NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS

as a manuscript

Nadezhda Chirkova

NEURAL NETWORKS FOR SOURCE CODE PROCESSING

PhD Dissertation Summary
for the purpose of obtaining academic degree
Doctor of Philosophy in Computer Science

Moscow — 2022

The PhD Dissertation was prepared at National Research University Higher School of Economics.

Academic Supervisor: Dmitry P. Vetrov, Candidate of Sciences, National Research University Higher School of Economics.

1 Introduction

Topic of the thesis

Neural networks have been successfully used in a wide range of applied tasks with structured data, including image, text, or video processing. One of the key properties of neural networks that distinguishes them from other machine learning models is the possibility to adapt architecture to different kinds of data. This work focuses on adapting neural network-based models to source code processing. Neural networks have been shown to substantially improve quality in such tasks as code completion [1], bug fixing [2], translating code from one programming language to another [3], or code documentation [4], providing help for programmers and simplifying software development process. The better adaptation of neural network-based models to source code should improve the quality of solving listed tasks even further.

Source code as a data domain resembles some properties of natural text, e. g. discrete sequential nature. As a result, code is often processed with architectures borrowed from natural language processing (NLP), for example, Transformers [5] or recurrent neural networks (RNNs). However, source code features a set of specific properties, taking which into account may improve the model quality. First, source code is strictly structured, as it follows the rules of the programming language. Second, most programming languages rely on the notion of variables, which store the results of intermediate operations and allow for the reuse of these results. Third, in contrast to natural language, source code may contain user-defined identifiers of arbitrary length or complexity, as it is usually allowed by the programming language.

This thesis is devoted to utilizing the specified properties in neural network-based models for improving performance in three applied tasks. The first task is code completion in which a neural network predicts next code tokens based on already written code. Neural network-based code completion modules are plugged in the majority of modern integrated development environments (IDEs) such as Visual Studio or PyCharm. The second task is variable misuse detection and repair, later referred to as the variable misuse task. In this task, a neural network predicts the location of the bug in the program snippet (if there is any) and the location from which the variable could be copied to fix the bug. Automatically detecting and fixing bugs is a long-standing problem, solving which will substantially simplify the software development process [6]. The third task is

function naming, in which a neural network predicts the name of the function given the function body. Assistance in function naming makes code more readable and simplifies code maintenance.

Relevance

Neural networks have been widely adopted in source code processing, see elaborate review in [7]. Earlier works processed code applying neural network architectures used in NLP. A line of recent works considered adapting neural networks to the specific properties of source code described above. First, in order to utilize the syntactic tree parsed from a code snippet, Wenhan et al. [8] propose to use recursive neural networks while Li et al. [9] and Kim et al. [1] propose passing the depth-first traversal of the tree to sequential models, RNNs and Transformers correspondingly. Shiv and Quirk [10] and Kim et al. [1] further propose to adjust the Transformer's architecture to take the tree structure into account. The drawback of these works is that the proposed approaches were tested on different applied tasks and datasets, making it hard to establish the best-performing approach. Second, in order to utilize the notion of variables, the dominating approach is to apply Graph Neural Networks (GNNs) and their variants to the graph constructed by treating code tokens as vertices and drawing edges based on data- or control-flow in the program [11]. The drawbacks of this approach is that it is relatively hard to implement and that the forward pass through GNNs is relatively slow because of the time-consuming message passing procedure. Third, in order to process rare and complex identifiers, Karampatsis and Sutton [12] propose using byte-pair encoding which splits them into smaller, more frequent pieces. The drawback of this approach is that splitting makes sequences much longer, slowing down neural networks' prediction.

This thesis provides further advancement in utilizing the specifics of source code in neural networks, particularly in RNNs and Transformers, as these are two most widely used architectures for code and natural text. The first work of this thesis focuses on processing variables in RNNs and introduces the RNN-based dynamic embeddings mechanism. The dynamic embedding of each variable in the program is firstly initialized based on the variable's name and then updated each time the variable occurs in the program. In contrast to conventionally used static embeddings that are based only on variable names, the proposed dynamic embeddings capture the role of a variable in the program through

the update mechanism. The experimental part of the work shows that the proposed dynamic embeddings significantly outperform standard RNNs in code completion and variable misuse tasks, for Python and JavaScript.

The second work is devoted to utilizing the syntactic structure of code in Transformers. In recent years, several modifications have been proposed to utilize the syntactic structure of code in Transformers [10, 1, 2]. However, these modifications were tested on different tasks and datasets, and as result, it remains unclear, which approach for utilizing code structure in Transformers performs better. This work compares the modifications in a unified framework on three tasks and two programming languages and provides the recommendations for the future use of Transformers for processing syntactic structure of code, e. g. using the Sequential relative attention method. Moreover, this work explores the capabilities of Transformer to process anonymized code, in which all identifiers were replaced with placeholders `Var1`, `Var2`, `Var3` etc. In this case, no textual information on the code snippet is available and the only source of information that the model can rely on is the code syntactic structure. The work shows that Transformer can make meaningful predictions for such an anonymized code and thus is capable of capturing syntactic information. Finally, the work analyses the effect of different components of processing syntax.

The third work tackles the problem of processing rare identifiers in source code and proposes an easy-to-implement preprocessing technique based on anonymization. Particularly, all rare identifiers, e.g. those with the frequency less than a threshold, are replaced with unique placeholders `Var1`, `Var2`, `Var3` etc. The experimental part of the work shows that for Transformer architecture, the proposed technique improves the accuracy of variable misuse detection and repair by 5–6% and of code completion – by 7–10%.

The goal of this work is to improve the performance of RNNs and Transformers in applied source code processing tasks by developing and investigating methods that take into account the specifics of source code as data domain.

2 Key results and conclusions

Contributions. The main contributions of this work are threefold:

1. We proposed an RNN-based dynamic embeddings mechanism for processing variables in source code, which capture the roles of variables in a program through the update

mechanism. The model with the proposed dynamic embeddings outperforms the conventional RNN model in code completion and variable misuse tasks by 0.5-18%, depending on the task and programming language (Python or JavaScript).

2. We conducted an empirical study of the capabilities of Transformers to utilize the syntactic structure of source code, including the comparison of five syntax-based Transformer modifications on variable misuse, function naming, and code completion tasks on two programming languages (Python or JavaScript), testing the general capability of Transformers to capture syntactic information, and analysing the effect of different components of processing syntax. The results of the study underline Sequential relative attention as the most effective and efficient approach for capturing syntactic information.

3. We proposed an easy-to-implement preprocessing technique for source code, namely the anonymization of rare identifiers, which improves the quality of Transformer in variable misuse and code completion tasks by 5–10%, for Python and JavaScript.

Theoretical and practical significance. The proposed models and conducted empirical studies pave the way towards further advancement in deep learning for source code. Through the use of the proposed dynamic embeddings, the proposed anonymization of rare identifiers, and Transformer with relative attention highlighted in the empirical study of Transformers, one can substantially improve the quality of code completion, variable misuse detection and repair, function naming, or other tasks. Providing high-performing solutions for specified tasks improves programmers' experience, simplifies software development, and improves the readability of code.

Key aspects/ideas to be defended:

1. An RNN-based dynamic embeddings mechanism for processing variables in source code and capturing the roles of variables in programs;

2. An empirical study of five modifications of the Transformer architecture for capturing the syntactic structure of source code and of the general capabilities of the Transformer architecture to capture code syntax, and its main conclusion about the effectiveness and efficiency of the Sequential relative attention approach;

3. An easy-to-implement approach for processing rare identifiers in source code based on their anonymization.

Personal contribution. The first work is conducted solely by the thesis' author. In the second and third works, the author proposed the key scientific ideas, implemented methods, conducted all experiments on variable misuse and function naming tasks, and wrote text. The contribution of the second author to this research was conducting experiments on the code completion task, discussing the obtained results with the thesis' author, and help with writing and editing text.

Publications and probation of the work

First-tier publications

1. **Nadezhda Chirkova**. On the Embeddings of Variables in Recurrent Neural Networks for Source Code. In Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021 (NAACL 2021). Pages 2679-2689. CORE A conference.

2. **Nadezhda Chirkova** *and Sergey Troshin*. Empirical Study of Transformers for Source Code. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021 (ESEC/FSE 2021). Pages 703-715. CORE A* conference.

3. **Nadezhda Chirkova** *and Sergey Troshin*. A Simple Approach for Handling Out-of-Vocabulary Identifiers in Deep Learning for Source Code. In Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021 (NAACL 2021). Pages 278-288. CORE A conference.

Reports at seminars

1. Seminar of the Bayesian methods research group, Moscow, 13 November 2020. Topic: "Deep learning for source code: handling syntactic structure and indentifiers".

2. Seminar of the Faculty of Computer Science in Voronovo, 29 May 2021. Topic: "Empirical study of Transformers for source code".

3. Computer Science Seminar, UC Davis, Online, 30 July 2021. Topic: "Empirical study of Transformers for source code".

Volume and structure of the work. The thesis contains an introduction, contents of publications and a conclusion. The full volume of the thesis is 64 pages.

3 Content of the work

3.1 On the Embeddings of Variables in Recurrent Neural Networks for Source Code

One of the easiest and widely used ways to process source code with neural networks is to treat source code as a sequence of tokens and use classical NLP architectures, such as RNNs or Transformers. In order to utilize the syntactic structure of code, a widely used approach is to traverse the abstract syntax tree (AST) parsed from the code snippet in the depth-first order and pass the resulting sequence to the RNN [9] or Transformer [1], see Figure 1(c, d). However, in both of these scenarios, the variables are processed in the same way as words in natural text, i. e. using the standard embedding layer, while the concept of variables is more complex. This work focuses on tackling this problem for RNNs.

A variable is a named area of data storage. A variable's name is used to match all the occurrences of the same variable in the program. Although names are often connected to the roles of variables, such a connection is not required by programming languages. Moreover, names rarely reflect the variable's role fully. For example, the variable that stores the sum of the salaries of the company's employees may be called `sum_salaries`, simply `sum` or even broadly used `x`, and names `x` or `sum` may be used in a lot of other contexts or programs. When a standard embedding layer is used to obtain the vector representations of variables, the latter are based only on variables' names. The goal of this work is to develop an approach in which the vector representations of variables are based on both variables' names and variables' roles in the programs, determined by contexts in which variables occur.

We propose the *dynamic* embeddings mechanism for processing variables in source code, which replaces conventionally used *static* embeddings in RNNs. At the beginning of program processing, the embeddings of all variables are initialized with standard (static) embeddings, see Figure 1 (e). When the RNN processes the program token by token, each time the token is a variable, this variable's embedding is updated based on its previous state and on the current hidden state of the RNN, see Figure 1 (g). The hidden state of the RNN is used in the dynamic embeddings update procedure in order to reflect the contexts in which variables occur.

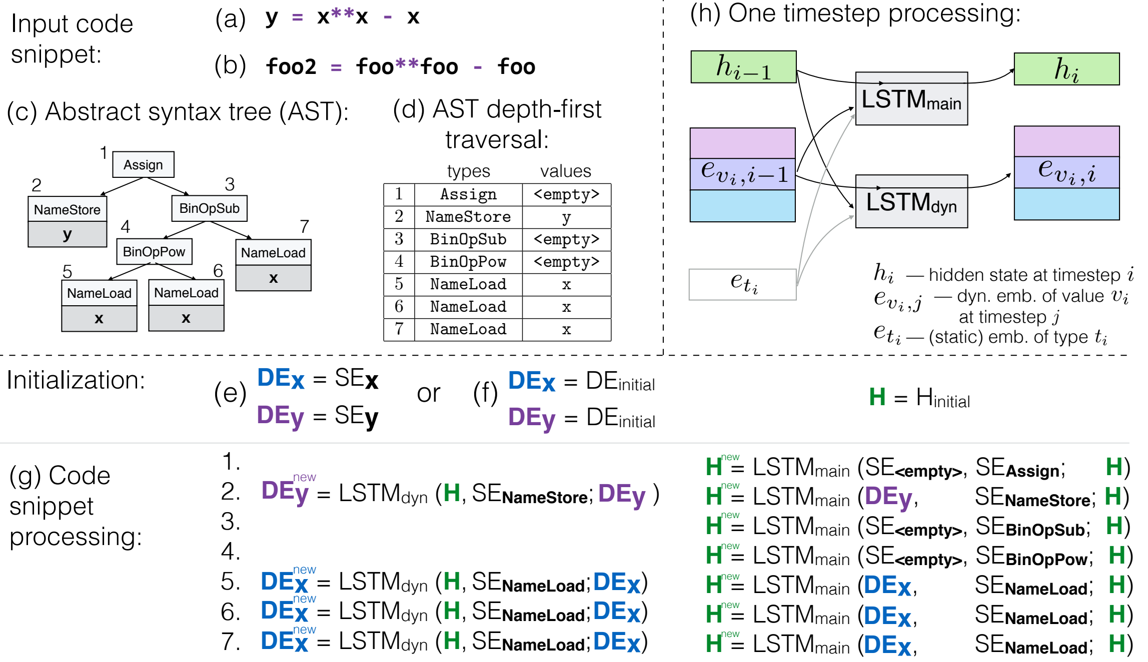

Figure 1: The overview of the proposed dynamic embeddings. (a) and (b): two variants of the input code snippet, variant (a) is used in other illustration blocks; (c) abstract syntax tree (AST); (d) AST converted to a sequence that will be passed to the RNN; (e) the static-embedding-based initialization of dynamic embeddings; (f) the constant initialization of dynamic embeddings; (h) the scheme of updating dynamic embeddings and hidden states; (g) the scheme of one timestep processing. SE: static embedding, DE: dynamic embedding.

More formally, consider an RNN applied over the depth-first traversal of the program snippet. Each node in the AST stores type representing the syntactic unit of the programming language, e. g. `Assign` or `NameLoad` in Figure 1 (c), and some nodes also store values representing user-defined variable names, function names, or constants, e. g. `x` and `y` in Figure 1 (c). By traversing the AST, we obtain a sequence $I = [(t_1, v_1), \dots, (t_L, v_L)]$. Here L denotes the length of the sequence, $t_i \in T$ denotes the type and $v_i \in V$ denotes the value, T and V are vocabularies of types and values respectively. In this work, we consider two tasks: code completion and variable misuse detection and repair. In code completion, the task is to predict the next type and value at each step $i = 1, \dots, L$; we consider only value prediction as it is a harder task. In variable misuse, we use a bidirectional RNN which outputs L representations of program tokens. The task is to point to the location of the bug and to the location which could be used to repair the bug. In case there is no bug, the task is to point at a specific no-bug location.

As a recurrent cell, we use Long Short-Term Memory Network (LSTM) [13]. The proposed model employs two LSTMs: the main LSTM which updates hidden state $h_i \in \mathbb{R}^{d_{hidden}}$, $i = 0, 1, \ldots, T$ and the dynamic LSTM which updates the dynamic embeddings of the values, as they store variables. The dynamic embedding of value v at step i is denoted $e_{v,i} \in \mathbb{R}^{d_{dyn}}$. For example, $e_{v_{i-1},i}$ denotes the embedding of a value located at position $i-1$, at step i, and $e_{v_{i-1},i-1}$ denotes its state at the previous step. The main LSTM updates the hidden state h_i based on the previous hidden state h_{i-1} and the embeddings of type t_i and value v_i of the current AST node:

$$h_i = \mathrm{LSTM}_{\mathrm{main}}(e_{v_i,i-1}, e_{t_i}; h_{i-1}). \quad (1)$$

The dynamic LSTM updates the dynamic embedding of the current value based on its previous state, the type of the current AST node and current hidden state:

$$e_{v_i,i} = \mathrm{LSTM}_{\mathrm{dyn}}(h_{i-1}, e_{t_i}; e_{v_i,i-1}) \quad (2)$$

$$e_{v,i} = e_{v,i-1}, \quad v \neq v_i \quad (3)$$

Equation (3) means that at step i, only the dynamic embedding of value v_i is updated, while the dynamic embeddings of all other values stay unchanged. The whole process is illustrated in Figure 1 (h). In practice, there are several special values, `<EMPTY>`, `<UNK>` and `<EOF>`, for which we use static embeddings. Dynamic embeddings can be initialized with static embeddings, in order to utilize value names, or with a constant embedding, same for all values.

We test the proposed approach in code completion and variable misuse tasks, on Python150k [14] and JavaScript150k [15] datasets. Both datasets contain 150K code files and their parsed ASTs. In code completion, we measure quality using accuracy (the percent of correctly predicted values), while in variable misuse we use joint localization and repair accuracy (the percent of correctly located and fixed bugs). We compare the proposed dynamic embeddings with the standard model incorporating static embeddings. In all our models, node type embeddings have 300 units, node value embeddings have 1200 units (for static embeddings), and the one-layer LSTM's hidden state has 1500 units, following Li et al. [9]. The proposed dynamic embeddings of values have 500 units in all experiments, to show that their superioriry is achieved with a much smaller dimension than the static embeddings dimension. In the code completion task, we equip all models

Model	Code completion		Variable misuse	
	PY	**JS**	**PY**	**JS**
Full data setting				
Static embeddings	64.69	65.05	54.78	35.06
Dynamic embeddings	**68.61**	**65.67**	**68.59**	**53.74**
Anonymized setting				
Static embeddings	60.28	57.67	25.17	13.16
Dynamic embeddings	**66.90**	**62.85**	**63.64**	**53.53**

Table 1: Performance of the proposed dynamic embeddings compared to the baseline static embeddings for two tasks on Python150k (Py) and JavaScript150k (JS) datasets. Accuracy (%) of LSTM with pointer (code completion), joint accuracy (%) of BiLSTM (variable misuse). All standard deviations are less than 0.05% for code completion and 0.1% for the variable misuse task.

with attention and pointer mechanisms, and in the variable misuse task, all models use bidirectional LSTMs (BiLSTMs).

The main results for both tasks are presented in Table 1 (full data setting) and show that the dynamic embeddings model outperforms the static embedding model in both tasks and on both datasets. We note that the number of parameters in the dynamic LSTM, 2.6M, is two orders smaller than those in the embedding layer, 134M, and also much smaller than the number of parameters in the main LSTM module, 13.8M.

Table 1 also presents the results for the anonymized setting, in which all values are replaced with unique placeholders `Var1`, `Var2`, `Var3` etc. As value names are not used in this setting, dynamic embeddings are initialized with a constant embedding. We observe that the proposed dynamic embeddings again substantially outperform static embeddings. Moreover, in this scenario, the predictions of the proposed model do not depend on the particular value naming, i. e. predictions do not change if all values are renamed. At the same time, the static embeddings model does not satisfy this conceptually reasonable property.

The proposed dynamic embeddings resemble the general idea proposed in [16] for processing rare named entities in natural text. In contrast to their work, we apply dynamic embeddings to the whole vocabulary of variable names, and incorporate dynamic embeddings into the model that utilizes the syntactic structure of code, providing a more meaningful context for updating dynamic embeddings.

3.2 Empirical Study of Transformers for Source Code

Initially proposed for the problem of machine translation, Transformers [5] became the state-of-the-art model in a range of other NLP tasks as well as in the tasks from other data domains, e. g. source code [1, 2, 4]. In order to utilize the syntactic structure of source code represented, for example, via the AST, a line of recent works propose Transformer modifications that incorporate AST processing into Transformer. Example modifications include tree-based positional encoding [10] or tree-based relative attention mechanism [1]. However, due to the lack of established benchmarks in source code processing, different modifications were tested on different applied tasks and datasets, complicating the establishment of the best-performing approach. The goal of this work is to compare different approaches for processing AST structure in Transformer in a unified framework on several applied tasks and to test the general capabilities of Transformer to capture code syntactic information.

The core of the Transformer architecture is a self-attention mechanism which updates the vector representations of a set of elements: $[x_1, \dots x_L] \to [z_1, \dots, z_L]$, $x_i, z_i \in \mathbb{R}^{d_{model}}$, $i = 1, \dots, L$. First, self-attention computes key, query, and value vectors from each input vector: $x_j^k = x_j W^K$, $x_j^q = x_j W^Q$ and $x_j^v = x_j W^V$. Then each output z_i is computed as a weighted combination of inputs:

$$z_i = \sum_j \tilde{\alpha}_{ij} x_j^v, \quad \tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}, \quad a_{ij} = \frac{x_i^q {x_j^k}^T}{\sqrt{d_z}} \tag{4}$$

Here we omit other details such as multi-head attention or layer normalization for brevity. Self-attention itself is invariant to the order or structure of the input elements, hence additional mechanisms are usually used to take the structure or order into account. This work considers five Transformer modification for processing the syntactic structure of source code defined by AST:

1. *Sequential positional embeddings.* Classic Transformer architecture [5] incorporates positional encoding, i. e. vector representations $p_i \in \mathbb{R}^{d_{model}}$ of positions $i = 1, 2, 3, \dots$ which are summed up with the embeddings $x_i \in \mathbb{R}^{d_{model}}$ of tokens:

   $$\hat{x}_i = x_i + p_i \tag{5}$$

   Vector representations p_i could be learnable or computed based on sine and cosine functions. We apply the described approach over the depth-first traversal of the AST.

2. *Sequential relative attention.* Shaw et al. [17] propose to incorporate information about sequential structure directly into the self-attention mechanism:

$$z_i = \sum_j \tilde{\alpha}_{ij}(x^v_j + e^v_{i-j}),\ \tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})},\ a_{ij} = \frac{x^q_i(x^k_j + e^k_{i-j})^T}{\sqrt{d_z}}, \tag{6}$$

where $e^v_{i-j}, e^k_{i-j} \in \mathbb{R}^{d_{model}}$ are learned embeddings for each relative position $i - j$, e. g. one token is located two tokens to the left from another token. We apply the described approach over the depth-first traversal of the AST.

3. *Tree positional encoding.* Shiv and Quirk [10] propose AST-based positional encoding. The position of each node in the tree is defined as a path from the root to this node and is encoded using stacked one-hot representations. The obtained positional encoding $[p_1, \ldots, p_L]$ is substituted into (5).

4. *Tree sequential attention.* Kim et al. [1] propose to modify the relative attention mechanism in order to utilize relations in AST:

$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij} \cdot r_{ij})}{\sum_j \exp(a_{ij} \cdot r_{ij})} \tag{7}$$

where relation embeddings $r_{ij} \in \mathbb{R}$ encode paths in AST, e. g. "Up Up Down Down Down". Scalar relations are used due to memory constraints, other formulas of self attention are same as in (4).

5. *GGNN Sandwich.* Hellendoorn et al. [2] propose to combine Transformer and Gated Graph Neural Networks (GGNN) models by alternating Transformer blocks and GGNN layers. We utilize this model for trees, though it is capable of processing arbitrary graphs.

As in Section 3.1, we represent each node in an AST with a type and an optional value, and the nodes without values store auxiliary `EMPTY` values.

We compare the described Transformer modifications for processing ASTs on three tasks, namely variable misuse localization and repair (VM), function naming (FN), and code completion (CC). We chose the tasks so that both encoder-based and decoder-based models are covered and that various aspects of code understanding are tested. We use the following metrics widely used in relevant works: joint localization and repair accuracy in VM, F-measure in FN, and mean reciprocal rank (MRR) in CC.

We consider Python150k and JavaScript150k datasets. While working with data we noted that the train / test data splits provided by the dataset authors do not follow

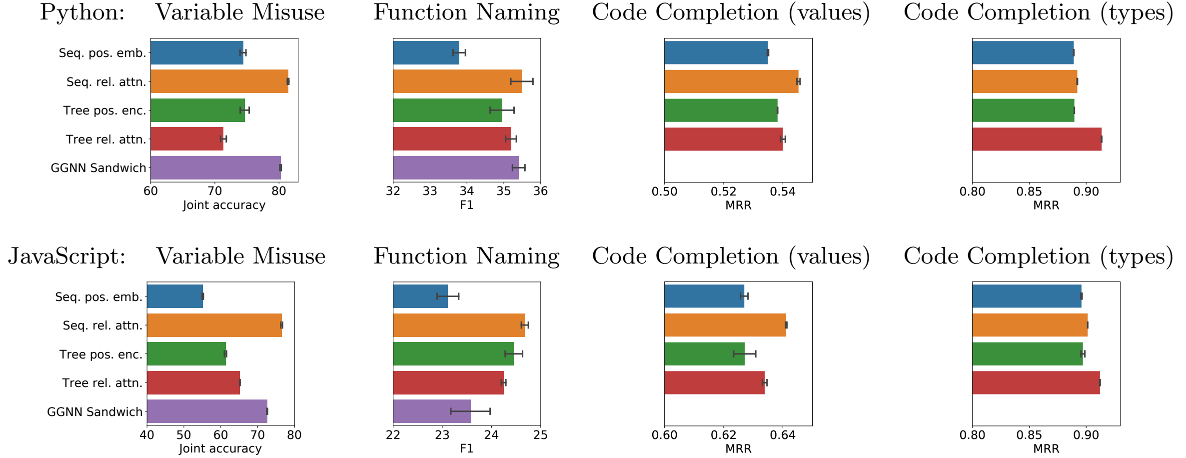

Figure 2: A comparison of different Transformer modifications for processing AST structure in Transformer

Model	Train time (h/epoch)	Preprocess time (ms/func.)	Add. train data (GB)
Seq. pos. emb.	2.3	0	0
Seq. rel. att.	2.7	0	0
Tree pos. enc.	2.5	0.4	0.3
Tree rel. attn.	3.9	16.7	18
GGNN Sandwich	7.2	0.3	0.35

Table 2: Time- and storage-consumption of different structure-capturing Transformer modifications for the variable misuse task on the Python dataset.

conventional practice of splitting by repository and conducting deduplication, which may bias the results. Thus we release new splits following the recommendations of Allamanis [18] and Alon et al. [19].

We re-implement the described Transformer modifications for a fixed base model with 6 layers and the model size $d_{model} = 512$, to avoid the influence of differences in baseline implementations. We tune hyperparameters of all modifications using the validation set, while training hyperparameters are chosen for the base model and fixed across modifications. All models have approximately the same number of learnable parameters.

Figure 2 reports the comparison of five Transformer modifications for processing ASTs for three tasks and two datasets. In VM, we observe that GGNN-Sandwich and Sequential Relative Attention perform best, and the former model was specifically developed for this task. In FN, almost all models (except sequential positional encoding) perform similarly.

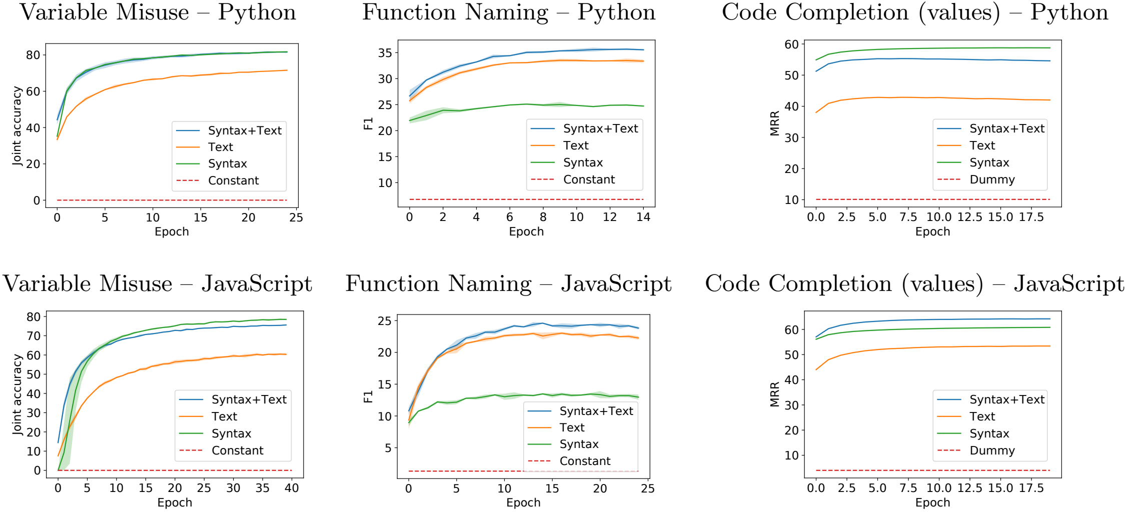

Figure 3: Comparison of syntax-based Transformer models with text-only and constant baselines.

In CC, we report the quality of value and type prediction separately. In value prediction, Sequential relative attention outperforms other models. In type prediction, Tree relative attention, the model which was developed specifically for the CC task, outperforms other models. To sum up, we observe that Sequential relative attention achieves best results in almost all tasks. Interestingly, this mechanism was not considered as a baseline in the works on Tree relative attention [1] and GGNN-Sandwich [2] which develop approaches inspired by this technique.

Table 2 reports efficiency metrics for five considered Transformer modifications and shows that Sequential relative attention is not only the most effective but also substantially more time- and memory-efficient than more complicated modifications. In the paper, we also show that combining Sequential Relative Attention with GGNN-Sandwich or Tree relative attention may lead to further performance improvement.

We also conduct an experiment testing the general capability of Transformers to capture information from ASTs. We consider two settings. In the first setting, we compare configurations *Syntax+Text* and *Text*. The first one corresponds to the conventional model and the latter approach corresponds to passing a sequence of AST values to the Transformer (textual information only), with types and structural information being removed. If the former model outperforms the latter one, then performance improvement should come from processing the AST. In the second setting, we compare configurations *Syntax* and *Constant*. The *Syntax* configuration corresponds to removing textual information (values) from an AST by anonymizing values, i. e. by replacing values with unique

		Full data			Anonymized data		
		VM	FN	CC	VM	FN	CC
Python	Full AST	**81.59%**	**35.73%**	**54.59%**	**81.71%**	**25.26%**	**58.76%**
	AST w/o struct.	26.81%	34.80%	53.1%	12.41%	23.29%	57.75%
	AST w/o types	71.55%	33.60%	42.01%	58.55 %	12.50%	41.26%
	AST w/o values	32.44%	25.25%	N/A	32.44%	**25.25%**	N/A
JavaScript	Full AST	**75.60%**	**24.62%**	**64.2**	**78.47%**	**13.66%**	**60.82%**
	AST w/o struct.	17.25%	23.40%	61.53%	5.37%	11.25%	58.59%
	AST w/o types	60.33%	23.09%	53.4%	43.53%	8.10%	42.91%
	AST w/o values	42.56%	13.64%	N/A	42.56%	**13.64%**	N/A

Table 3: Ablation study of processing different AST components in Transformer. Bold emphasises best models and ablations that do not hurt performance. Standard deviations: VM – 0.5%, FN – 0.4%, CC – 0.1%. AST w/o struct.: Transformer treats input as a bag without structure; AST w/o types: only values or anonymized values are passed to Transformer; AST w/o values: only types are passed to Transformer. N/A – not applicable. CC: value prediction.

placeholders `Var1`, `Var2`, `Var3` and so on. With such code representation, the only way Transformer can make meaningful predictions is to capture information from the AST. We compare the described *Syntax* configuration with a constant baseline, e. g. predicting the most frequent function name, the most frequent next value in CC, or predicting the absence of bugs in VM. All models use Sequential relative attention. Figure 3 reports the results for three tasks and two datasets and shows that the *Syntax+Text* model always outperforms the *Text* model, and the *Syntax* model always outperforms the constant baseline. Based on these results, we conclude that Transformers are indeed capable of capturing syntactic information from ASTs.

Finally, we conduct an ablation study of what AST components influence Transformers predictions in three tasks. We ablate the following components, one component at a time: types (passing a sequence of values to the Transformer, identical to the *Text* model described above), structure (passing an *unordered bag* of (type, value) pairs to the Transformer), and values (passing a sequence of types to the Transformer). Ablation study is conducted for the *Syntax+Text* and *Syntax* models described in the previous experiment. Table 3 reports the results. We find that in VM and CC, ablating any of the considered AST components hurts performance, for both considered models. In FN, performance of the *Syntax+Text* model is slightly affected by ablating types or structure and substan-

tially affected by ablating values (textual information). At the same time, performance of the *Syntax* model is slightly affected by ablating structure, substantially – by ablating types and, importantly, not affected by ablating anonymized values. The latter result means that in FN Transformer sees the anonymized code as a sequence of types, and such a representation loses much information about the algorithm implemented by the code snippet. We conclude that in contrast to VM and CC tasks, in FN Transformer utilizes information from AST only partially.

3.3 A Simple Approach for Handling Out-of-Vocabulary Identifiers in Deep Learning for Source Code

In contrast to natural text, source code data comprises much larger vocabularies, because developers are allowed to use identifiers (variable and function names) of arbitrary complexity, e. g. `students_with_high_marks` or `foldforfiles`. Since such ad-hoc identifiers only occur several times in the dataset, their learnable embeddings do not step far from the random initialization and thus are meaningless. As a result, the common practice for processing rare identifiers in source code is to collect a vocabulary of frequent identifiers and to replace all other identifiers with an `UNK` token [1, 9]. In this work, we propose a simple alternative preprocessing technique based on anonymization, which results in substantial performance improvement in variable misuse and code completion tasks, and could potentially be used in other tasks as well.

A widely used approach for processing rare tokens in NLP is to split them into subtokens, e. g. by using byte-pair encoding (BPE). The first drawback of this approach is sequences elongation which substantially slows down the prediction. The second drawback is that existing AST-processing techniques imply that each AST node stores a single value, and combining these techniques with BPE requires additional investigation.

We propose a simple anonymization-based technique to tackle out-of-vocabulary identifiers in source code. The approach is inspired by the property of variables: renaming a variable in the code snippet does not change the algorithm which is implemented by the snippet. Based on the results reported in Section 3.2, we use Transformer with sequential relative attention applied over the AST depth-first traversal, as a base model. Our approach selects a vocabulary of top-K most frequent AST values, and all out-of-vocabulary values are anonymized, i. e. replaced with placeholders `Var1`, `Var2`, `Var3` etc. For ex-

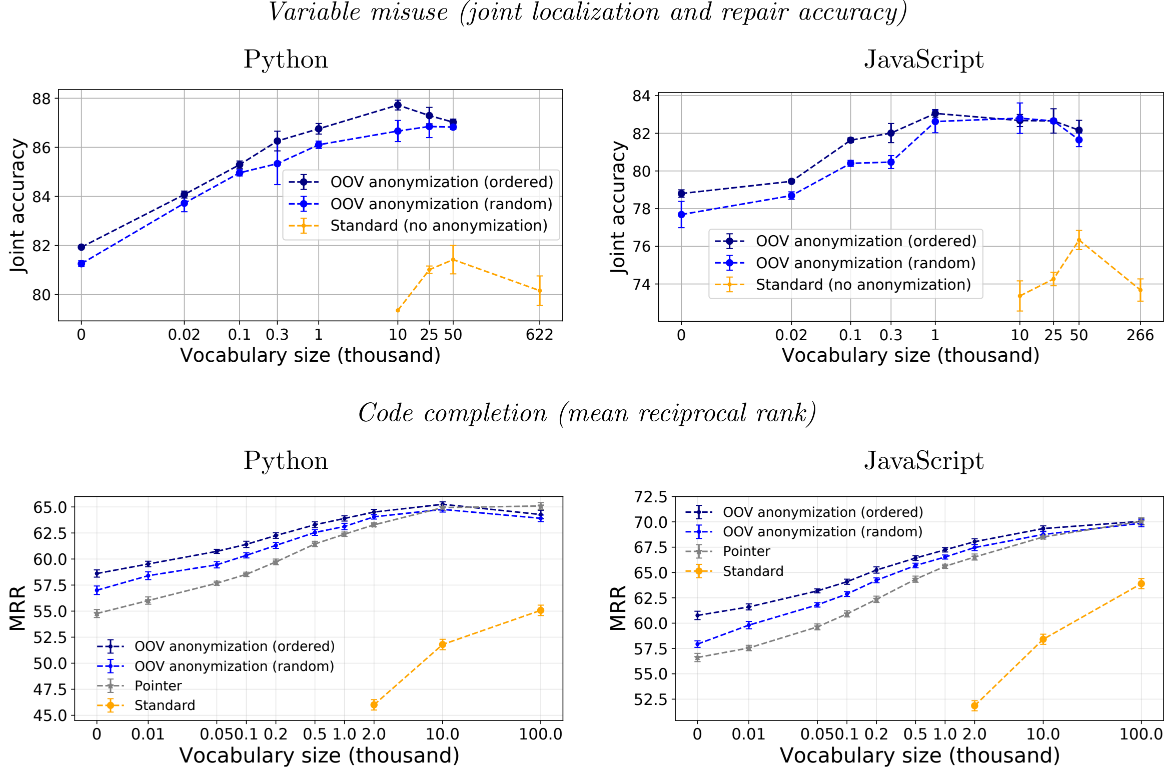

Figure 4: The performance of the proposed anonymization of rare identifiers, compared to the standard model in which rare identifiers are replaced with an UNK token. Results for Transformer in variable misuse and code completion tasks. Mean value ± standard deviation (over 3 runs) is reported. In the code completion task, the pointer mechanism is considered as an additional baseline.

ample, if we have a vocabulary which includes values `print` and `fnames`, then the code snippet

```
for incldir in fnames[lib_folder]:
    print(incldir)
```

with out-of-vocabulary values `incldir` and `lib_folder` will be transformed into

```
for Var1 in fnames[Var2]:
    print(Var1)
```

All occurrences of the same value in a code snippet are replaced with one placeholder, but a placeholder may replace different values in different code snippets. We consider two strategies for choosing placeholders: a random anonymization and an ordered anonymization. In the random strategy, the placeholder for each out-of-vocabulary value is chosen randomly from `Var1`...`Var500`. In the ordered strategy, the first out-of-vocabulary value

in the code snippet is replaced with `Var1`, the second value – with `Var2`, the third one – with `Var3` and so on. In the experiments, the ordered anonymization performs slightly better than the random anonymization.

We test the proposed anonymization-based approach on the variable misuse and code completion tasks. The experimental setup is the same as in Section 3.2, including metrics and implementation. Figure 4 reports the results. In the variable misuse task, the proposed approach outperforms the standard approach of replacing rare identifiers with the `UNK` token, by 5–6%. In the code completion task, the proposed approach again substantially outperforms the standard approach (by 6–10%). In this task, we also compare to the pointer mechanism which can copy values from the previously seen context and thus also tackles the out-of-vocabulary problem. We observe that the proposed anonymization-based approach outperforms the pointer mechanism baseline for the majority of vocabulary sizes and performs worse only in one case. However, the proposed approach is much easier to implement than the pointer mechanism.

4 Conclusion

In the final section, we summarize the main contributions of the work. The results of the work provide a set of practical recommendations regarding the use of RNNs and Transformers for source code processing.

1. We proposed RNN-based dynamic embeddings for computing the embeddings of variables in source code during forward pass through the network. After initializing embeddings with standard (static) embeddings at the beginning of the program processing, the dynamic embedding of each variable is updated each time the variable occurs in the program. Compared to the standard model with static embeddings, the proposed model substantially improves performance in code completion and variable misuse detection and repair, on two datasets, in both full-data and anonymized settings.

2. We studied performance of five Transformer modifications which process the AST parsed from the program and found that Sequential relative attention is the most effective and efficient method, not considered as a baseline in previous works. This approach performs best in three tasks out of four, on two datasets, and could be combined with other methods, Tree relative attention and GGNN-Sandwich, for further performance improvement. We also found that in code completion and variable misuse detection and repair, Transformer's performance drops substantially if any component of the AST is omitted in the network input, while in function naming, Transformer mostly relies on textual input and uses AST mostly for node type information.

3. We proposed a simple anonymization-based preprocessing approach for handling out-of-vocabulary identifiers. The proposed approach replaces rare identifiers with unique placeholders `Var1`, `Var2`, `Var3` ... which occur in many code snippets and thus have relatively high frequencies. The proposed approach was tested for Transformers and shown to substantially improve performance in variable misuse detection and repair and code completion, compared to the commonly used approach of replacing rare identifiers with the `UNK` token. In addition, the proposed approach in almost all cases outperforms the pointer mechanism baseline in code completion, being much easier to implement.

References

[1] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162, 2021. doi: 10.1109/ICSE43902.2021.00026.

[2] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations, ICLR 2020*, 2020. URL `https://openreview.net/forum?id=B1lnbRNtwr`.

[3] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *arXiv preprint arXiv:2006.03511*, 2020.

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf`.

[6] S. Amirhossein Mousavi, Donya Azizi Babani, and Francesco Flammini. Obstacles in fully automatic program repair: A survey, 2020.

[7] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Comput. Surv.*, 53(3), jun 2020. ISSN 0360-0300. doi: 10.1145/3383458. URL `https://doi.org/10.1145/3383458`.

[8] Wang Wenhan, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. Modular tree network for source code representation learning. *ACM Transactions on Software Engineering and Methodology*, 29:1–23, 10 2020. doi: 10.1145/3409331.

[9] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4159–4165. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/578. URL https://doi.org/10.24963/ijcai.2018/578.

[10] Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 12081–12091. Curran Associates, Inc., 2019.

[11] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

[12] Rafael-Michael Karampatsis and Charles Sutton. Maybe deep neural networks are the best choice for modeling source code, 03 2019.

[13] Sepp Hochreiter and JÃŒrgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[14] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. Probabilistic model for code with decision trees. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[15] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 761–774, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837671. URL https://doi.org/10.1145/2837614.2837671.

[16] Sosuke Kobayashi, Naoaki Okazaki, and Kentaro Inui. A neural language model for dynamically representing the meanings of unknown words and entities in a discourse. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 473–483, Taipei, Taiwan, November 2017.

Asian Federation of Natural Language Processing. URL https://www.aclweb.org/anthology/I17-1048.

[17] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2074. URL https://www.aclweb.org/anthology/N18-2074.

[18] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019.

[19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=H1gKYo09tX.