***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 [11, 8, 15, 19]. Nevertheless, the versatile Transformer architecture has displayed good results on several benchmarks, in the recent work [27] 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 [3].

First of all, an expressive code representation has to be selected. Several ways including token-based, structured and graph-based approaches have been reviewed [21]. 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 [1]. 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 [26], LSTM [25] and CNN [18] 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 [1].

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 training parallelization and graph code representation created from AST. To make this possible we will use Pure Transformers [12] instead of models that have some architectural alterations to support graph structure [14, 9, 28].

Our main contributions:

1) Source code graph representation with AST
2) Transformer model that can be directly trained on graph structure data and applied for different tasks including code and documentation generation
3) Model pretrained on Python source code represented as graph

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 1 [2].

A. Dataset

To train and test the model we gathered 600 Python repositories from GitHub containing type annotations from Typilus [2]. 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. Finally, they split the data into train-validation-test sets with proportions of 70-10-20, respectively.

B. Metrics

To test the model we use two metrics from the Typilus paper [2]:

Exact Match: τ_p and τ_g match exactly.
Match up to Parametric Type: Exact match when ignoring all type parameters.

III. Previous Work

A. 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 [12, 14, 9, 28].

AST and DFG have already been used with Transformers in the code generation and summarization tasks [24, 23, 22], as well as

Name	% Exact Match	% Match up to Parametric Type
GraphTyper	41%	45.9%
Typilus	54.6%	64.1%

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

IV. Proposed Solution

A. Model Architecture

We base our model architecture on TokenGT [12]. For training, cross entropy loss with weights is used due to the imbalance of the dataset.

V. Experiment Results and Ablation

For now, the model has been trained and tested on one repository. The resulting accuracy for all types is 41% and 45.9% accuracy up to parametric type.

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 AST parsing is used the can train the model for multiple programming languages [24]. Second, we can train the model using a technique similar to generative pretrained models [20, 6] to generate code. Third, our model can be used to generate code summarization or docstring generation [4, 16]. Another useful task is to detect errors and generate fixes [5, 10, 17]. Finally, we can extend our model with information about changes to analyse them and propose refactoring possibilities [7].

VII. Conclusion

As for the conclusion, we were able to create a universal model based on TokenGT [12] 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 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 [13].

References

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

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

[3] 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.

[4] 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).

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

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

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

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

[9] 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).

[10] 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).

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

[12] 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).

[13] 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).

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

[16] 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).

[17] 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.

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

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

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

[21] 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.

[22] 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.

[23] 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).

[24] 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).

[25] 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.

[26] 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.

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

[28] 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).