2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00032

Learning Program Representations with a Tree-Structured Transformer

Wenhan Wang[†], Kechi Zhang[*], Ge Li[*§], Shangqing Liu[†], Anran Li[†], Zhi Jin[*§], Yang Liu[†]
[*]Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education,
Institute of Software, EECS, Peking University, Beijing, China
[†]Nanyang Technological University, Singapore
Email: wenhan.wang@ntu.edu.sg, zhangkechi@pku.edu.cn, lige@pku.edu.cn, shangqin001@e.ntu.edu.sg,
anran.li@ntu.edu.sg, zhijin@pku.edu.cn, yangliu@ntu.edu.sg

***Abstract*—Learning vector representations for programs is a critical step in applying deep learning techniques for program understanding tasks. Various neural network models are proposed to learn from tree-structured program representations, *e.g.*, abstract syntax tree (AST) and concrete syntax tree (CST). However, most neural architectures either fail to capture long-range dependencies which are ubiquitous in programs, or cannot learn effective representations for syntax tree nodes, making them incapable of performing the node-level prediction tasks, *e.g.*, bug localization. In this paper, we propose Tree-Transformer, a novel recursive tree-structured neural network to learn the vector representations for source codes. We propose a multi-head attention mechanism to model the dependency between siblings and parent-children node pairs. Moreover, we propose a bi-directional propagation strategy to allow node information passing in two directions, bottom-up and top-down along trees. In this way, Tree-Transformer can learn the information of the node features as well as the global contextual information. The extensive experimental results show that our Tree-Transformer significantly outperforms the existing tree-based and graph-based program representation learning approaches in both the tree-level and node-level prediction tasks.**

I. Introduction

The rapid development of software engineering applications is incurring enormous growth of code-related data, which makes "Deep Learning (DL) for Software Engineering (SE)" particularly important. DL for SE have been improving software development in many application fields, such as clone detection [1], [2], vulnerability detection [3] and code summarization [4], [5]. One major challenge for these DL for SE applications is how to represent source code to capture their syntactical and semantic information effectively.

Existing DL for SE works have incorporated syntactic or semantic structure information, *e.g.*, abstract syntax tree (AST) and data/control flow into the learning process through tree-structured neural networks [6]–[9] or graph neural networks (GNN) [10]–[12]. The tree-based approaches extract the syntax tree of the source code and use it as the input to the learning model, while graph-based methods use various program analysis techniques to convert the program into a graph and use it as the input to the model. For example, Mou et al. [6] propose the tree-based convolutional neural network (TBCNN) and apply it to the program classification using ASTs. Allamanis et al. [10] convert programs to graphs by adding data flow edges to ASTs, and apply the GNNs to the program graphs to identify misused variables.

Although the existing structured-based techniques have shown their advantages on various software engineering tasks, they still have the following limitations. For better illustration, we summarize the existing structured-based techniques in Table I. **First**, many graph-based approaches fail to capture the long-term dependencies in source code (see column "Global" in Table I). Since the long-term dependencies are ubiquitous in programs, *e.g.*, a statement that calls a variable may be far from the definition of that variable, capturing long-range dependencies is critical for learning code representations [11], [13], [14]. However, most GNNs only learn local dependencies within small neighbourhoods, due to their message-passing mechanism. **Second**, many tree-structured neural models for code cannot learn useful node representations (see column "Nodes" in Table I). When structured deep learning models are needed to solve certain tasks, *e.g.*, identifying misused variables [10] or inferencing variable types [15], we need to make predictions based on (syntax tree or program graph) node representations. An effective node representation should contain the information about the node features as well as its contextual information. However, since the node information in many tree-structured neural networks is propagated in a uni-directional bottom-up manner, these models cannot obtain sufficient context information for low-level tree nodes, especially for leaf nodes [6], [9], [16]. **Third**, most existing structured-based approaches ignore the order information in programs (see column "Order" in Table I). Since the programs are highly ordered, *e.g.*, program statements are executed in specific orders according to their control flow, whereas the message passing mechanism of GNN is permutation invariant and cannot capture the order information. **Last**, the existing graph-based methods and some tree-based approaches require additional processing of the syntax tree, which brings large overhead and loses the original structural information of the syntax tree (see column "Original" in Table I). For example, building program graphs often requires experts to utilize various static analysis techniques [3], [10], and some of which are not adaptable to different program languages or program understanding tasks.

[§]Corresponding authors

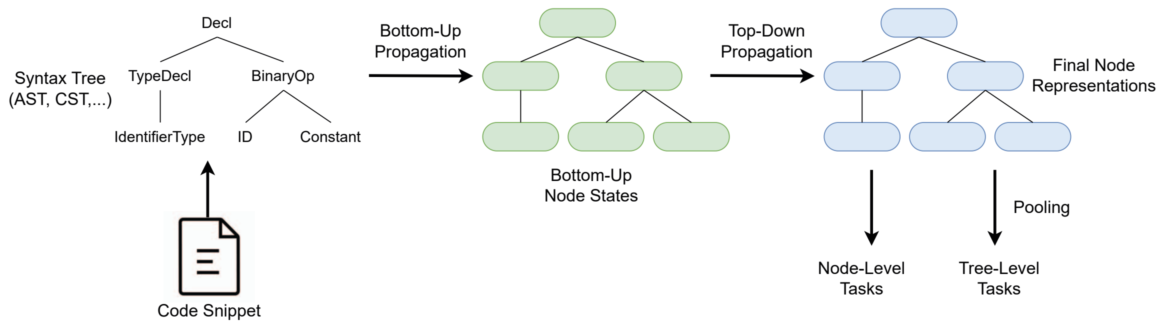


Fig. 1. The overall pipeline of Tree-Transformer.

To tackle the issues above, in this paper, we propose Tree-Transformer, a transformer-based approach to learning program representations for various downstream tasks. To model the global dependencies for all syntax tree nodes, we propose a multi-head attention-based bidirectional propagation method in opposite directions. The bidirectional propagation consists of two Tree-Transformer units, a bottom-up unit, and a top-down unit. The bottom-up unit aims to aggregate the contextual information from leaves to the root node , while the top-down unit aims to distribute the root information to its descendants. In this way, each node can capture the contextual information from all other nodes, which enables Tree-Transformer to learn long-range dependencies and meaningful node representations. Moreover, we adopt the position encoding mechanism of Transformer to realize the learning of sibling order information. Our contributions are summarized as follows:

- We propose Tree-Transformer, a recursive tree-structured neural network which formulates the parent-child and sibling relations on the trees with multi-head attention to learn vector representations for program syntax trees.
- We design a bottom-up and top-down bidirectional information propagation method on the Tree-Transformer to capture the global dependencies between any pair of syntax tree nodes and enable node-level prediction where previous tree-structured neural networks fail.
- We have conducted extensive experiments on tree-level and node-level prediction tasks on program syntax trees. The experimental results demonstrate that our approach significantly outperforms existing tree-structured neural networks, *e.g.*, Tree-LSTM, and graph neural networks, *e.g.*, GIN. For the node-level prediction task, we further design a wrong operator localization and repair task to evaluate the advantages of Tree-Transformer.

II. Motivating Example

Here, we take a specific example to demonstrate the weakness of existing program representation learning approaches on syntax trees, and the design motivation of Tree-Transformer.

Figure 2 (a) demonstrate a typical abstract syntax tree. When using graph neural networks on ASTs, node information is propagated bi-directionally along AST edges (see Figure 2 (b)). Most GNN models adopt the message passing mechanism [22], *i.e.*, each node only receives information from its k-th local neighborhoods (k is the number of GNN layers). For example, the dashed box in Figure 2 (b) is the 1-hop local neighborhood of node "assignment". Consequently, such GNN methods lack the capability of capturing longer dependencies.

TABLE I
A comparison of existing tree/graph-based program representation learning techniques.

Approach	Global	Nodes	Order	Original
GNN+AST	✗	✓	✗	✓
Child-Sum Tree-LSTM [16]	✓	✗	✗	✓
N-ary Tree-LSTM [16]	✓	✗	✓	✗
TBCNN [6]	✗	✗	✓	✓
TreeCaps [9]	✓	✗	✓	✓
Code2Vec [17]	✗	✓	✗	✗
Code2Seq [18]	✗	✓	✗	✗
ASTNN [7]	✓	✗	✓	✗
Tree-PE [19]	✓	✓	✓	✗
Code Transformer [5]	✓	✓	✓	✗
TreeBERT [20]	✓	✓	✓	✗
Tree-Transformer(Ours)	✓	✓	✓	✓
GNN+Augmented AST [10]	✗	✓	✓	✗
Devign [3]	✗	✓	✓	✗
GREAT [13]	✓	✓	✓	✗
HPG+HGT [21]	✗	✓	✓	✗

In most tree-structured neural networks [6], [16], [23], node information is propagated in a uni-directional bottom-up fashion (see Figure 2 (c)). This means that each node receives context information from its descendants. Low-level nodes, especially leaf nodes, cannot receive context from an adequate number of nodes, so the representations learned for these nodes are unlikely to perform well on node-level prediction tasks. For example, in Figure 2 (c), the leaf node "b" cannot receive any other information other than itself.

Another issue for GNNs and some tree-structured neural networks is that they cannot naturally handle the node order information in ASTs. To alleviate this issue, some works adopt order-sensitive neural networks [1], [4], [19], [24], which are mainly built for N-ary trees, especially binary trees. However, an AST node may have an arbitrary number of children, and converting ASTs to N-ary trees will destroy the original tree structure information. Figure 2 (d) shows an AST converted into a binary tree with left-child right-sibling (adopted by [19]), and we can obviously see that the original tree structure

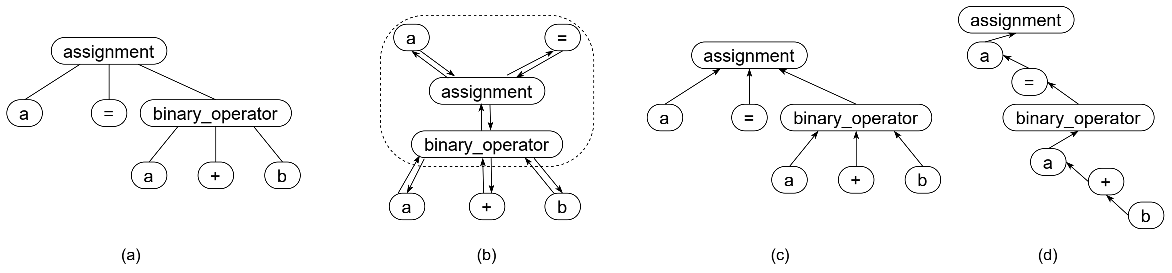


Fig. 2. An example of existing tree-structured and graph neural networks for learning on abstract syntax trees.

is vastly changed.

III. Approach

As shown in Figure 1, the Tree-Transformer consists of two consecutive steps:

- Bottom-up propagation: a bottom-up Tree-Transformer unit propagates the node messages recursively from children to parents to obtain the bottom-up states of nodes.
- Top-down propagation: a top-down Tree-Transformer unit distributes the learned contextual information from the parent node to their children.

After the bi-directional propagation of Tree-Transformer, the obtained top-down node states can be treated as final node representations, and further utilized for node-level prediction tasks (e.g. bug localization). We can also pool the node representations learned by Tree-Transformer into a single vector for tree-level prediction tasks (e.g. program classification). In the following we will describe the bi-directional propagation of Tree-Transformer in details.

A. Bottom-up Propagation

Formally, a code snippet C can be parsed into an AST $\mathcal{T} = (\mathcal{V}_{leaf}, \mathcal{V}_{non})$, where $\mathcal{V}_{leaf}$ and $\mathcal{V}_{non}$ denote the set of AST leaf nodes and non-leaf nodes respectively. Any node $i \in \mathcal{V}_{non}$ connects with a set of child nodes $\mathrm{c}_i = \{i_1, i_2, ..., i_n\}$ where n is the number of child nodes for i. In ASTs, different nodes may have different numbers of n. For any node $v \in \mathcal{T}$, we utilize a learnable embedding matrix $\mathbf{E}$ to obtain the initial node embedding $\boldsymbol{e}_v \in \mathbb{R}^d$, where d is the dimension of node embeddings.

To gather the children information for the non-leaf node i, we design the bottom-up propagation based on the multi-head attention [25]. The architecture of the bottom-up Tree-Transformer unit is shown in Figure 3 (a).

In multi-head attention, each attention head can be formulated as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V, \tag{1}$$

Where Q, K, V denote the query, key and value, and d is the dimension of key vectors.

The bottom-up Tree-Transformer unit obtain the bottom-up states of nodes in a bottom-up manner similar to recursive neural networks [26], i.e., a node i's bottom-up state $\boldsymbol{h}_{i\uparrow}$ is updated from its initial node embedding $\boldsymbol{e}_i$ and its children's bottom-up states $\boldsymbol{H}_{\mathrm{c}_i\uparrow} = (\boldsymbol{h}_{i_1\uparrow}, \boldsymbol{h}_{i_2\uparrow}, ..., \boldsymbol{h}_{i_n\uparrow})$. If the child nodes of i are leaf nodes, i.e., $\{i_1, ..., i_n\} \subseteq \mathcal{V}_{leaf}$, $\boldsymbol{H}_{\mathrm{c}_i\uparrow}$ are equal to their leaf node embeddings i.e., $\boldsymbol{H}_{\mathrm{c}_i\uparrow} = (\boldsymbol{e}_{i_1}, \boldsymbol{e}_{i_2}, ..., \boldsymbol{e}_{i_n})$.

In a bottom-up Tree-Transformer unit, we first apply a fraternal self-attention $\mathrm{MultiHead}_{f\uparrow}$ on $\boldsymbol{H}_{\mathrm{c}_i\uparrow}$ to model the sibling dependency between nodes in c_i. In the fraternal attention, the query, key and value all come from the children node bottom-up state sequence. Then we use a parental multi-head attention $\mathrm{MultiHead}_p$ to capture the dependency between i and its children. In the parental attention of i, the query is its initial node embedding $\boldsymbol{e}_i$, and the key/value are the output of the fraternal attention on i's children. The output of the parental attention is thereafter used to update the bottom-up state of i. Like the sequential transformer model, each multi-head attention operation in the Tree-Transformer unit is followed by a layer normalization and residual connection. Finally, we use a position-wise feed-forward layer same as [25] calculates the bottom-up state. Formally, the bottom-up Tree-Transformer unit calculates $\boldsymbol{h}_{i\uparrow}$, the bottom-up state of i by:

$$\boldsymbol{H}'_{\mathrm{c}_i\uparrow} = \mathrm{MultiHead}_{f\uparrow}(\boldsymbol{H}_{\mathrm{c}_i\uparrow}, \boldsymbol{H}_{\mathrm{c}_i\uparrow}, \boldsymbol{H}_{\mathrm{c}_i\uparrow}) \tag{2}$$

$$\boldsymbol{H}'_{\mathrm{c}_i\uparrow} = \mathrm{LayerNorm}(\boldsymbol{H}'_{\mathrm{c}_i\uparrow} + \boldsymbol{H}_{\mathrm{c}_i\uparrow}) \tag{3}$$

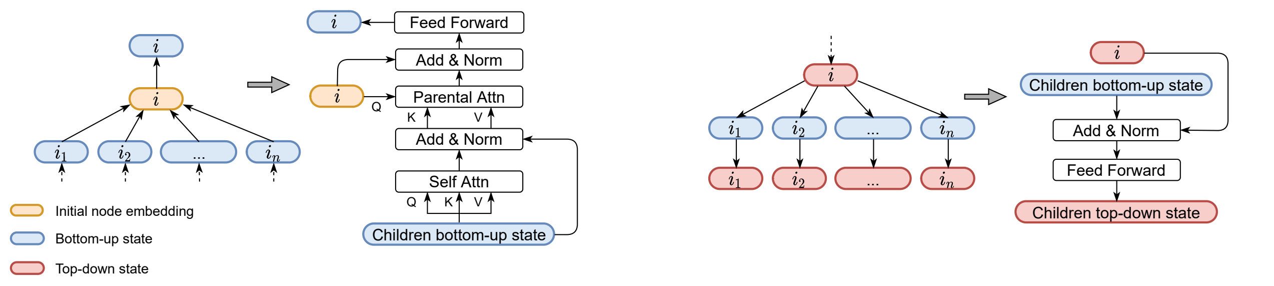
$$\boldsymbol{A}_{\uparrow} = \mathrm{MultiHead}_p(\boldsymbol{e}_i, \boldsymbol{H}'_{\mathrm{c}_i\uparrow}, \boldsymbol{H}'_{\mathrm{c}_i\uparrow}) \tag{4}$$

$$\boldsymbol{A}'_{\uparrow} = \mathrm{LayerNorm}(\boldsymbol{A} + \boldsymbol{e}_i) \tag{5}$$

$$\boldsymbol{h}_{i\uparrow} = \mathrm{LayerNorm}(\mathrm{FFN}_{\uparrow}(\boldsymbol{A}'_{\uparrow}) + \boldsymbol{A}'_{\uparrow}) \tag{6}$$

To capture the sibling order information in syntax trees, we apply position encodings in the fraternal self-attention $\mathrm{MultiHead}_{f\uparrow}$. This allows our model to handle sibling orders for arbitrary trees, while many existing order-sensitive models on trees [16], [19] only work on trees with a fixed branching factor (N-ary trees).

Different from the standard position encoding of Transformer [25], we adopt the Untied Positional Encoding (TUPE) [27] in the bottom up fraternal attention $\mathrm{MultiHead}_{f\uparrow}$. In traditional position encoding, the token embeddings and position embedding vectors are added before the multi-head attention, this makes the model cannot distinguish token information and position information after these two are mixed up. The TUPE position encoding unties the position information from the token information, this enables the



(a): Bottom-up unit (left) and its detail (right)

(b): Top-down unit (left) and its detail (right)

Fig. 3. The detailed architecture of Tree-Transformer units. (a): Bottom-up Tree-Transformer unit. (b): Top-down Tree-Transformer unit.

Transformer model to learn more precise position information without the interference of input tokens. When we use TUPE in Tree-Transformer, the output $\boldsymbol{H}'_{c_i\uparrow} = (z_1, z_2, ..., z_n)$ in Equation (2) is computed by:

$$\boldsymbol{z}_m = \sum_{j=1}^{n} \frac{\exp(\boldsymbol{\alpha}_{mj})}{\sum_{j'=1}^{n} \exp(\boldsymbol{\alpha}_{mj'})} (\boldsymbol{h}_{i_j\uparrow} \boldsymbol{W}^V) \tag{7}$$

$$\alpha_{mj} = \frac{1}{\sqrt{2d}} (\boldsymbol{h}_{i_m\uparrow} \boldsymbol{W}^Q)(\boldsymbol{h}_{i_j\uparrow} \boldsymbol{W}^K)^T + \frac{1}{\sqrt{2d}} (\boldsymbol{p}_m \boldsymbol{U}^Q)(\boldsymbol{p}_j \boldsymbol{U}^K)^T \tag{8}$$

Where $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V \in \mathbb{R}^{d \times d}$ are query/key/value projection matrices and $\boldsymbol{U}^Q, \boldsymbol{U}^K \in \mathbb{R}^{d \times d}$ are (absolute) position projection matrices for queries and keys. $\boldsymbol{p} \in \mathbb{R}^d$ are the fixed positional embedding vectors.

B. Top-down Propagation

After bottom-up propagation, Tree-transformer obtains the bottom-up states of all nodes in AST i.e., $\{\boldsymbol{h}_{v\uparrow} | \forall v \in \mathcal{V}\}$. Tree-Transformer then performs the top-down propagation, which is shown in Figure 3 (b). When the top-down propagation is finished, each node can obtain the contextual information from all other nodes, thus enables to capture the global dependency which is missed in most existing tree/graph-structured neural networks.

The top-down Tree-Transformer unit uses the state $\boldsymbol{h}_{i\downarrow}$ of a single node i to simultaneously update its children's bottom-up states $\boldsymbol{H}_{c_i\uparrow} = \{\boldsymbol{h}_{i_1\uparrow}, \boldsymbol{h}_{i_2\uparrow}, ..., \boldsymbol{h}_{i_n\uparrow}\}$ into top-down states $\boldsymbol{H}_{c_i\downarrow} = \{\boldsymbol{h}_{i_1\downarrow}, \boldsymbol{h}_{i_2\downarrow}, ..., \boldsymbol{h}_{i_n\downarrow}\}$. If i is the root node, $\boldsymbol{h}_{\text{root}\downarrow} = \boldsymbol{h}_{\text{root}\uparrow}$. In the top-down parental attention, we aim to use a top-down parental attention to pass information from parent to children. In contrast to the bottom-up parental attention, in the top-down attention, children states $\boldsymbol{H}_{c_i\uparrow}$ are used as query, and their parent top-down state $\boldsymbol{h}_{i\downarrow}$ as key/value. This is not a common case for multi-head attention, because when the length of key/value is 1, the softmax function over keys/values is meaningless (it will always output 1). So we make a slight change and simplification to the top-down parental "attention" function. Instead of computing attention scores, we directly add the top-down states of the parent node to all its children (this acts similar to a residual connection, the attention is omitted). The calculation process of the top-down unit is demonstrated below:

$$\boldsymbol{A}_{\downarrow} = \text{LayerNorm}(\mathbf{1} \cdot \boldsymbol{h}_{i\downarrow} + \boldsymbol{H}_{c\uparrow}) \tag{9}$$

$$\boldsymbol{H}_{c\downarrow} = \text{LayerNorm}(\text{FFN}_{\downarrow}(\boldsymbol{H}'_{c\downarrow}) + \boldsymbol{H}'_{c\downarrow}) \tag{10}$$

After top-down propagation, Tree-Transformer obtains the top-down node states $\{\boldsymbol{h}_{v\downarrow} | \forall v \in \mathcal{V}\}$ which are used as the final node representations.

By combining bottom-up and top-down propagation, Tree-Transformer is capable for modeling dependencies between any node pairs along paths with arbitrary lengths. On the contrary, although traditional Transformers can capture global dependencies, they can only model paths with a maximum length (the number of Transformer layers).

C. Calculating Tree-Level Representation for Program-Level Prediction

Different from previous (recursive) tree-structured neural networks, which use the state of root nodes as representation vector for trees, we use a pooling function over the top-down states for all nodes $\{v | v \in \mathcal{T}\}$ to obtain the final representation of a tree $\mathcal{T}$. We adopt the global attention pooling function proposed in [28]:

$$\mathbf{h}_{\mathcal{T}} = \sum_{v \in \mathcal{T}} \text{softmax}(\boldsymbol{W}_{\text{gate}} \boldsymbol{h}_{v\downarrow}) \odot \boldsymbol{h}_{v\downarrow} \tag{11}$$

$\boldsymbol{W}_{\text{gate}}$ is a weight of $\mathbb{R}^d$, $\odot$ is the element-wise multiplication and $\mathbf{h}_{\mathcal{T}}$ can be utilized as the tree-level prediction.

IV. Experimental Setup

In this section, we first introduce the selected tasks and baselines for evaluation, then present the settings for conducting out experiments.

A. Tasks and Datasets

We select three different tasks to evaluate the learning capacity of Tree-Transformer on learning tree-level and node-level representations. The detailed statistics of all datasets are listed in Table II.

TABLE II
BASIC STATICTICS OF THE DATASETS WE USE IN THIS PAPER. FOR CODENET DATASETS, THEIR VOCABULARY SIZE IS THE SIZE OF THEIR TOKEN VOCABULARY PLUS TYPE VOCABULARY.

	POJ	Java250	Python800	C++1000	C++1400	Wrong Operator	Type Inference
Train samples	36,400	45,000	144,000	300,000	252,000	155,628	608,156
Validation samples	5,200	15,000	48,000	100,000	84,000	16,868	24,424
Test samples	10,400	15,000	48,000	100,000	84,000	86,231	27,870
Avg. nodes	189.58	339.33	232.27	376.90	472.04	222.39	652.53
Avg. children per node	1.90	3.05	2.77	3.09	3.12	2.84	3.03
Avg. depth	13.32	17.26	14.48	15.46	16.43	13.28	17.09
Vocabulary	44	222	161	346	346	286,456	100,128 (manually set)

*1) **Program Classification**:* In this task, we use Tree-Transformer to classify program ASTs based on the functionalities they implemented. We select this task to measure the ability of Tree-Transformer on learning tree-level representations. Specifically, we use two different datasets for evaluation. The first dataset is POJ algorithm classification dataset [6], which has been widely adopted to evaluate the capability of program representation models. POJ dataset contains 104 classes of C programs from student programming platforms. In our experiment, we follow the AST parsing process of [6]: using pycparser[1] to parse the functions to obtain ASTs. The second one is the CodeNet dataset [29], which contains over 14M code samples from two open judge platforms. Here we use the code classification benchmarks, which include four classification datasets in three programming languages: Java250, Python800, C++1000, and C++1400. These four datasets contain programs in 250, 800, 1000, and 1400 classes, respectively. Since CodeNet has already provided simplified parse trees (SPT) for those benchmarks, we directly use them for evaluation. CodeNet SPTs are generated by the ANTLR4 [30] parser with a series of post-processing steps, including removing internal nodes with only one child. Notice that in both program classification datasets, the identifier names are discarded in syntax trees because we aim to perform classification on the algorithms alone without the additional information of identifier names. In open judge platforms, programmers tend to name identifiers according to the description of the programming problems, and the naming patterns may bring additional guidance for the program classifiers.

*2) **Wrong Operator Localization and Repair**:* In order to evaluate Tree-Transformer on node-level prediction, we propose a novel tree-based wrong operator localization/repair task. Given the syntax tree of a code snippet with an erroneous binary operator (e.g., changing "+" into "-"), this task requires a model to locate the position of the misused operator node among all binary operator nodes, and predict the correct operator for this position. We synthesize a new dataset from the wrong operator detection dataset, released by CuBERT [31]. The original dataset is built for the binary classification between correct and buggy code snippets. To enable the localization and repair task, we only keep code snippets with more than one binary operator. On average, each code snippet in our dataset contains 5.98 binary operators. For this dataset, we use tree-sitter[2] to parse source code into concrete syntax trees (CST). CSTs contains more nodes than ASTs, mainly including brackets and punctuation.

*3) **Type Inference**:* In this task, we utilize Tree-Transformer to classify the type of identifiers in TypeScript, a dynamically-typed language. Similar to wrong operator localization/repair, this task can also be seen as a prediction task on syntax tree nodes. For this task, we employ the public type inference dataset ManyTypes4TypeScript [32], which consists of 13,953 public Github projects. We use the tree-sitter TypeScript parser to parse code snippets into CSTs and adopt a linear classifier on identifier nodes for predicting identifier types. We follow the original settings of ManyTypes4TypeScript and choose the vocabulary of identifier types as 50,000, so this task is a node classification task with 50,000 classes.

B. Compared Baselines

We compare our approach against existing tree-structured and graph-structured neural networks. We further compare with some transformer-based models built for programming languages. To sum up, we choose the following models as baselines:

- **Tree-structured neural networks**. We compare Tree-Transformer with Tree-LSTM [16], TBCNN [6] and TreeCaps [9]. As program syntax trees can have arbitrary branching factors, we use Child-Sum Tree-LSTM as our baseline. For TreeCaps, we use the variable-to-static (VTS) version in our experiments.
- **Graph neural networks**. We choose graph convolutional network (GCN) [33], graph isomorphism network (GIN) [34], and gated graph neural network (GGNN) [28] as our baselines. We evaluate the GNN baselines with two different sets of inputs: the original syntax tree and syntax tree augmented with NextToken edges (connect a terminal token node to the next terminal) [10], [29].
- **Transformer-based models**. We choose two tree/graph-based Transformer models as our baselines: Tree-PE [19] and GREAT [13]. As Tree-PE can only handle N-ary trees, we convert the input trees to 10-ary trees for program classification and 15-ary for wrong operator localization and

[1]https://github.com/eliben/pycparser

[2]https://tree-sitter.github.io/tree-sitter/

repair. As GREAT usually takes graphs with multiple edge types as inputs, we use syntax trees with NextToken edges as its inputs. We also compare our approach with sequential Transformer models on source code token sequences. Our sequential baselines including a vanilla Transformer [25] and pre-trained models: CodeBERT [35] and C-BERT [36]. Different from tree/graph-based approaches, in sequence-based approaches, we follow previous works [29] and use the token sequences of the original code, which means the identifier names are kept.

C. Experimental Settings

1) General Settings: We set the Tree-Transformer node embedding dimension to 128 for POJ and 256 for other datasets. The number of attention heads is set to 4. We train our models with an Adam optimizer with a default learning rate of 0.002 and a warm-up phase of 2,000 steps. We implement Tree-Transformer with DGL [37] to enable efficient batching ,and ran our experiments on a NVIDIA RTX 8000 GPU with 48GB memory.

2) Settings for Program Classification: For the POJ dataset, we follow previous works [9] and split the dataset into train/validation/test sets by the ratio of 7:1:2. For CodeNet, the train/validation/test ratio is 3:1:1. Since each node in CodeNet SPTs contains two parts of information: parsing rules and tokens, thus we concatenate the token embeddings and the parsing rule embeddings as the initial node embeddings. For GNN baselines, we adopt the same pooling function as Tree-Transformer. For Tree-LSTM, we employ two different approaches to acquire the representation vectors for trees: using the root node's hidden state or using the same attention pooling as our model. For TreeCaps, we follow its original setting and use its "code capsule" to compute the probabilities of output classes.

3) Settings for Wrong Operator Localization and Repair: Locating the wrong operator node is achieved by learning a pointer pointing to a single node in a tree, which is the same as previous works on localization and repair tasks [13], [38]. Unlike [38], which also uses a pointer for the repair task, We treat this step as a node classification task: a classifier predicts the label of the repair operator within the set of all binary operators. In our dataset, the operator set $OPs = \{-, +, *, \%, >, ==, \text{or}, <, /, \text{and}, >=, <=, != , \text{in}, \text{is}, \text{is not}, \text{not in}\}$. We sum up the localization loss and the repair loss as the training loss for this task.

In this task, all the wrong operator nodes are located on the leaf nodes in CSTs. However, the tree-structured neural networks, e.g., Tree-LSTM, follow a bottom-up manner to propagate information, which means that the leaf nodes cannot receive the information from other nodes and cannot learn well-contextualized node representations. Thus we directly omit these tree-structured baselines and only utilize graph-based models for comparison.

4) Settings for Type Inference: The original ManyTypes4TyprScript dataset is created for token-base type inference on code token sequences, which cannot be directly converted to tree-base data without any data loss. In the parsing step, we remove code snippets that cannot be correctly parsed by tree-sitter and CSTs with larger than 5000 nodes, thus resulting in a slightly smaller dataset than the original one. The vocabulary size for CST terminal token nodes is manually set to 100,000 for this task. Similar to wrong operator localization and repair, we only compare with baselines that are capable of node-level predictions.

V. Experimental Results

In this section, we present and analyze our experiment results to address the following research questions:

- **RQ1**: How does Tree-Transformer perform on syntax tree-level prediction tasks?
- **RQ2**: How does Tree-Transformer perform on node-level prediction tasks?
- **RQ3**: How does each component of Tree-Transformer contribute to our performances?

A. RQ1: How does Tree-Transformer perform on syntax tree-level prediction tasks?

Table III shows the classification results on CodeNet and POJ datasets. We can see that on all five datasets, Tree-Transformer outperforms the tree-structured and graph-structured baselines by a significant margin. This highlights the effectiveness of modeling global dependencies along trees with multi-head attention. When using the attention pooling the same as Tree-Transformer, Tree-LSTM does not show significant improvements, suggesting that our improvements over Tree-LSTM mainly come from our model design rather than the pooling strategy. Moreover, Tree-Transformer outperforms all of our graph-structured baselines (including GNNs and GREAT), even when they are integrated with additional NextToken edges. Although adding additional edges to syntax trees (see rows "GNN+Graph") can marginally improve the performances of program classification, it still cannot overcome the inherent weaknesses of graph-based models. We also include a heterogeneous graph-based model (HPG+HGT) [21] for comparison. Although HPG+HGT outperforms (homogeneous) GNN baselines by introducing additional node and edge type information, it still cannot compete against our Tree-Transformer.

From the results, we notice that Tree-LSTM outperforms the GNN baselines when the given inputs are trees. Although the research interest in tree-structured neural networks is undermined by the advance of GNNs, tree-structured models are still competitive and should not be ignored. When compared with large-scale pre-trained Transformers, we find that Tree-Transformer is still competitive. The average accuracies of C-BERT and CodeBERT on CodeNet is lower than ours, although for some datasets, for example, Java250 and Python800, they have a higher performance than ours. Compared with pre-trained baselines, which require massive data for pre-training, Tree-Transformer is much more light-weight, and this further confirms the effectiveness of our model.

TABLE III
PROGRAM CLASSIFICATION ACCURACY(%) ON CODENET AND POJ DATASETS.

	Java250	Python800	C++1000	C++1400	Overall	POJ
GCN	89.06	91.81	93.54	92.89	91.83	93.93
GIN	90.76	93.17	95.54	94.50	93.49	95.76
GGNN	88.46	89.92	89.75	88.01	89.04	93.22
Tree-LSTM (root)	93.19	93.95	95.79	95.20	94.53	94.70
Tree-LSTM (attention)	93.71	93.83	95.79	95.24	94.64	94.95
TBCNN	90.32	91.10	93.17	93.03	91.91	94.15
TreeCaps	91.42	90.26	93.55	93.24	92.12	95.88
Tree-PE [19]	91.65	91.11	92.30	88.97	91.01	94.19
GREAT	93.15	93.30	93.46	93.72	93.41	92.64
GCN+Graph [29]	92.70	93.82	95.76	95.26	94.39	95.57
GIN+Graph [29]	93.26	94.17	96.34	95.95	94.93	95.96
GGNN+Graph	93.61	92.23	91.72	92.48	92.51	94.80
HPG+HGT [21] [3]	93.95	94.99	-	-	-	-
Tree-Transformer (Ours)	95.32	95.30	**97.11**	**96.98**	**96.18**	**96.12**
Transformer	93.49	93.99	89.93	67.87	86.32	88.13
C-BERT [29]	**97.40**	**97.09**	93.79	91.83	95.03	-
CodeBERT	96.47	97.41	86.13	83.05	90.77	98.40

An interesting finding on pre-trained models is that both pre-trained baselines perform very well on Java250 and Python800, but their accuracies are low on CodeNet C++ datasets, even though these two models are pre-trained on completely different corpora (CodeBERT is pre-trained on the CodeSearchNet dataset [39] (without C/C++ code snippets), while C-BERT is pre-trained on Github C repositories). To further understand this unexpected phenomenon, we make an analysis of CodeNet datasets based on token sequences, which is demonstrated in Table IV. In this table, we use code token sequences tokenized by the CodeBERT BPE tokenizer. We first list the lengths of token sequences, and it shows that the token sequences for C++ datasets are longer than Java/Python datasets. This brings a common weakness of most existing Transformer-based pre-trained models: these models require a fixed maximum input length due to the limitation of memory cost. For example, the maximum sequence length of CodeBERT and C-BERT is 512, which is shorter than the average sequence length of the C++1400 dataset. The overlength inputs are cropped to the maximum length and may harm the model from understanding the program's original semantics. We also make an analysis of the differences between each program classes at the token level. We treat the programs from the same class as a single text snippet by concatenating them and calculate the TF-IDF distances between every two classes. The average TF-IDF cosine similarity between all classes is reported in Table IV. We can see that the average similarity scores of C++ datasets are higher, which means that in these datasets, programs from different classes are more similar at the token level compared to Java/Python datasets. As Transformer-based pre-trained models are conducted on token sequences, this "similarity" of tokens may hinder the models from learning the differences between program functionalities. In fact, as previous works [40] have pointed out, Transformer models for code can be more sensitive to identifiers than logical structures, which further supports our findings.

TABLE IV
STATISTICS OF CODENET DATASETS IN TOKENIZED SEQUENCES.

	Java250	Python800	C++1000	C++1400
Sequence length	444.5	207.8	488.8	583.2
Average TF-IDF	0.722	0.420	0.787	0.754

B. RQ2: How does Tree-Transformer perform on node-level prediction tasks?

Table V demonstrates the results of wrong operator localization and repair. We report two accuracy metrics: localization accuracy and joint accuracy of localization and repair. We can find that compared with graph-structured baselines, Tree-Transformer achieves significantly better performance, especially on the joint loc&rep accuracy. Our model gains an improvement of 7% on joint accuracy compared with the best-performing baseline GIN. This indicates that our bi-directional propagation enables the model to learn effective node representations for node-level prediction tasks. The Transformer-based model for trees [19] performs poorly on this task, its accuracies lower than all GNN baselines. This suggests that changing the original tree structures by converting arbitrary trees to N-ary trees is harmful to node-level prediction. In the wrong operator dataset, the branching factor of 10% syntax trees is larger than 15, so the structures of these trees are changed for this baseline.

In this task, the pre-trained model CodeBERT outperforms Tree-Transformer and achieves extremely high accuracy on both localization and repair. However, we must be aware that CodeBERT requires a large model size and pre-training on over 8 million data samples (while our model can be trained on a single GPU and does not require pre-training). A

[3]HPG [21] can only build program graphs for Java and Python.

previous work [31] has suggested that large pre-trained models can significantly outperform trained-from-scratch models and achieve very high results on synthesized localization & repair tasks (e.g., variable misuse [10], [38] and our WrongOp task). We believe this is because those pre-trained models excel in capturing the ”naturalness” of input code from a large amount of data, so they easily recognize synthesized bugs that are ”unnatural.”

TABLE V
THE ACCURACY OF WRONG OPERATOR LOCALIZATION AND REPAIR.

Model	Loc Acc(%)	Loc & Rep Acc (%)
GCN	84.97	60.44
GIN	85.69	61.61
GGNN	84.77	58.71
Tree-PE [19]	78.65	53.99
GREAT	85.43	59.32
GCN+Graph	86.37	65.05
GIN+Graph	86.48	64.66
GGNN+Graph	84.11	61.01
Tree-Transformer (Ours)	88.26	68.58
Transformer	73.13	45.22
CodeBERT	**94.61**	**83.89**

The results for type inference are shown in Table VI. Tree-Transformer outperforms the GNN baselines with a large gap. This further demonstrates the strength of Tree-Transformer, as some predictions on variable types require long-dependency reasoning. For example, a user-defined type could be used long after its definition code block. Similar to the wrong operator dataset, Tree-Transformer outperforms vanilla Transformer on this type inference dataset, but the results are lower than the pre-trained model CodeBERT. The large performance gap between trained-from-scratch Transformer and pre-trained Transformer indicates that our Tree-Transformer may also benefit from pre-training techniques and further improve its performance. We will leave this as our future work.

C. RQ3: How does each component of Tree-Transformer contribute to our performances?

We perform an ablation study to further investigate the impact of each component in Tree-Transformer. Table VII shows the results of Tree-Transformer when removing each component on program classification (Java250) and wrong operator localization. In this table, “-” means removing a certain component from Tree-Transformer. The experiment of removing top-down propagation on wrong operator localization is omitted because leaf nodes cannot receive the information from their parent nodes only through bottom-up propagation. Note that after removing fraternal attention, the position encoding within it is also removed accordingly. To separate position encodings from the fraternal attention, we add a new variant of Tree-Transformer where fraternal attention is removed, and position embedding vectors are added before the bottom-up parental attention.

From the results, first, we can see that our top-down propagation is succinct and powerful on both tasks. Without it, the performance drops greatly, which indicates that it is effective on both node-level prediction and tree-level classification tasks. Furthermore, even with only bottom-up propagation, Tree-Transformer still outperforms tree-structured baselines, showing that our attention-based neural network unit is effective. Moreover, modeling fraternal dependencies and sibling positions also contribute to both tasks. However, the effect of position encoding in Java250 is less significant than in WrongOp: removing position encodings in Java250 hardly affects the classification accuracy. This may indicate that in our program classification datasets, sibling order information is not essential for distinguishing programs of different classes. On the contrary, sibling orders in wrong operator localization are more important because locating wrong operators requires reasoning on relationships between operators and operands, and sibling dependency is a key part of these relationships.

TABLE VI
THE ACCURACY OF TYPE INFERENCE ON MANYTYPES4TYPESCRIPT CST DATA.

Model	Accuracy(%)
GCN	46.35
GIN	47.40
GGNN	45.47
Tree-PE [19]	51.71
GREAT	52.45
GCN+Graph	46.75
GIN+Graph	47.91
GGNN+Graph	45.71
Tree-Transformer (Ours)	54.78
Transformer	49.54
CodeBERT	**65.76**

TABLE VII
ABLATION STUDY ON PROGRAM CLASSIFICATION (JAVA250) AND WRONG OPERATOR LOCALIZATION AND REPAIR.

Model	Java250	WrongOp	
		Loc	Loc & Rep
Tree-Transformer	**95.32**	**88.26**	**68.58**
-position encoding	94.99	85.78	63.32
-fraternal attention	94.66	86.26	63.91
-fraternal attention +position encoding	94.95	87.18	65.64
-top-down propagation	94.01	N/A	N/A

An interesting finding is that only removing position encodings (keeping the fraternal attention) results in worse accuracies than jointly removing position encodings and fraternal attention on the WrongOp dataset. Because fraternal attention alone cannot learn from the sibling order information, adding this attention without given position information will deepen the model and make Tree-Transformer harder to train. If we keep the position encoding and only remove the fraternal attention, the results are still lower than the complete model. This suggests that position encoding works better when integrated with fraternal attention in Tree-Transformer.

VI. DISCUSSION

In this section, we mainly discuss the strengths and limitations of Tree-Transformer compared with existing

Transformer-for-code models, mainly from the perspective of efficiency. We also discuss the threats to validity of our work.

A. Strengths

The model structure of our Tree-Transformer is largely different from existing Transformer-based syntax tree modeling approaches: Tree-Transformer basically follows the tree-structured recursive formula [26], [41], while existing approaches [5], [20], [42], [43] usually uses a sequential Transformer with node traversal sequences as inputs.

A main advantage of Tree-Transformer over sequential Transformer-based approaches is that Tree-Transformer can handle larger inputs. The core of Transformer is its multi-head attention mechanism. In a sequential Transformer encoder, the model performs self-attention on its input sequence. For a program syntax tree $\mathcal{T}$ with N nodes, the time and memory cost of self-attention on sequential Transformers is $O(N^2)$. This means that sequential Transformer on long sequences may take up a large amount of memory. To avoid out-of-memory problems, existing sequential approaches often set a not-so-long limit for their inputs, making them difficult to deal with long programs. For example, TPTrans [42] assign a maximum number 512 for input CST leaf nodes. AST-Trans [43] requires the input ASTs not larger than 200 nodes. On the other hand, in Tree-Transformer, there are two multi-head attentions: the fraternal self-attention between siblings and a parent-children between a group of siblings and their mutual parent. Suppose the branching factor of $\mathcal{T}$ is k, then the memory cost of fraternal attention is $O(k^2)$, and for parent-children attention is $O(k)$. In most syntax trees, $k \ll N$, so the memory cost of Tree-Transformer is significantly less than sequential Transformer. This allows Tree-Transformer to model syntax trees larger than the capacity of sequential Transformer-based approaches. For example, we do not apply any limitations on the syntax tree size for program classification and wrong operator localization/repair.

B. Limitations

The main limitation of Tree-Transformer, along with other recursive tree-structured models (e.g., Tree-LSTM), is that it runs slower than sequential Transformers or GNNs. In Tree-Transformer, before we update a node's state in the bottom-up pass, we must update all its children first and vice versa in the top-down pass. So for a syntax tree $\mathcal{T}$ with depth $\mathcal{D}$, the bottom-up/top-down Tree-Transformer unit must run for $2\mathcal{D}$ timesteps before we get the final node representations for $\mathcal{T}$. However, in sequential Transformer or GNNs with L layers, their Transformer/message passing unit needs to run for L times. For most program syntax trees and Transformer/GNN models, $L < 2\mathcal{D}$ (e.g., our GREAT baseline only consists of 6 layers), so generally, the time efficiency of Tree-Transformer is lower than sequential Transformer/GNNs.

C. Experiments on Efficiency

To make a clear demonstration of the efficiency of Tree-Transformer, we compare the running time and memory occupation of Tree-Transformer with three baseline models: GIN, vanilla Transformer, and CodeBERT on the Java250 dataset. The results are shown in Table VIII. We can generally find out that the memory efficiency of Tree-Transformer is much higher than Transformer/GIN, while its time efficiency is lower. When the batch size is 256, which is the default setting in our experiments, the training time of Tree-Transformer is about four times as GIN and three times as Transformer, which is totally acceptable. As for the memory usage, Tree-Transformer only takes around $\frac{1}{5}$ of vanilla Transformer. When we increase the batch size to 1024, vanilla Transformer encounters the out-of-memory problem (we were using a GPU with 48GB memory), while Tree-Transformer only uses about 18GB memory. Generally, the memory effectiveness of Tree-Transformer makes it suitable for large input trees and large training batch sizes. As for the large pre-trained model CodeBERT, its training(fine-tuning) and inference time are both much longer than small-sized models, and its memory cost is higher even with a much smaller batch size.

D. Threats to Validity

The first threat involves the data processing step, which is the parsing of syntax trees in our experiments. To reduce this threat, we choose open source parsers which are evaluated by previous researchers on different downstream tasks. For the off-the-shelf syntax trees provided by the dataset creators, they are also parsed from widely-used parsers.

Another threat relates to the datasets we used in this paper. In the type inference task, we adopt the public ManyTypes4TypeScript benchmark, but filtered out some data samples to suit our experiments. This makes the dataset in this paper different from the public version. To reduce this threat, we ran all baseline approaches on the same filtered dataset as our Tree-Transformer. We will also make our preprocessing pipeline and preprocessed dataset public.

VII. Related Work

In this section, we first briefly introduce existing tree-structured neural networks, then summarize their application in software engineering along with other structure-based deep learning approaches for code.

A. Tree-Structured Neural Networks

There has been long research interest in applying deep neural networks to tree-structured data. The earliest work on tree-structured neural networks is recursive neural network [26], [41]. Recursive neural networks calculate the representation of a tree by accumulating node representations in a bottom-up manner. The recursive architecture further inspires later works [8], [16], [23], [44]. For example, Tai et al. [16] proposed Tree-LSTM, which uses an LSTM unit to replace the fully-connect neural network unit in recursive neural networks. Ahmed et al. [23] proposed a recursive Transformer by performing self-attention on siblings. Although many classical tree-structured neural networks are proposed in the natural language processing community, in recent years, software

TABLE VIII
COMPARISON OF TIME AND MEMORY EFFICIENCY BETWEEN TREE-TRANSFORMER, SEQUENTIAL TRANSFORMER AND GIN.

	GIN		Transformer		Tree-Transformer		CodeBERT	
Batch size	256	1024	256	1024	256	1024	32	64
Training time (s/batch)	0.13	0.45	0.42	-	1.09	2.53	0.61	1.17
Training time (s/epoch)	23	20	73	-	187	111	864	824
Inference time (s/batch)	0.05	0.17	0.12	-	0.51	0.86	0.21	0.41
Inference time (s/epoch)	2	2	7	-	30	13	99	97
Memory (MB)	15748	24348	36120	OOM	5729	18779	23821	46630

engineering researchers have started to propose tree-structured models based on the programming language domain. For example, Mou et al. [6] proposed tree-based convolutional neural network (TBCNN). Bui et al. [9] further proposed TreeCaps, which extends TBCNN with capsule networks [45] and achieved state-of-the-art results on program classification. Wang et al. [8] further extended Tree-LSTM with different neural submodules for children state aggregation.

B. Modeling Code Structures with Neural Networks

The above tree-structured neural networks have been widely applied in various software engineering tasks, such as code classification [6], [7], [9], code summarization [4], [9], code search [24], etc. In recent years, as the popularity of graph neural networks grows, researchers started to apply GNN on programs. Some works directly use GNNs to model program syntax trees [29], [46], while other works tried to augment syntax trees with additional hand-crafted edges [2], [10] or leverage control-flow/data-flow graphs [3], [24], [47]–[49].

Some other works tried to extract substructures from syntax trees, and use sequential (or partly sequential) neural networks to model code substructures. For example, Zhang et al. [7] proposed ASTNN, which decomposes an AST into a sequence of subtrees following the order of statements, and combines GRU and recursive neural network to encode the subtree sequence. Alon et al. [17], [18] sample paths between nodes from ASTs, and use feed-forward neural networks/LSTM to encode the set of AST paths.

Recently there have been some attempts to integrate the tree/graph structures of code into the popular Transformer model. Most of these works manipulate the attention mechanism or position encoding of Transformers to provide the model with structural information [5], [19], [42], [50]. For example, Shiv et al. [19] proposed a novel position encoding technique to extend Transformers to N-ary trees, and evaluate their approach on a code translation task. Hellendoorn et al. [13] proposed Graph Relational Embedding Attention Transformers (GREAT) to model program graphs. Guo et al. [51] proposed a pre-trained model GraphCodeBERT, which integrates simplified data flow structure into code token sequences. Zügner et al. [5] proposed Code Transformer, which integrates multiple distance metrics of AST nodes into the relative position encoding of Transformers. Jiang et al. [20] proposed TreeBERT, a Transformer-based pre-trained model on ASTs. TreeBERT extracts paths from ASTs as the inputs to the transformer model, and proposed a novel tree-based position encoding for nodes. Peng et al. [42] proposed TPTrans, which integrates the information of paths in CSTs by encoding the paths as the position encodings in a Transformer model. Guo et al. [52] proposed a pre-trained model Unixcoder, which uses the structure information of ASTs in the pre-training stage. However, the tree structure is not used in the fine-tuning/inference stage.

VIII. CONCLUTION

In this paper, we propose a novel tree-structured network: Tree-Transformer for program representation learning. Tree-Transformer leverages the powerful multi-head attention in two dimensions: fraternal and parental, to capture the dependencies between siblings and ancestors/predecessors on trees. Motivated by the neglect of the global dependency among nodes in current recursive-structured neural networks or GNNs, we propose bidirectional information propagation along trees and extend existing recursive neural network architecture with a novel top-down unit. Experiments on three different tasks: program classification, wrong operator localization&repair, and type inference have demonstrated the effectiveness of our proposed approach. In the future, we would like to further explore the potential of our Tree-Transformer with pre-training techniques. Another possible improvement is to integrate Tree-Transformer with GNNs so that the model can simultaneously handle syntactic structure information and control/data flows.

ACKNOWLEDGMENTS

This research is partially supported by the National Research Foundation, Prime Ministers Office, Singapore, under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019).

REFERENCES

[1] H.-H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 2017, pp. 3034–3040.

[2] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, “Detecting code clones with graph neural network and flow-augmented abstract syntax tree,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[3] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Advances in Neural Information Processing Systems 2019*. Neural Information Processing Systems (NIPS), 2019.

[4] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[5] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, “Language-agnostic representation learning of source code from structure and context,” in *International Conference on Learning Representations*, 2021.

[6] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[7] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 783–794.

[8] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, “Modular tree network for source code representation learning,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–23, 2020.

[9] N. D. Bui, Y. Yu, and L. Jiang, “Treecaps: Tree-based capsule networks for source code processing,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 30–38.

[10] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *International Conference on Learning Representations*, 2018.

[11] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, “Retrieval-augmented generation for code summarization via hybrid gnn,” in *International Conference on Learning Representations*, 2021.

[12] P. Fernandes, M. Allamanis, and M. Brockschmidt, “Structured neural summarization,” in *International Conference on Learning Representations*, 2019.

[13] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, “Global relational models of source code,” in *International conference on learning representations*, 2020.

[14] S. Liu, X. Xie, L. Ma, J. Siow, and Y. Liu, “Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search,” *arXiv preprint arXiv:2111.02671*, 2021.

[15] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, “Typilus: Neural type hints,” in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[16] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 1556–1566.

[17] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[18] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in *International Conference on Learning Representations*, 2019.

[19] V. Shiv and C. Quirk, “Novel positional encodings to enable tree-based transformers,” *Advances in Neural Information Processing Systems*, vol. 32, pp. 12 081–12 091, 2019.

[20] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, “Treebert: A tree-based pre-trained model for programming language,” in *UAI 2021: Uncertainty in Artificial Intelligence*, 2021.

[21] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, “Learning to represent programs with heterogeneous graphs,” in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. IEEE, 2022, pp. 378–389.

[22] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry,” in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[23] M. Ahmed, M. R. Samee, and R. E. Mercer, “You only need attention to traverse trees,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 316–322.

[24] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, “Multimodal attention network learning for semantic source code retrieval,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[26] C. Goller and A. Kuchler, “Learning task-dependent distributed representations by backpropagation through structure,” in *Proceedings of International Conference on Neural Networks (ICNN’96)*, vol. 1. IEEE, 1996, pp. 347–352.

[27] G. Ke, D. He, and T.-Y. Liu, “Rethinking positional encoding in language pre-training,” in *International Conference on Learning Representations*, 2021.

[28] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, “Gated graph sequence neural networks,” in *International Conference on Learning Representations*, 2016.

[29] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, “Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks,” in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[30] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[31] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[32] K. Jesse and P. T. Devanbu, “Manytypes4typescript: A comprehensive typescript dataset for sequence-based type inference,” in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 294–298.

[33] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *International Conference on Learning Representations*, 2017.

[34] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” in *International Conference on Learning Representations*, 2019.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[36] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, “Exploring software naturalness through neural language models,” *arXiv preprint arXiv:2006.12641*, 2020.

[37] M. Y. Wang, “Deep graph library: Towards efficient and scalable deep learning on graphs,” in *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[38] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, “Neural program repair by jointly learning to localize and repair,” in *International Conference on Learning Representations*, 2019.

[39] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.

[40] A. N. Sontakke, M. Patwardhan, L. Vig, R. K. Medicherla, R. Naik, and G. Shroff, “Code summarization: Do transformers really understand code?” in *Deep Learning for Code Workshop*, 2022.

[41] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning, “Parsing natural scenes and natural language with recursive neural networks,” in *ICML*, 2011.

[42] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, “Integrating tree path in transformer for code representation,” in *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[43] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo, “Ast-trans: Code summarization with efficient tree-structured attention,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 150–162.

[44] Z. Teng and Y. Zhang, “Head-lexicalized bidirectional tree lstms,” *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 163–177, 2017.

[45] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic routing between capsules,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 3859–3869.

[46] Z. Yao, F. Xu, P. Yin, H. Sun, and G. Neubig, “Learning structural edits via incremental tree transformations,” in *International Conference on Learning Representations*, 2021.

[47] S. Xue, L. Zhang, A. Li, X.-Y. Li, C. Ruan, and W. Huang, “Appdna: App behavior profiling via graph-based deep learning,” in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1475–1483.

[48] A. Li, S. Xue, X. Li, L. Zhang, and J. Qian, “Appdna: Profiling app behavior via deep-learning on function call graphs,” *IEEE Transactions on Emerging Topics in Computing*, 2020.

[49] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon, “Graphcode2vec: generic code embedding via lexical and program dependence analyses,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 524–536.

[50] Z. Li, Q. Zhou, C. Li, K. Xu, and Y. Cao, “Improving BERT with syntax-aware local attention,” in *Findings of the Association for Computational Linguistics*, 2021, pp. 645–653.

[51] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, “Graphcodebert: Pre-training code representations with data flow,” in *International Conference on Learning Representations*, 2021.

[52] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “Unixcoder: Unified cross-modal pre-training for code representation,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.