***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% code in the company. Copilot is based on Codex, a Transformer model trained on code as sequence. However, 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 very good 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 feasible to relentlessly increase the length of context sequence in a Transformer. Therefore, a different approach is needed to boost the efficiency in the task of code synthesis [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 programs 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 from both: 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. 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. We compare the results with the models from previous work in Table I [17].

A. Metrics

To test the model we use two metrics from the Typilus paper [17]:

Exact Match: Predicted and ground truth types match exactly.

Match up to Parametric Type: Exact match when ignoring all type parameters.

III. Previous Work

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]. 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].

AST and DFG have already been used with Transformers in the code generation and summarization tasks [18, 19, 20]. Moreover, graph representation of code has been used for the task of type inference in dynamically typed programming languages such as Python [17] and Javascript [21].

However, the power of 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 [22, 17, 23, 24, 25] 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 [17]. We clone these repositories and utilize pytype 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 133,000 near code duplicates to prevent bias.

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.

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
	Typilus	45.47	55.02
	Type4Py	50.70	64.58
	TypeWriter	57.60	72.15
Top-3	GraphTyper	56.10	58.30
	Typilus	63.70	67.30
	Type4Py	65.90	70.40
	TypeWriter	68.20	73.20
Top-5	GraphTyper	66.10	74.20
	Typilus	71.60	79.80
	Type4Py	72.70	80.90
	TypeWriter	73.30	81.50
Top-10	GraphTyper	75.80	80.60
	Typilus	78.10	83.80
	Type4Py	78.70	84.70
	TypeWriter	79.20	85.40

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 accelerate 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 type 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. As loss, weighted cross entropy is used due to the imbalance of the dataset. The overall architecture of the model is displayed at Figure 1. We develop two types of Transformers: Masked Transformer Encoder-only Model and Masked Transformer Encoder-Decoder Model.

1) Masked Transformer Encoder-only Model: Predicting type annotations in graph domain is a node classification task. However, since we are using a Pure Transformer with graphs represented as sequence of tokens, the task can be reduced to token classification. In the Natural Language Processing (NLP) domain this is a very common 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 [26, 27]. We adapt similar architecture by randomly masking type annotations. We then apply a classifier head to the output of TokenGT [13] to get logits of type annotations.

2) Masked Transformer Encoder-Decoder Model: In addition, we develop an Encoder-Decoder model architecture, inspired by GMAE [28]. However, instead of masking the entire nodes as in the mentioned paper, we only mask features corresponding to type annotations same as with Encoder-only Model.

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 [26]. For example, error [29] and vulnerability [30] data can be added to the code graph to detect and fix them [31, 32, 33, 34, 35].

V. EXPERIMENTS AND ABLATION RESULTS

To select the final model architecture we test different models by:

1) Increasing the number of parameters (Big)
2) Testing different context length, i.e. maximum number of nodes in graph (512 vs 1024)
3) Validated the necessity of node and type identifiers that encode graph structure (Ablated)
4) Testing different Transformer architectures (Encoder-only vs Encoder-Decoder)

For our experiments and ablation analysis we train and test the models on one sample repository. To test the models we calculate Top-n predictions similar to the previous work [22]. Each model is trained on dataset preprocessed to contain maximum 512 and 1024 nodes in a graph. Table II depicts the results of the experiments and ablation. As we can see, both models that increase the number of parameters, Big and Deep, also increase the predictive power of the model.

As for the context length, 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. Thus, in future research we will be working on finding a better and more compact graph representation of code.

Finally, our ablation analysis demonstrates that indeed the graph structure embeddings play key role in model quality. By removing them from the model we are left with a simple

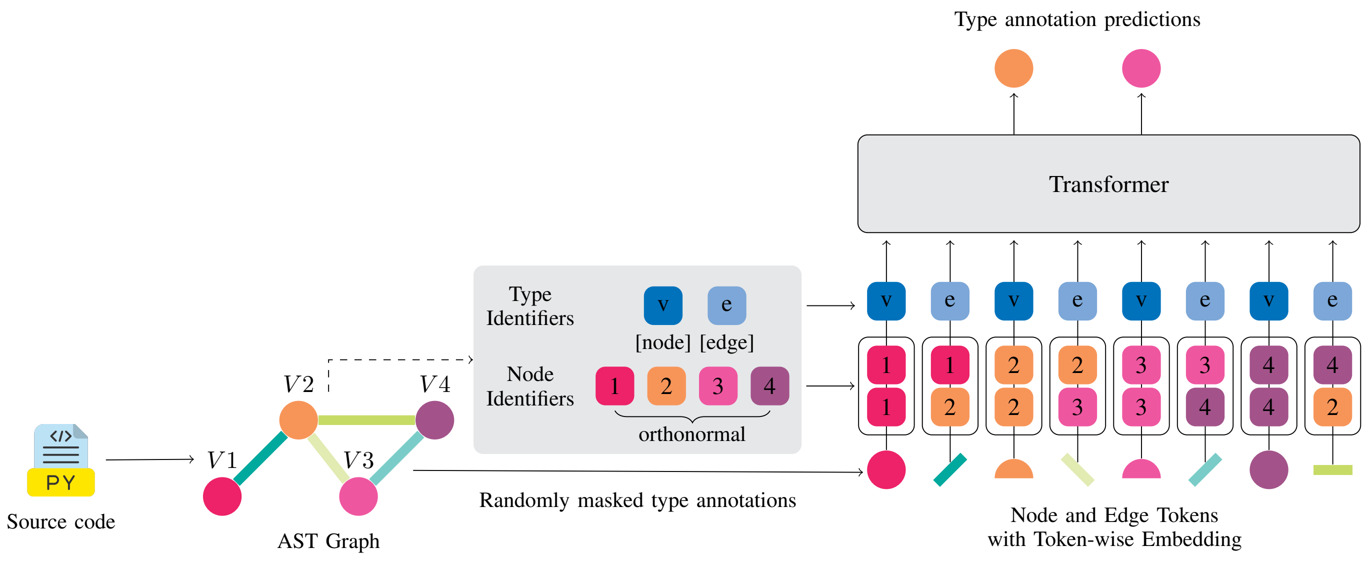

Fig. 1. GraphTyper Architecture

Transformer that makes predictions only based on AST nodes and edges types.

Therefore, in the resulting model we increase the number of layers, attention heads, as well as the dimension of the hidden layers. In addition, we preprocess the complete dataset to have 1024 nodes in graph at most. Finally, we add the node and type identifiers from the TokenGT paper [13]. The resulting model has 441 million parameters: 12 encoder layers, 16 attention heads, 2048 hidden dimension and 4096 feed-forward layer dimension.

VI. 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.

First, if a universal version of graph code representation is used the can train the model for multiple programming languages [18]. Second, we can train the model using a technique similar to generative pretrained models [36, 37] to generate code. Third, our model can be used to generate code summarization or docstring generation [38, 39]. Another useful task is to detect errors and generate fixes [40, 41, 42]. Finally, we can extend our model with information about changes to analyse them and propose refactoring possibilities [43].

VII. 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 the code graph is used directly by the model. 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

TABLE II
Expirement results of Top-n predictions for different model variants.

	% Match	Exact		Up to Parametric Type	
	Context Length	512	1024	512	1024
Top-n	Model				
Top-1	Ablated (51 mln)	10.15	21.39	19.46	31.09
	Base (51 mln)	30.88	31.12	36.55	35.83
	Deep (214 mln)	26.69	32.93	29.93	37.95
	Big (331 mln)	30.90	32.77	33.59	38.29
	Final (432 mln)	29.25	34.71	36.97	36.43
Top-3	Ablated (51 mln)	15.06	38.94	29.40	54.08
	Base (51 mln)	40.33	42.79	50.37	56.30
	Deep (214 mln)	40.15	44.07	48.82	53.55
	Big (331 mln)	42.75	43.36	49.62	51.46
	Final (432 mln)	39.93	45.47	53.19	55.02
Top-5	Ablated (51 mln)	16.81	41.19	37.91	56.97
	Base (51 mln)	42.82	45.30	56.01	60.12
	Deep (214 mln)	44.18	46.23	57.11	60.36
	Big (331 mln)	47.28	45.21	58.25	55.92
	Final (432 mln)	46.58	50.70	63.29	64.58
Top-10	Ablated (51 mln)	22.97	46.27	46.72	62.05
	Base (51 mln)	48.03	48.72	64.98	64.59
	Deep (214 mln)	49.49	52.69	67.77	69.63
	Big (331 mln)	52.72	50.25	68.53	66.20
	Final (432 mln)	54.21	57.60	72.30	72.15

features and node types, since the representation does not differ depending on graph structure.

VIII. Acknowledgments

This research was supported in part through computational resources of HPC facilities at HSE University [44].

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

[18] 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).

[19] 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).

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

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

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

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

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

[34] 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.

[35] 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.

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

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

[38] 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).

[39] 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).

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

[41] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. Oct. 22, 2018. DOI: 10.48550/arXiv.1802.09477. arXiv: 1802.09477[cs, stat]. URL: http://arxiv.org/abs/1802.09477 (visited on 10/23/2022).

[42] 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.

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

[44] 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).