***Abstract*—Although software development is mostly a creative process, there are many scrutiny tasks. As in other industries, there is a trend for automation of routine work. In many cases, machine learning and neural networks have become a useful assistant in that matter. Programming is not an exception: GitHub has stated that Copilot is already used to write up to 30% of code in the company. Copilot is based on Codex, a Transformer model trained on code as a sequence. However, a sequence is not a perfect representation for programming languages. In this work, we claim and demonstrate that by combining the advantages of Transformers and graph representations of code, it is possible to achieve excellent results even with comparably small models.**

***Keywords*—Neural networks, Transformers, graphs, abstract syntax tree**

GraphTyper: Neural Types Inference from Code Represented as Graph

German Arutyunov* gaarutyunov@edu.hse.ru
Sergey Avdoshin* savdoshin@hse.ru
*HSE University, 20, Myasnitskaya st., Moscow, Russia

I. Introduction

Application of Transformers yet again has managed to break the deadlock: this time in the task of code generation [1, 2, 3, 4]. Nevertheless, the versatile Transformer architecture has displayed good results on several benchmarks, in the recent work [5] it was shown that increasing the size of the model doesn't result in a better performance. Moreover, it is evident that context matters a lot to produce a working code. However, it is not practical to relentlessly increase the length of context sequence in a Transformer. Therefore, a different approach is needed to boost the efficiency in machine programming tasks [6].

First of all, an expressive code representation has to be selected. Several ways, including token-based, structured and graph-based approaches, have been reviewed [7]. For instance, graph representation using abstract syntax tree (AST), data-flow graph (DFG) and control-flow graph (CFG) yield good results in such tasks as variable misuse detection and correction [8]. Such graph representation can capture an extensive amount of information about the program's code.

Secondly, a versatile model architecture that supports learning on graphs must be used. Multiple models such as RNN [9], LSTM [10] and CNN [11] with flattened graphs have been used. However, graph-aware model architecture is more suitable for the graph representation of code. For this reason, Graph Neural Networks (GNN) are a more reasonable choice of architecture, namely message-passing neural networks [8].

Nonetheless, in this work we aim to make the most of both worlds: the advantages of Transformer architecture and graph representation of code. For instance, we will use Transformer architecture optimizations [12] and graph code representation created from AST and DFG. To make this possible, we will use Pure Transformers [13] instead of models that have some architectural alterations to support graph structure [14, 15, 16].

II. Problem Statement

In this work, we test the ability of Pure Transformers to add types to Python source code based on its graph structure. This task was selected as a starting point for future research due to its practical relevance.

Firstly, dynamically typed languages, such as Python and JavaScript, have gained quite some traction during the last years [17]. However, it doesn't mean they're easier [18, 19, 20] or less error-prone than statically typed languages [21]. Moreover, lack of type hints in libraries might lead to expensive errors in fields such as Data Science [22].

There are some tools outside the neural networks domain that perform static type checking and inferencing type annotations [23, 24]. Nonetheless, these utilities do not work without type hints in the source code of the dependencies, which is pretty common. To alleviate this, there are proposals about Domain-Specific Languages (DSL) for Data Science [22]. However, it wouldn't work on existing code base and massive adoption is not very likely.

On the other hand, absence of type hints is not a restriction for neural networks (see. Section V-B). In addition, they don't only find erroneous types in existing codebase [25] but can also be used during development to annotate code on the fly [26].

Most importantly, inferring types requires a model to learn a lot about the source code. Therefore, developing a model with versatile architecture to infer types allows it to be later applied for other tasks.

A. Metrics

To test the model, we use two metrics from the Typilus paper [25]:

Exact Match: Predicted and ground truth types match exactly.
Match up to Parametric Type: Exact match when ignoring all type parameters.

III. Previous Work

A. Graph Representation of Code

AST and DFG have already been used with Transformers in the code generation and summarization tasks [27, 28, 29]. In addition, some joint graph structure representations that include different code graphs have been developed, namely code property graph (CPG) [30], that incorporates AST, CFG and PDG (program dependency graph). This graph representation has already been used for vulnerability detection [30] and similarity detection [31].

B. Graph Transformers

Graph Transformers is a novel architecture that has been developing in the past few years. They have been applied for several tasks, mostly in the field of molecule generation, node classification and node feature regression [13, 14, 15, 16].

TABLE I
QUANTITATIVE EVALUATION OF MODELS MEASURING THEIR ABILITY TO PREDICT GROUND TRUTH TYPE ANNOTATIONS.

Top-n	Model	Exact Match	Up to Parametric Type
Top-1	GraphTyper	34.71	36.43
	TypeBERT	45.40	48.10
	TypeWriter	56.10	58.30
	Typilus	66.10	74.20
	Type4Py	75.80	80.60
	TypeGen	**79.20**	**87.30**
Top-3	GraphTyper	45.47	55.02
	TypeBERT	51.40	53.50
	TypeWriter	63.70	67.30
	Typilus	71.60	79.80
	Type4Py	78.10	83.80
	TypeGen	**85.60**	**91.00**
Top-5	GraphTyper	50.70	64.58
	TypeBERT	54.10	56.50
	TypeWriter	65.90	70.40
	Typilus	72.70	80.90
	Type4Py	78.70	84.70
	TypeGen	**87.00**	**91.70**

Apart from models with alterations to Transformer base architecture [16, 14] researchers have recently developed simpler models [13] that are compatible with many popular techniques developed for standard Transformers [12].

C. Type Inference with Neural Networks

The task of type inference has been also extensively covered in recent research. Many different architectures have been used for this task: GNNs [25], RNNs [32, 26] and Transformers [33, 34] among others. Moreover, graph representation of code has been used for the task of type inference in dynamically typed programming languages such as Python [25] and Javascript [35].

However, the power of Graph Transformers and Graph Representation of code hasn't been combined yet to solve the task of type inference in source code. This is the gap our model aims to fill. The results of our model compared to previous work [26, 25, 32, 33, 34] are displayed in Table I.

IV. PROPOSED SOLUTION

A. Dataset

To train and test the model we gathered 600 Python repositories from GitHub containing type annotations from Typilus [25]. We clone these repositories and use pytype [24] for static analysis, augmenting the corpus with inferred type annotations. The top 175 most downloaded libraries are added to the Python environment for type inference. Through deduplication, we remove over 133 thousand code duplicates to prevent bias.

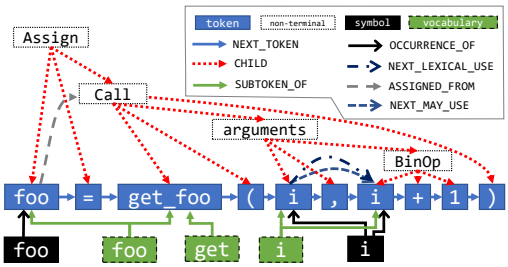

Fig. 1. Sample graph for foo=get_foo(i, i+1) showing different node and edge types implemented by Allamanis et al. [25].

The resulting dataset comprises 118,440 files with 5,997,459 symbols, of which 252,470 have non-Any non-None type annotations. The annotations exhibit diversity with a heavy-tailed distribution, where the top 10 types cover half of the dataset, primarily including str, bool, and int. Only 158 types have over 100 annotations, while the majority of types are used fewer than 100 times each, forming 32% of the dataset. This distribution underscores the importance of accurately predicting annotations, especially for less common types. The long-tail of types consists of user-defined and generic types with various type arguments.

The source files are processed to generate graphs that contain AST, DFG, as well as lexical and syntactical information. An example of such a graph is shown on Figure 1. In addition to extracting graphs from source code AST, we split them by setting a maximum node and edges number in one graph. For this, we prune the graphs around nodes that have annotations that are later used as targets during training and testing. Finally, we split the data into train-validation-test sets with proportions of 70-10-20, respectively.

B. Model Architecture

We base our model architecture on TokenGT [13]. The main advantage of this model is that standard Transformer architecture is not altered to support graph data. It allows us to use some advantages developed specifically for Transformers. For instance, Performer [12] is used to speed up training by using linear time as space complexity.

The main idea of the authors is that combining appropriate token-wise embeddings and self-attention over the node and edge tokens is expressive enough to accurately encode graph structure to make graph and node-wise predictions. The embeddings in the model are composed of orthonormal node identifiers, namely Laplacian eigenvectors obtained from eigendecomposition of graph Laplacian matrix. In addition, type identifiers are used to encode types of tokens (nodes or edges).

In our model, we use node and edge types extracted from code as token features. Node ground truth annotations are added to the features and randomly masked during training. The overall architecture of the model is displayed at Figure 2.

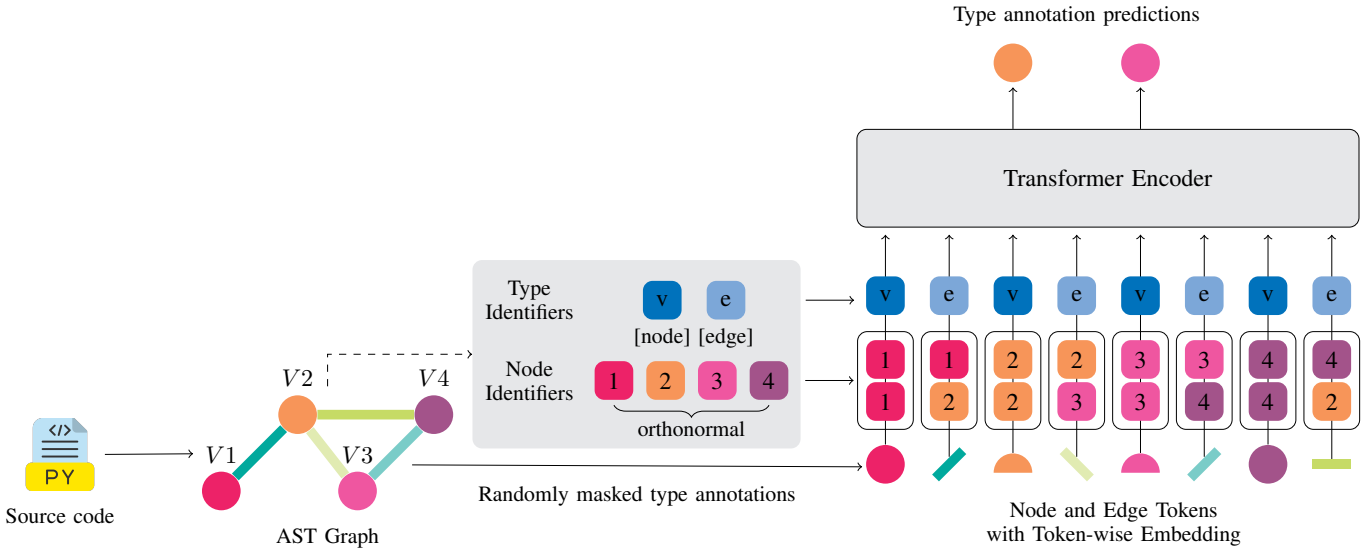

Fig. 2. GraphTyper Architecture. The source code is first transformed into AST graph, then type annotations are randomly masked. The graph is enriched by token type identifiers (node or edge) and orthonormal node identifiers obtained from eigendecomposition of Laplacian matrix. The resulting graph is fed through a Transformer Encoder to obtain type annotations for masked nodes.

1) Masked Transformer Encoder Model: Predicting type annotations in graph domain is a node classification task. However, since we are using a Pure Transformer with graphs represented as a sequence of tokens, the task can be reduced to token classification. In the Natural Language Processing (NLP) domain, this is a ubiquitous task, also known as Named Entity Recognition (NER).

Encoder-only architecture has been widely used for the NER task, namely BERT is one of the most popular models [36, 37]. We adapt similar architecture by randomly masking type annotations. We then apply an MLP layer to the output of TokenGT [13] to get logits of type annotations.

Masked model architecture is very versatile, and the pre-trained model can be later easily fine-tuned for other tasks, similar to the approaches from the NLP-domain [36]. For example, error [38] and vulnerability [39] data can be added to the code graph to detect and fix them [40, 41, 42, 43, 44].

V. Experiments and Ablation Results

To select the final model architecture, we test different models. For our experiments and ablation analysis, we train and test the models using one sample repository. We also limit the number of types in the vocabulary to one hundred to speed up training and use less resources. To test the models, we calculate Top-n predictions similar to the previous work [26]. Table II depicts the results of the experiments and ablation.

A. Validating the necessity of node and type identifiers that encode graph structure

First of all, we remove the node and type identifiers introduced by Kim et. Al [13] our ablation analysis demonstrates that indeed, the graph structure embeddings play a key role in model quality. By removing them from the model, we are left with a simple Transformer that makes predictions only based on AST nodes and edges types without any information about graph structure. Such a model outputs the worst results among all the experiments.

B. Using the model without node type annotations

In addition, we try to remove the type annotations from the model completely. This alteration turns our training into a masked NER task. Surprisingly, our model performs well in such conditions. This means that the selected graph representation of code contains a lot of necessary information to infer types.

C. Increasing the number of parameters

As we can see, increasing the number of parameters also increases the predictive power of the model. However, increasing the parameters indefinitely is not very practical and requires a lot of computational resources [6]. Moreover, keeping the low number of parameters allows us to use longer context length (more node and edges in graph) during inference with same resource capabilities. Therefore, we don't change the parameter number of the final model, so it remains compact.

D. Testing different context length

As for the context length, i.e., maximum number of nodes in graph (512 vs. 1024), our findings are aligned with the conclusions from previous work [6]: longer context increases the performance of the model. However, the AST representation of source code is very bloated and even having a lot of nodes in the graph might not capture enough useful information to make quality predictions. In addition, increasing the context length drastically slows down the training process. Thus, in future research, we will be working on finding a better and more compact graph representation of code.

TABLE II
EXPIREMENT RESULTS OF TOP-N PREDICTIONS FOR DIFFERENT MODEL VARIANTS.

Top-n	Model	Exact	Up to Parametric Type
Top-1	Plane Transformer	10.15	19.46
	+ Node & Type Identifiers	30.88	36.55
	+ Type Annotations	33.36	**42.28**
	+ Decoder (Autoencoder)	15.90	16.65
	or Longer Context	**38.49**	39.80
	or More Parameters	29.39	31.82
Top-3	Plane Transformer	15.06	29.40
	+ Node & Type Identifiers	40.33	50.37
	+ Type Annotations	41.71	52.90
	+ Decoder (Autoencoder)	28.26	32.81
	or Longer Context	**53.14**	**57.41**
	or More Parameters	44.85	49.72
Top-5	Plane Transformer	16.81	37.91
	+ Node & Type Identifiers	42.82	56.01
	+ Type Annotations	43.62	57.00
	+ Decoder (Autoencoder)	44.17	56.33
	or Longer Context	**58.80**	**67.38**
	or More Parameters	49.74	56.14

E. Testing different Transformer architectures

Recently, Masked Graph Autoencoders have been applied for the tasks of link prediction and node classification [45], as well as feature reconstruction [46, 47]. To validate the robustness of the Encoder-only Model, we also implement a Masked Autoencoder Model. For this, we adapt the approach of Hou et. al [47] for our model. We introduce a learnable mask token and a decoder based on the encoder layers. We reconstruct the type annotations by re-masking the target nodes before feeding them into the decoder. However, we do not observe as good results as with a simple Encoder-only model.

VI. KNOWN LIMITATIONS

A. Size of Type Vocabulary

Since we define our task as node (token) classification, we feed our transformer output into a classifier linear head. Therefore, our type vocabulary is limited. Because of the computational resources constraints, we limit it to one thousand types.

This issue is addressable by formulating the task as Deep Similarity Learning Problem [48, 49]. In this way, the model will output vector representations of types that can be grouped into cluster of similar types. After that, an algorithm such as KNN [50] is used to transform vector representation into a probability of each type [25, 26]. Illustration for such an approach is depicted on Figure 3. The approach follows the methodology described in Typilus paper [25].

B. Absence of Natural Language information

In our work, we use only categorical features of nodes and edges of code graph, e.g. AST node types and Python type annotations. Therefore, it would be challenging to apply it directly for tasks such as code generation, because the representation doesn't encode any information about variable names.

There are several approaches that would help address this issue. Firstly, it is possible to use the model output as graph encoding that would be later fed into another model along with tokenized code [51]. This approach could also address the issue from the previous Section VI-A, since types would be treated as a set of text tokens [34]. Secondly, it is possible to use neural networks to infer variables' names from the context they are used in [52].

VII. FUTURE WORK

In this work, we explored the application of Graph Transformers for type inference. The versatile architecture of the proposed solution lets us explore other tasks.

A. Universal code graph representation

If a universal version of graph code representation is used, similar to CPG [30], we can train the model for multiple programming languages [27]. However, because of the differences of type systems, separate models would be trained for each programming language for better results.

B. Detecting duplicates

It is crucial to address the issue of duplicates in source code to train neural networks for code [25, 26]. Several architectures have already been used for such task: Transformers [53], GNNs [54] and RNNs [55]. We believe that the graph representation obtained with our model can be successfully used for code clone detection.

C. Code and docstring generation

Firstly, we can train the model using a technique similar to generative pretrained models [56, 57] or masked language models [51] to generate code. Secondly, our model can be used to generate code summarization or docstring generation [58, 59]. This could only be possible if we adapt some of the approach discussed in the previous Section VI-B

D. Vulnerability and error detection

Another useful task is to detect errors and generate fixes [60, 61]. This is possible by simply adding features that contain error indication or types. Similar approach can be used to scan for vulnerabilities [41, 44, 40]. Fixing bugs and vulnerabilities, however, would imply that the graph structure could change. Therefore, solving this task would require the model to be modified for graph generation [62].

E. Refactoring

Finally, we can extend our model with information about changes to analyze them and propose refactoring possibilities [63]. This goal could be achieved by using the model from the previous Section VII-D.

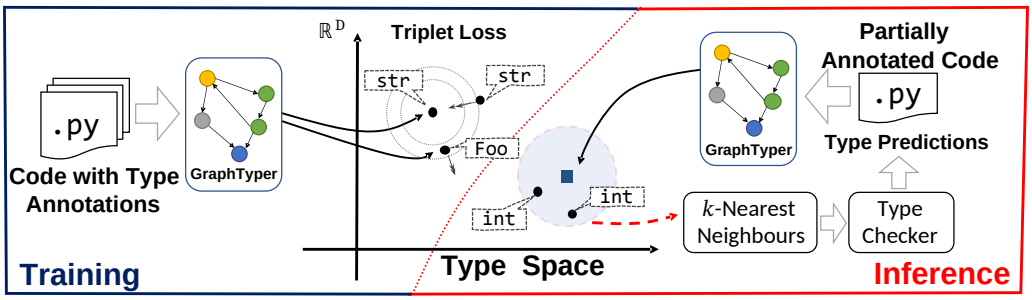

Fig. 3. Solution to the problem using Deep Similarity Learning [25].

VIII. Conclusion

As for the conclusion, we were able to create a universal model based on TokenGT [13] and code represented as graphs. One of the most important advantages of this model is that it uses the code graph directly. Secondly, the model can be modified to fit other tasks, such as code generation and summarization, docstring generation, refactoring and many more. The code graph can also be extended by different features and node types, since the representation does not differ depending on graph structure.

IX. Acknowledgments

This research was supported in part through computational resources of HPC facilities at HSE University [64].

References

[1] Dan Hendrycks et al. "Measuring coding challenge competence with apps". In: *arXiv preprint arXiv:2105.09938* (2021).

[2] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[3] Yujia Li et al. "Competition-Level Code Generation with AlphaCode". In: (), p. 74.

[4] Erik Nijkamp et al. "A Conversational Paradigm for Program Synthesis". In: *arXiv preprint arXiv:2203.13474* (2022).

[5] Frank F. Xu et al. "A Systematic Evaluation of Large Language Models of Code". In: *arXiv preprint arXiv:2202.13169* (2022).

[6] German Arsenovich Arutyunov and Sergey Mikchailovitch Avdoshin. "Big Transformers for Code Generation". In: *Proceedings of the Institute for System Programming of the RAS* 34.4 (2022). Publisher: Institute for System Programming of the Russian Academy of Sciences, pp. 79–88. DOI: 10.15514/ispras-2022-34(4)-6. URL: https://doi.org/10.15514%2Fispras-2022-34%284%29-6.

[7] S.M. Avdoshin and G.A. Arutyunov. "Code Analysis and Generation Methods Using Neural Networks: an Overview". In: *INFORMATION TECHNOLOGIES* 28.7 (2022), pp. 378–391. DOI: 10.17587/it.28.378-391.

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to represent programs with graphs". In: *arXiv preprint arXiv:1711.00740* (2017).

[9] Martin White et al. "Deep learning code fragments for code clone detection". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[10] Huihui Wei and Ming Li. "Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code." In: *IJCAI*. 2017, pp. 3034–3040.

[11] Lili Mou et al. "Convolutional neural networks over tree structures for programming language processing". In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[12] Krzysztof Choromanski et al. *Rethinking Attention with Performers*. _eprint: 2009.14794. 2020.

[13] Jinwoo Kim et al. *Pure Transformers are Powerful Graph Learners*. July 6, 2022. DOI: 10.48550/arXiv.2207.02505. arXiv: 2207.02505[cs]. URL: http://arxiv.org/abs/2207.02505 (visited on 09/04/2022).

[14] Devin Kreuzer et al. *Rethinking Graph Transformers with Spectral Attention*. Number: arXiv:2106.03893. Oct. 27, 2021. DOI: 10.48550/arXiv.2106.03893. arXiv: 2106.03893[cs]. URL: http://arxiv.org/abs/2106.03893 (visited on 06/17/2022).

[15] Vijay Prakash Dwivedi and Xavier Bresson. *A Generalization of Transformer Networks to Graphs*. Number: arXiv:2012.09699. Jan. 24, 2021. DOI: 10.48550/arXiv.2012.09699. arXiv: 2012.09699[cs]. URL: http://arxiv.org/abs/2012.09699 (visited on 06/17/2022).

[16] Chengxuan Ying et al. *Do Transformers Really Perform Bad for Graph Representation?* Nov. 23, 2021. DOI: 10.48550/arXiv.2106.05234. arXiv: 2106.05234[cs]. URL: http://arxiv.org/abs/2106.05234 (visited on 10/10/2022).

[17] Paul Mooney Julia Elliott. *2021 Kaggle Machine Learning & Data Science Survey*. 2021. URL: https://kaggle.com/competitions/kaggle-survey-2021.

[18] Martin P. Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: *IEEE Software* 26.6 (2009), pp. 27–34. DOI: 10.1109/MS.2009.193.

[19] Martin P. Robillard and Robert Deline. "A field study of API learning obstacles". In: *Empirical Softw. Engg.*

16.6 (Dec. 2011), pp. 703–732. ISSN: 1382-3256. DOI: 10.1007/s10664-010-9150-8. URL: https://doi.org/10.1007/s10664-010-9150-8.

[20] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal K. Roy. "Useful, But Usable? Factors Affecting the Usability of APIs". In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 151–155. DOI: 10.1109/WCRE.2011.26.

[21] Nabeel Alzahrani et al. "Python Versus C++: An Analysis of Student Struggle on Small Coding Exercises in Introductory Programming Courses". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE '18. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 86–91. ISBN: 9781450351034. DOI: 10.1145/3159450.3160586. URL: https://doi.org/10.1145/3159450.3160586.

[22] Lars Reimann and Günter Kniesel-Wünsche. *Safe-DS: A Domain Specific Language to Make Data Science Safe*. 2023. arXiv: 2302.14548 [cs.SE].

[23] *Pyre: A performant type-checker for Python 3*. URL: https://pyre-check.org (visited on 05/12/2024).

[24] *PyType: A static type analyzer for Python code*. URL: https://github.com/google/pytype (visited on 05/12/2024).

[25] Miltiadis Allamanis et al. "Typilus: Neural Type Hints". In: *PLDI*. 2020.

[26] Amir M. Mir et al. "Type4py: Deep similarity learning-based type inference for python". In: *arXiv preprint arXiv:2101.04470* (2021).

[27] Kesu Wang et al. "Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification". In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. May 16, 2022, pp. 390–400. DOI: 10.1145/3524610.3527915. arXiv: 2205.00424[cs]. URL: http://arxiv.org/abs/2205.00424 (visited on 11/04/2022).

[28] Ze Tang et al. *AST-Transformer: Encoding Abstract Syntax Trees Efficiently for Code Summarization*. Dec. 2, 2021. DOI: 10.48550/arXiv.2112.01184. arXiv: 2112.01184[cs]. URL: http://arxiv.org/abs/2112.01184 (visited on 11/04/2022).

[29] Zeyu Sun et al. "Treegen: A tree-based transformer architecture for code generation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. Issue: 05. 2020, pp. 8984–8991.

[30] Fabian Yamaguchi et al. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[31] Jiahao Liu et al. "Learning Graph-based Code Representations for Source-level Functional Similarity Detection". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 345–357. DOI: 10.1109/ICSE48619.2023.00040.

[32] Michael Pradel et al. "TypeWriter: neural type prediction with search-based validation". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 209–220. ISBN: 9781450370431. DOI: 10.1145/3368089.3409715. URL: https://doi.org/10.1145/3368089.3409715.

[33] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. "Learning type annotation: is big data enough?" In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1483–1486. ISBN: 9781450385626. DOI: 10.1145/3468264.3473135. URL: https://doi.org/10.1145/3468264.3473135.

[34] Yun Peng et al. *Generative Type Inference for Python*. 2023. arXiv: 2307.09163 [cs.SE].

[35] Jessica Schrouff et al. "Inferring javascript types using graph neural networks". In: *arXiv preprint arXiv:1905.06707* (2019).

[36] Zihan Liu et al. *NER-BERT: A Pre-trained Model for Low-Resource Entity Tagging*. 2021. arXiv: 2112.00405 [cs.CL].

[37] Harshil Darji, Jelena Mitrović, and Michael Granitzer. "German BERT Model for Legal Named Entity Recognition". In: *Proceedings of the 15th International Conference on Agents and Artificial Intelligence*. SCITEPRESS - Science and Technology Publications, 2023. DOI: 10.5220/0011749400003393. URL: http://dx.doi.org/10.5220/0011749400003393.

[38] David Bieber et al. *Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions*. 2022. arXiv: 2203.03771 [cs.LG].

[39] Shiyu Sun et al. *Exploring Security Commits in Python*. 2023. arXiv: 2307.11853 [cs.CR].

[40] Van-Anh Nguyen et al. "ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection". In: *arXiv preprint arXiv:2110.07317* (2021).

[41] Zhen Li et al. "Vuldeepecker: A deep learning-based system for vulnerability detection". In: *arXiv preprint arXiv:1801.01681* (2018).

[42] Sicong Cao et al. "Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection". In: *Information and Software Technology* 136 (2021). Publisher: Elsevier, p. 106576.

[43] Zhen Li et al. "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities". In: *IEEE Transactions on Dependable and Secure Computing* (2021). Conference Name: IEEE Transactions on Dependable and Secure Computing, pp. 1–1. ISSN: 1941-0018. DOI: 10.1109/TDSC.2021.3051525.

[44] Rebecca Russell et al. “Automated vulnerability detection in source code using deep representation learning”. In: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[45] Qiaoyu Tan et al. *MGAE: Masked Autoencoders for Self-Supervised Learning on Graphs*. 2022. arXiv: 2201.02534 [cs.LG].

[46] Sixiao Zhang et al. *Graph Masked Autoencoders with Transformers*. 2022. arXiv: 2202.08391 [cs.LG].

[47] Zhenyu Hou et al. *GraphMAE: Self-Supervised Masked Graph Autoencoders*. 2022. arXiv: 2205 . 10803 [cs.LG].

[48] S. Chopra, R. Hadsell, and Y. LeCun. “Learning a similarity metric discriminatively, with application to face verification”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. 2005, 539–546 vol. 1. DOI: 10.1109/CVPR.2005.202.

[49] Wentong Liao et al. *Triplet-based Deep Similarity Learning for Person Re-Identification*. 2018. arXiv: 1802.03254 [cs.CV].

[50] T. Cover and P. Hart. “Nearest neighbor pattern classification”. In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: 10 . 1109 / TIT . 1967 . 1053964.

[51] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. *StructCoder: Structure-Aware Transformer for Code Generation*. June 10, 2022. DOI: 10.48550/arXiv.2206.05239. arXiv: 2206.05239[cs]. URL: http://arxiv.org/abs/2206.05239 (visited on 10/11/2022).

[52] Rohan Bavishi, Michael Pradel, and Koushik Sen. *Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts*. 2018. arXiv: 1809.05193 [cs.SE].

[53] Aiping Zhang et al. “Efficient transformer with code token learner for code clone detection”. In: *J. Syst. Softw.* 197.C (Mar. 2023). ISSN: 0164-1212. DOI: 10.1016/j.jss.2022.111557. URL: https://doi.org/10.1016/j.jss.2022.111557.

[54] Wenhan Wang et al. “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.

[55] Jitendra Yasaswi, Suresh Purini, and C.V. Jawahar. “Plagiarism Detection in Programming Assignments Using Deep Features”. In: *2017 4th IAPR Asian Conference on Pattern Recognition (ACPR)*. 2017, pp. 652–657. DOI: 10.1109/ACPR.2017.146.

[56] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[57] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[58] Antonio Valerio Miceli Barone and Rico Sennrich. “A parallel corpus of python functions and documentation strings for automated code documentation and code generation”. In: *arXiv preprint arXiv:1707.02275* (2017).

[59] Xuye Liu et al. “HAConvGNN: Hierarchical attention based convolutional graph neural network for code documentation generation in jupyter notebooks”. In: *arXiv preprint arXiv:2104.01002* (2021).

[60] Sahil Bhatia and Rishabh Singh. “Automated correction for syntax errors in programming assignments using recurrent neural networks”. In: *arXiv preprint arXiv:1603.06129* (2016).

[61] Alexandru Marginean et al. “Sapfix: Automated end-to-end repair at scale”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[62] Ahmad Khajenezhad et al. *Gransformer: Transformer-based Graph Generation*. 2022. arXiv: 2203 . 13655 [cs.LG].

[63] Rocío Cabrera Lozoya et al. “Commit2vec: Learning distributed representations of code changes”. In: *SN Computer Science* 2.3 (2021). Publisher: Springer, pp. 1–16.

[64] P. S. Kostenetskiy, R. A. Chulkevich, and V. I. Kozyrev. “HPC Resources of the Higher School of Economics”. In: *Journal of Physics: Conference Series* 1740.1 (Jan. 2021). Publisher: IOP Publishing, p. 012050. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1740/1/012050. URL: https://doi.org/10.1088/1742-6596/1740/1/012050 (visited on 03/21/2022).