatible types to 0. As a result, the decoder generates the correct node `context` instead of `p` of type `int` that has the highest probability in the student distribution.

(iii) Distillation. Distillation transfers the domain knowledge formulated by syntax and semantic rules into the APR models' weights [15], which speeds up the training and helps APR models learn the domain knowledge and compute better probability distributions. Distillation is performed by introducing an extra loss to minimize the difference between student and teacher distribution. The overall training objective is to minimize (i) the loss between the student distributions and the ground-truth labels, and (ii) the loss between the student distributions and the teacher distributions, where (iii) is the standard loss used by APR techniques [6]–[8], [10], while (2) is a novel contribution of this paper. Specifically, the joint loss is as follows:

$$\begin{aligned} Loss = & L_{CE}(\mathrm{P}^{(p)}, y^{(p)}) + L_{CE}(\mathrm{P}^{(e)}, y^{(e)}) + L_{CE}(\mathrm{P}^{(n)}, y^{(n)}) \\ & L_{KL}(\mathrm{P}^{(p)}, \hat{\mathrm{P}}^{(p)}) + L_{KL}(\mathrm{P}^{(e)}, \hat{\mathrm{P}}^{(e)}) + L_{KL}(\mathrm{P}^{(n)}, \hat{\mathrm{P}}^{(n)}) \end{aligned} \tag{3}$$

where the $L_{CE}()s$ are cross-entropy [26] between the student distribution and the ground-truth labels ($y^{(p)}$, $y^{(e)}$ and $y^{(n)}$ are the ground-truth for parents, edges, and nodes respectively), and $L_{KL}()$s are the loss that we add, which calculates the Kullback–Leibler divergence [27] between the student dis-

tributions and the teacher distributions. By minimizing the training objective during training, the APR models learn to generate syntactically and semantically correct ASTs.

F. Training and Inference

In the training phase, the model takes the ASG of the normalized buggy function and AST of normalized patched code to learn the transformation from the former to the latter. We also leverage ensemble learning [28], [29] to train multiple models to increase the diversity of the learned fix patterns. Following previous work [7], [8], we first train different models with random hyper-parameters (e.g., number of encoder blocks, decoder blocks, or hidden states dimension), and then select the Top-k models according to their loss on the validation set.

In the inference phase, for the given buggy function, each of the k trained models generates a list of ASTs of normalized patched code. Among all the generated ASTs, KNOD first ranks them via their ranks and average probabilities.

G. Patch Generation and Validation

The generated ASTs are converted to normalized source code, which still contains normalized tokens. KNOD reconstructs them into concrete patches by replacing the normalized tokens with the corresponding concrete identifiers, based on the mapping recorded in the code normalization phase (Section II-B). For example, in Figures 1 ⑥ and ⑦, the normalized tokens `VAR3` and `FUNC13` are replaced by the concrete identifiers `rhsContext` and `getContextForNoInOperator`, respectively. Such concrete patched code are the final patches generated by KNOD. For normalized patched code with *unknown* tokens (e.g., `TYPE-UNK`), KNOD performs type-analysis to find compatible concrete values for reconstruction. In particular, for each unknown token, KNOD first analyzes its parent and sibling (if any) nodes in the ASTs, summarize rules that the unknown token should follow (e.g., its data type), and then replaces the unknown tokens with all valid concrete identifiers to ensure semantic matching.

All generated patches are validated against test cases. Following prior work [7], [8], [30], validation terminates when it finds a *plausible* patch that either (i) passes all test cases or (ii) passes all originally-passed tests and at least one originally-failed test case.

III. Experiment Setup

We evaluate KNOD with three research questions: **RQ1: Effectiveness and Generalizability.** How does KNOD perform compared to existing APR techniques? **RQ2: Ablation Study.** What is the contribution of each component in KNOD? and **RQ3: Ranking.** How does KNOD rank the correct patches compared to other tools?

A. Datasets

Training Data. We construct the training data for KNOD from the dataset shared in previous work [7], [10], which are mined from open-source GitHub Java projects. Following previous work [7], [8], [10], we remove projects that are in or cloned from Defects4J projects from our training set. In total, our training data contains 576,002 pairs of buggy programs and their developer patches, which is randomly split into training set (90%) and validation set (10%). The validation set is used to tune and select models.

Bug Benchmarks. We evaluate KNOD on three well-established bug benchmarks, including: (1) Defects4J v1.2 [13], the most widely-used version of the Defects4J benchmark with 393 Java bugs [1], (2) Defects4J v2.0 [13]: the latest version of the Defects4J benchmark with additional 444 Java bugs, and (3) QuixBugs [14], the widely-used benchmark with 40 Java bugs.

B. Evaluated Techniques

To compare the effectiveness of KNOD with existing APR techniques, we include the following state-of-the-art APR techniques for comparison.

- **Non-DL-based APR:** we compare KNOD with SimFix [31] and TBar [32], since they are the most effective heuristic-based and template-based non-DL-based APR techniques [33].
- **DL-based APR:** we compare KNOD with state-of-the-art DL-based APR techniques for Java programs, including SequenceR [6], DLFix [34], CoCoNuT [7], CURE [8], RewardRepair [5], and Recoder [10].

To study the contribution of each novelty of KNOD, we implement and evaluate the following variants of KNOD.

- **KNOD$_{\text{-decoder}}$**: replacing the entire three-stage tree decoder with a traditional sequential decoder.
- **KNOD$_{\text{-distTrain}}$**: removing domain-rule distillation from the decoder during training (keeping it during inference).
- **KNOD$_{\text{-distInf}}$**: removing domain-rule distillation from the decoder during inference (keeping it during training).

C. Experimental Procedure

Fault Localization. We perform our experiment under two different settings of fault localization: (1) perfect localization, where the actually fault localization is given to the tools, and (2) spectrum based fault localization, where KNOD uses the suspicious faulty locations reported by Ochiai [35] (a spectrum based fault localization tool). Both settings are widely used in previous works [5], [7], [8], [10].

Patch Correctness. In line with previous work [5], [7], [8], [10], for evaluation purpose only, we manually check the correctness of plausible patches returned by KNOD. We consider a plausible patch *correct* if it is semantically equivalent to developer patches. The labeling procedure involves two participants. The agreement ratio is 92.1% and inconsistent cases are resolved by further discussion.

Implementation. For ASG construction, we use the widely-used toolkits `javalang` [36] and `JavaParser` [37] to first parse buggy functions and patches into ASTs. The APR models are implemented with PyTorch [38]. To select the

[1]Following previous work [7], [8], two duplicated bugs (Closure-63 and 93) are removed from our evaluation.

hyperparameters, we use random search within the following range: number of encoder layers (6-8), number of parent and edge decoder layers (1-2), number of node decoder layers (4-8), embedding and hidden states dimension (256-384). We use a dropout rate set to 0.1 to avoid overfitting, and use Adam optimizer with learning rate being $2.5e^{-4}$. We tune the top-5 models with the lowest perplexity on the validation set until convergence for ensemble learning. In the inference stage, we use beam search [39] with beam size set to 1,000 to generate patches for each bug in the bug benchmarks. During validation, we set a five-hour running-time limit, which is the same as existing work [7], [10], [31], [34], [40].

Infrastructure. We train KNOD on one 56-core server with eight NVIDIA GeForce RTX 2080 TI GPUs, and evaluate KNOD on the same server with one NVIDIA GeForce RTX 2080 TI GPU.

D. Threats to Validity

Threats to internal validity lie in the approach implementation and manual patch correctness identification. To mitigate these threats, multiple authors check the code and participate in the manual labeling procedure. *Threats to external validity* lie in bug benchmarks used in our evaluation, which cannot guarantee the generalizability on other benchmarks. To mitigate these threats, we perform our experiments on three widely-used benchmarks with up to 877 real-world Java bugs. Evaluation on more benchmarks of different program languages could be done in the future since our approach is not specifically designed for Java.

IV. Result

A. RQ1: Effectiveness and Generalizability

Techniques	Defects4J v1.2	Defects4J v2.0	QuixBugs
SequenceR [6]	14	-	-
SimFix [31]	28	-	-
DLFix [34]	38	-	-
CoCoNuT [7]	44	-	13
RewardRepair [5]	45	45	20
TBar [32]	53	-	-
CURE [8]	56	19	**25**
Recoder [10]	64	-	17
KNOD (our approach)	**72**	**50**	**25**

TABLE III: Number of correctly fixed bugs by each tool on three benchmarks with perfect fault localization. "-" means that tool has not released its performance on the benchmark.

Techniques	Defects4J v1.2	Defects4J v2.0	QuixBugs
TBar [32]	23	8	-
SimFix [31]	24	2	-
RewardRepair [5]	29	22	19
DLFix [34]	30	-	-
Recoder [10]	**45**	19	17
KNOD (our approach)	38	**24**	**23**

TABLE IV: Number of correctly fixed bugs by each tool on three benchmarks with spectrum based fault localization [35].

Results with perfect fault localization. Table III shows the number of bugs that are correctly fixed by KNOD and other

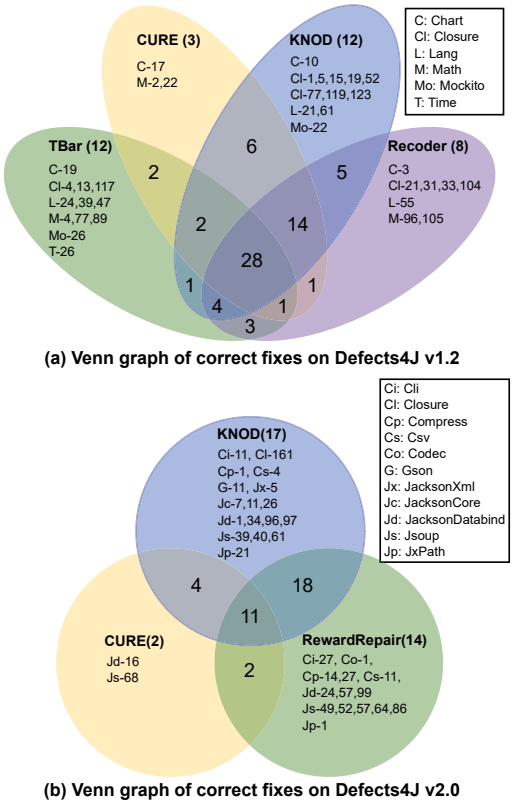


Fig. 4: Uniquely fixed bugs of KNOD and the existing state-of-the-art tools with perfect fault localization. Numbers in () are the number of uniquely fixed bugs.

APR techniques on three benchmarks with perfect fault localization. KNOD fixes 72 bugs, outperforming all the compared techniques on the most widely-used benchmark Defects4J v1.2, 8 and 19 more bugs than the best DL-based and non-DL-based APR approach Recoder and TBar respectively. In addition to the widely-used benchmark Defects4J v1.2, Table III also presents the effectiveness of KNOD on additional two benchmarks, Defects4J v2.0 and QuixBugs. KNOD is consistently effective on both additional benchmarks, i.e., fixing 50 bugs on Defects4J v2.0 and 25 bugs on QuixBugs, indicating the generalizability of KNOD on different bugs.

In addition, we calculate the patch precision of KNOD, i.e., the ratio of correct patches to plausible patches. We find that KNOD achieves 86.7% precision (i.e., 72 out of 83 plausible patches generated for Defects4J v1.2 are correct), which is substantially higher than existing APR techniques under the same configuration [8] (e.g., the top-3 precision of existing APR DLFix/TBar/RewardRepair is 58.4%/62.4%/70.3%).

Results with spectrum-based fault localization. Table IV shows the number of bugs correctly fixed by KNOD and other tools with spectrum-based fault localization. KNOD still fixes the most number of bugs on Defects4J v2.0 and QuixBugs, 24 and 23 respectively. KNOD fixes the second most on Defects4J

v1.2 (38) and still outperforms the most recent APR paper RewardRepair (29). By analyzing the bugs that Recoder fixes but KNOD does not, we find KNOD correctly fixes most of them with perfect localization, suggesting that KNOD can achieve better results with a better fault localization technique.

```
- return "title=\"" + toolTipText
+ return "title=\"" + ImageMapUtilities.htmlEscape(toolTipText)
        + "\"alt=\"\"";
```

(a) Chart-10 in Defects4J v1.2

```
+ if (gramps.isDelProp()) { return false; }
  String propName = parent.getLastChild().getString();
```

(b) Closure-5 in Defects4J v1.2

```
+ if (getBaseValue() == null) { return 1; }
  return ValueUtils.getLength(getBaseValue());
```

(c) JxPath-21 in Defects4J v2.0

Fig. 5: Examples of bugs only fixed by KNOD.

Uniquely Fixed Bugs. Figure 4(a) presents the number of overlapped and unique bugs that are fixed by KNOD and the other three APR techniques that fix the most number of bugs on Defects4J v1.2 (i.e., TBar, CURE, and Recoder). KNOD complements the state-of-the-art APR techniques by fixing 12 unique bugs. On Defects4J v2.0, KNOD complements RewardRepair by fixing 21 unique bugs, and complements CURE by fixing 35 unique bugs (as shown in Figure 4(b)), which shows KNOD can be an excellent complementary technique to existing APR tools.

Figure 5 presents three bugs—Chart-10, Closure-5, and JxPath-21—that are fixed only by KNOD. KNOD is able to fix Chart-10, which is hard to fix as it involves two project-specific identifiers (`ImageMapUtilities` and `htmlEscape`) that are declared outside of the buggy function, thanks to the domain-rule distillation that helps find the correct type-matched function call. KNOD is also good at inserting code snippets to fix bugs (Figure 5 (b) and (c)).

Execution Time. KNOD spends 12.8s on average generating one thousand candidate patches for a given bug (using one NVIDIA RTX 2080 TI GPU).

B. RQ2: Ablation Study

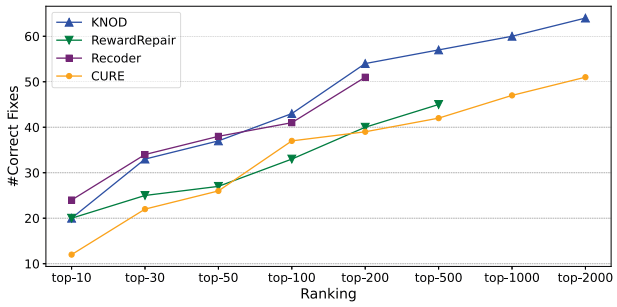
Table V shows the effectiveness of each novel component of KNOD on Defects4J v1.2: adding each component improves the effectiveness of KNOD. Specifically, the novel three-stage tree decoder enables KNOD to fix 16 more bugs (KNOD vs. KNOD$_{\text{-decoder}}$), as sequential decoder cannot leverage structural information well. Adding domain-rule distillation during training fixes 10 more bugs (KNOD vs. KNOD$_{\text{-distTrain}}$), which means only applying syntax and semantic checking in inference do not work well as the model gives poor ranking without learning domain-rule distillation during training. Yet, including domain-rule distillation in the inference phase also help to fix 3 more bugs (KNOD vs. KNOD$_{\text{-distInf}}$), which can be considered as a second guarantee of syntax/semantic checking.

In addition to the number of correct fixes, Table V also includes the compilation rate of patches generated by each model. Without the three-stage tree decoder, KNOD$_{\text{-decoder}}$ generates a lot more uncompilable patches, which shows that the sequential decoder fails to learn code syntax and semantics well. Moreover, applying domain-rule distillation during training and inference stages both help with generating more compilable patches.

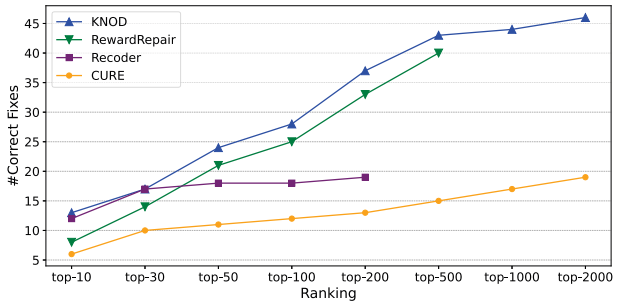
In summary, each component of KNOD positively contributes to its effectiveness. The superiority of KNOD$_{\text{-distInf}}$ to KNOD$_{\text{-distTrain}}$ also confirms that injecting domain-rule distillation into the *training* phase is more effective than only considering domain knowledge in the *inference* phase.

Variants	#Bugs	Compilation Rate
KNOD$_{\text{-decoder}}$	56	33.6%
KNOD$_{\text{-distTrain}}$	62	43.8%
KNOD$_{\text{-distInf}}$	69	46.1%
KNOD	**72**	**47.0%**

TABLE V: Ablation study of each component on Defects4J v1.2.



(a) Ranking of correct fixes on Defects4J v1.2



(b) Ranking of correct fixes on Defects4J v2.0

Fig. 6: Ranking of correct fixes generated by KNOD and existing state-of-the-art DL-based APR tools.

C. RQ3: Ranking

We further study the ranking of the correct fixes generated by KNOD. In Figure 6, we compare KNOD's ranking with those of CURE, Recoder and RewardRepair on Defects4J v1.2 and Defects4J v2.0. Other tools were excluded because they either have not shared the list of the generated candidate patches,

or we cannot successfully reproduce their results. For a fairer comparison, we set the beam size to 200 in this experiment, which is the smallest beam size used by CURE, Recoder and RewardRepair. KNOD uses the ensemble result from five models, while CURE uses a ensemble of ten models and the other two techniques use one. Since the number of models used is a design choice of different techniques, we compare these techniques as is for a fair ranking comparison.

KNOD generates more correct fixes than CURE and RewardRepair in all the ranges from top-10 to top-2000, which shows that KNOD consistently outperforms RewardRepair and CURE. Although Recoder fixes a competitive number of bugs as KNOD on Defects4J v1.2, KNOD generates a lot more correct fixes on Defects4J v2.0 in the top-200 patches (37 versus 19), which shows KNOD's better generalizability to other benchmarks.

Since one can choose the number of candidate patches to validate based on the resource budget, our results show that overall, KNOD fixes the most number of bugs in all budget settings.

V. Limitation

The main limitations of KNOD are that (1) KNOD cannot fix multi-hunk bugs (i.e., bugs that need fix for multiple locations and files) very well (although KNOD can fix some relatively simple multi-hunk bugs), and (2) the performance of KNOD depends on the accuracy of fault localization tools. These limitations are shared by most DL-based APR techniques. An important bottleneck in fixing multi-hunk bugs is fault localization, which is an orthogonal problem, which means fixing multi-hunk bugs and developing better fault localization tools are important related works to explore. Despite the limitations, KNOD still outperforms the existing state-of-the-art APR tools on many benchmarks.

VI. Related Work

A. DL-based APR

Researchers have proposed various APR techniques [2]–[4], which leverage heuristics [41]–[43], templates [32], [44], constraints [45]–[47], or advanced deep learning techniques [5]–[10], [48] to enable patch generation. Based on the output formats of the decoders, existing DL-based APR can be categorized into *code-generation APR* [6]–[8] or *edit-generation APR* [9], [10], [49]. In particular, the decoders of edit-generation APR [9], [10] first generate edits at different levels (e.g., AST-level edits) and then transform the buggy code into patches based on these edits; the code-generation APR tools, SequenceR [6], CoCoNuT [7], CURE [8] and RewardRepair [5], directly generate the code sequences of patches; while DLFix [34], CODIT [50], and our approach KNOD generate the ASTs of patches. Different from existing work that generates AST or AST edits based on production rules, KNOD directly generates ASTs with an explicit tree structure, which forces the model to capture the code structure. To this end, our work proposes a novel three-stage decoder with domain-rule distillation to comprehensively utilize domain knowledge in source code.

RewardRepair [5] trains models based on *dynamic* domain knowledge (i.e., the patch execution and compilation information). KNOD is different because it leverages *static* domain knowledge, since collecting such dynamic domain knowledge and incorporating it into the training phases is often very expensive. Also, KNOD uses direct and finer-grained syntactic and type rules (as opposed to indirect and coarser-grained test case passing/failing information).

B. DL-based Code Generation

Recent code generation techniques leverage advanced DL to directly generate code from natural language specifications. Early DL-based code generation techniques [51] generate code based on tokens. Due to the rich structural information in source code, recent work [52]–[56] leverages encoder-decoder architectures to generate ASTs. Different from DL-based code generation, our technique is designed for program repair and our proposed decoder is novel in architecture and domain-rule distillation. In addition to code generation, some DL-based techniques [57]–[59] generate token-level or AST-level edits for program. Instead of generating edits, our approach directly generates patch code in the AST format via a novel decoder.

A recent direction of DL-based code generation is applying large language models (LLMs) trained on source code to generate code, such as CodeBert [60], CodeT5 [61], CodeGen [62], InCoder [63], and Codex [64]. These LLMs are generic models, while KNOD is a customized model that contains a novel three-stage tree decoder and domain-knowledge distillation to fix bugs.

VII. Conclusion

We propose a DL-based APR approach KNOD, which incorporates domain knowledge to guide patch generation in a *direct and comprehensive* way. KNOD includes (1) a novel three-stage decoder to *directly* generate patch ASTs based on the inherent tree structure of ASTs with *three decoders*, and (2) a novel domain-rule distillation component to explicitly inject domain knowledge into the decoding procedure during *both the training and inference phases*. KNOD is consistently effective on three widely-used benchmarks, fixing 147 bugs in total with perfect fault localization. Our ablation study further confirms the contribution of both novelties in our domain-knowledge-guided tree-decoder architecture.

Data Availability: our replication package is available at [65].

Acknowledgment

We thank the reviewers for their insightful comments and suggestions. This work is partially supported by a J.P. Morgan AI Faculty Research Award. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 492–501. [Online]. Available: https://doi.org/10.1145/1134285.1134355

[2] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[3] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, 2018. [Online]. Available: https://doi.org/10.1145/3105906

[4] ——, "The living review on automated program repair," Dec. 2020, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01956501

[5] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the International Conference on Software Engineering*, 2022. [Online]. Available: http://arxiv.org/pdf/2105.04123

[6] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair," *TSE*, 2019.

[7] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "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*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[8] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[9] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[10] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[11] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, 2022. [Online]. Available: http://arxiv.org/pdf/2104.08308

[12] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of ASE*, 2022. [Online]. Available: http://arxiv.org/pdf/2203.12755

[13] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *ISSTA*, 2014, pp. 437–440.

[14] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge," in *SPLASH*, 2017, p. 55–56.

[15] Z. Hu, X. Ma, Z. Liu, E. Hovy, and E. Xing, "Harnessing deep neural networks with logic rules," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2410–2420. [Online]. Available: https://aclanthology.org/P16-1228

[16] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *CoRR*, vol. abs/1812.08693, 2018.

[17] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019.

[18] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," vol. abs/1911.06455, 2019. [Online]. Available: http://arxiv.org/abs/1911.06455

[19] V. P. Dwivedi and X. Bresson, "A generalization of transformer networks to graphs," *CoRR*, vol. abs/2012.09699, 2020. [Online]. Available: https://arxiv.org/abs/2012.09699

[20] J. Zhang, J. Du, Y. Yang, Y.-Z. Song, S. Wei, and L. Dai, "A tree-structured decoder for image-to-markup generation," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 11 076–11 085. [Online]. Available: https://proceedings.mlr.press/v119/zhang20g.html

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[22] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," *CoRR*, vol. abs/1705.03122, 2017. [Online]. Available: http://arxiv.org/abs/1705.03122

[23] L. J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. abs/1607.06450, 2016. [Online]. Available: http://arxiv.org/abs/1607.06450

[24] X. Wang, H. Pham, P. Yin, and G. Neubig, "A tree-based decoder for neural machine translation," *CoRR*, vol. abs/1808.09374, 2018. [Online]. Available: http://arxiv.org/abs/1808.09374

[25] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," 2015. [Online]. Available: https://arxiv.org/abs/1506.03134

[26] G. Cybenko, D. O'Leary, and J. Rissanen, *The Mathematics of Information Coding, Extraction and Distribution*, ser. The IMA Volumes in Mathematics and its Applications. Springer New York, 1998. [Online]. Available: https://books.google.com/books?id=jDrp4QEGioMC

[27] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79 – 86, 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729694

[28] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and Systems Magazine*, vol. 6, no. 3, pp. 21–45, 2006.

[29] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, pp. 1–39, 2009.

[30] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *FSE*, ser. ESEC/FSE 2017. ACM, 2017, p. 831–841. [Online]. Available: https://doi.org/10.1145/3106237.3106274

[31] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 298–309. [Online]. Available: https://doi.org/10.1145/3213846.3213871

[32] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-Based Automated Program Repair," in *ISSTA*. ACM, 2019.

[33] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," *CoRR*, vol. abs/2008.00914, 2020. [Online]. Available: https://arxiv.org/abs/2008.00914

[34] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-Based Code Transformation Learning for Automated Program Repair," in *ICSE*. ACM, 2020, p. 602–614.

[35] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[36] C. Thunes, "javalang," 2020. [Online]. Available: https://github.com/c2nes/javalang

[37] N. Smith, D. Van Bruggen, and F. Tomassetti, "Javaparser: Visited," 2019. [Online]. Available: https://github.com/javaparser/javaparser

[38] "Pytorch," 2022. [Online]. Available: https://pytorch.org/

[39] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[40] S. Saha, R. k. Saha, and M. r. Prasad, "Harnessing Evolution for Multi-Hunk Program Repair," in *ICSE*. IEEE, 2019, pp. 13–24.

[41] Y. Yuan and W. Banzhaf, "ARJA: automated repair of java programs via multi-objective genetic programming," *IEEE Trans. Software*

Eng., vol. 46, no. 10, pp. 1040–1067, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2874648

[42] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[43] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 648–659. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115675

[44] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 19–30. [Online]. Available: https://doi.org/10.1145/3293882.3330559

[45] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[46] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Software Eng.*, vol. 43, no. 1, pp. 34–55, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

[47] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, ser. Lecture Notes in Computer Science, T. E. Colanzi and P. McMinn, Eds., vol. 11036. Springer, 2018, pp. 65–86. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9_3

[48] D. Drain, C. B. Clement, G. Serrato, and N. Sundaresan, "Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons," *CoRR*, vol. abs/2105.09352, 2021. [Online]. Available: https://arxiv.org/abs/2105.09352

[49] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," in *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 19–20. [Online]. Available: https://doi.org/10.1145/3387940.3392181

[50] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[51] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang, and A. W. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1057

[52] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1004

[53] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, R. Barzilay and M. Kan, Eds. Association for Computational Linguistics, 2017, pp. 440–450. [Online]. Available: https://doi.org/10.18653/v1/P17-1041

[54] ——, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, E. Blanco and W. Lu, Eds. Association for Computational Linguistics, 2018, pp. 7–12. [Online]. Available: https://doi.org/10.18653/v1/d18-2002

[55] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, R. Barzilay and M. Kan, Eds. Association for Computational Linguistics, 2017, pp. 1139–1149. [Online]. Available: https://doi.org/10.18653/v1/P17-1105

[56] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 8984–8991. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/6430

[57] Z. Yao, F. F. Xu, P. Yin, H. Sun, and G. Neubig, "Learning structural edits via incremental tree transformations," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=v9hAX77–cZ

[58] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=BJl6AjC5F7

[59] S. Brody, U. Alon, and E. Yahav, "A structural model for contextual code changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 215:1–215:28, 2020. [Online]. Available: https://doi.org/10.1145/3428283

[60] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *CoRR*, vol. abs/2002.08155, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[61] W. Yue, W. Weishi, J. Shafiq, and C. H. Steven, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

[62] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[63] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022. [Online]. Available: https://arxiv.org/abs/2204.05999

[64] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[65] "Replication package of this work," 2022. [Online]. Available: https://github.com/lin-tan/knod