Karlsruhe University of Applied Sciences

Department of Computer Science and Business Information Systems

Research Proposal

Master's Thesis

Leveraging Syntactic and Semantic Awareness in Pretrained Transformers for Code Understanding and Generation

Author:	Daniel Schneider
Matriculation Number:	91145
Workplace:	SAP SE, Walldorf
First Supervisor:	Prof. Dr. Dennis Janka[1]
Second Supervisor:	Prof. Dr. Reimar Hofmann[1]
Advisors:	Martin Kraemer[2], Eduardo Vellasques[2]

[1]Karlsruhe University of Applied Sciences

[2]SAP SE

Contents

List of Abbreviations

BPE ..... Byte Pair Encoding
LM ..... Language Model
NLG ..... Natural Language Generation
NLP ..... Natural Language Processing

1 Motivation

In recent years, generative artificial intelligence (GenAI) has gained much attention across society. Especially, with publicly available GenAI tools like ChatGPT [1], society tries to automate various tasks in different fields to increase its productivity. The focus of this thesis shall be the field of software engineering, where there has also been some effort to increase productivity by leveraging GenAI tools like GitHub Copilot (Fig. 1.1) [2].

These tools are often based on a deep learning model that leverages the Transformer architecture [3]. While this architecture was first employed for the understanding and generation of natural language, the main focus to increase productivity in software engineering tasks is the understanding (CU) and generation (CG) of code [3]. Typical tasks that require CU and CG include code translation, text-to-code generation, and code completion.

By leveraging the Transformer architecture for CU and CG, many state-of-the-art models do not adapt this architecture or training strategies [4]. Thus, these models process code like natural language. However, code differs from natural language in its structure and syntax. While code is highly structured following strict syntactic rules, natural language is more flexible, ambiguous and context-dependent [4].

Therefore, this thesis shall investigate the incorporation of code syntax and semantics when leveraging the Transformer architecture to improve its performance in terms of CU and CG. For this, an adapted Transformer architecture that incorporates the syntax and data flow of the source and target code shall be used [4].

However, pretraining Transformer models is not always feasible in practice, as it requires a lot of computational resources, vast datasets, and extensive training time [5]. For instance, the smallest Llama 2 model, with seven billion parameters, required 184,320 GPU hours for pretraining on NVIDIA A100 GPUs [6]. Renting eight NVIDIA A100 GPUs at a rate of $32.77 per hour on an AWS cluster would result in a total cost of approximately $755,000 [7]. Accounting for multiple iterations for refinement and additional expenses, the total cost of pretraining such a model could easily reach several million USD.

For these reasons, an unadapted pretrained Transformer model for CU and CG shall be leveraged in this thesis. Its architecture shall be adapted to incorporate code syntax and semantics as described above. Subsequently, continued pretraining shall be performed for this adapted Transformer model and its performance shall be evaluated in terms of CU and CG. Continued pretraining can be performed more efficiently than full pretraining, requiring fewer computational resources, less data, and less training time [8, 9]. Thus, the evaluation in this thesis shall indicate whether continued pretraining, in contrast to

full pretraining, is sufficient for employing the described approach in practice.

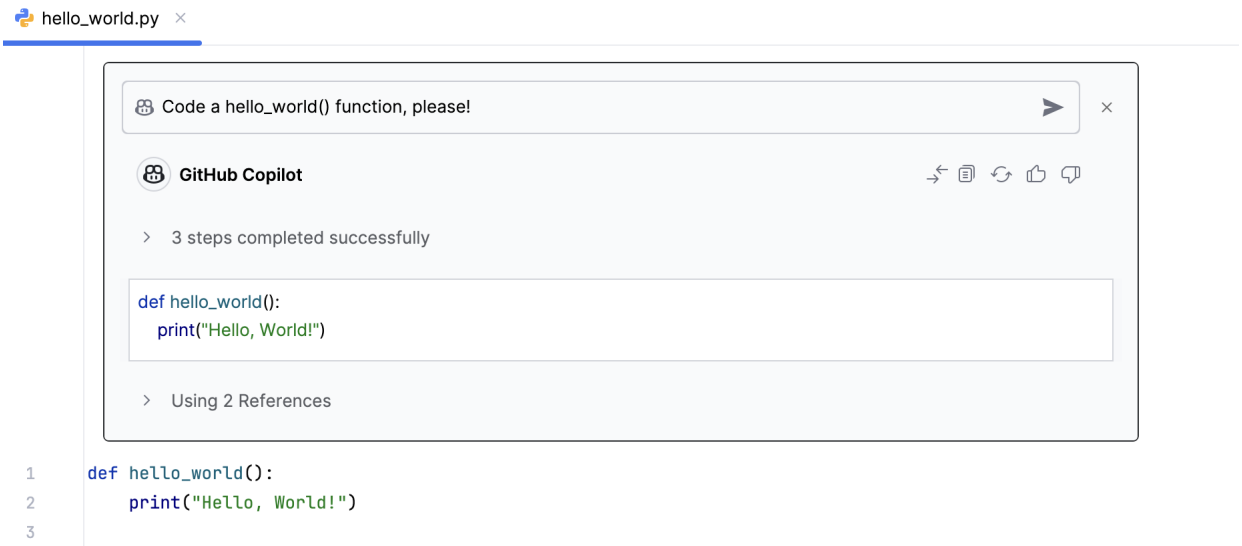

Figure 1.1: GitHub Copilot [2] is a GenAI tool for code understanding and generation. It was developed to increase productivity in software engineering tasks.

2 Objectives

In this thesis, a pretrained Transformer model for CU and CG that does not incorporate code syntax and semantics shall be leveraged. The architecture of this Transformer model shall be adapted to incorporate code syntax and semantics by leveraging the abstract syntax tree (AST) and the data flow graph (DFG). Based on this adapted Transformer model, continued pretraining shall be performed. The subsequent evaluation shall contribute to answering the following research question (RQ) of this thesis.

RQ: *Does incorporating code syntax and semantics improve the performance of a pretrained Transformer model in CU and CG when performing continued pretraining?*

As explained in Section 1, many state-of-the-art Transformer models for CU and CG do not incorporate code syntax and semantics. Leveraging such models by adapting them and improving their performance based on continued pretraining would be beneficial in practice, as fully pretraining the adapted models could be avoided. Thus, answering this RQ shall indicate whether continued pretraining, in contrast to full pretraining, is sufficient for improving the performance of such adapted Transformer models.

Building on this RQ, further analyses can be conducted in this thesis. For instance, an ablation study can be performed to assess the individual contributions of code syntax and code semantics to the performance of the adapted Transformer model. Additionally, the ablation study could explore different auxiliary losses for training the adapted Transformer

model with respect to code syntax and semantics. Furthermore, various techniques and configurations for continued pretraining can be compared.

3 Related Work

Some approaches model code completion as a langauge modeling task predicting the next node in the target AST. For that purpose, Li et al. [10] traverse the source AST and flatten it to be inputted as a sequence of nodes into an LSTM. They further apply an extended self-attention mechanism for the hidden states of the LSTM, which accounts for parent-child relationships between AST nodes. Kim et al. [11] also traverse the source AST, but provide the corresponding nodes as a sequence input to a Transformer. They also extend the self-attention mechanism to account for AST path-based relationships. Alon et al. [12] embed each root-leaf path of the source AST based on the final hidden states of stacked LSTM layers and generate the target AST node-by-node for code completion.

For text-to-code generation, Rabinovich et al. [13] utilize a modular decoder whose submodules are composed to generate the target AST recursively in a top-down manner. Brockschmidt et al. [14], Sun et al. [15], Yin and Neubig [16] leverage programming language grammars to sequentially generate the target AST node-by-node with dedicated neural architectures. Based on the natural language input and the partial target AST generated so far, the most probable production rule is chosen to generate the next target AST node. Jiang et al. [17] extend an LSTM decoder with an attention mechanism that incorporates all previously chosen production rules to predict the production rule for generating the next target AST node.

While all of the above approaches leverage the syntactic code structure based on the AST, Guo et al. [18] leverage the semantic code structure based on the DFG. They adapt the self-attention mechanism for their pretrained encoder-only model, called GraphCodeBERT, to account for relationships between variables in the DFG. Further, they employ a pretraining task predicting these relationships.

Leveraging both the AST and DFG, Tipirneni et al. [4] adapt their pretrained encoder-decoder model, called StructCoder. The encoder incorporates root-leaf path embeddings of the source AST, variable embeddings of the source DFG, and an adapted self-attention mechanism to account for the source AST and source DFG. Further, the decoder is trained based on two auxiliary tasks predicting root-leaf paths in the target AST and data flow relationships in the target DFG, respectively.

GraphCodeBERT and StructCoder are one of the few pretrained models for CU and CG that incorporate code syntax and semantics, while other pretrained models for CU and CG primarily employ pretraining tasks originally developed for natural language [4].

4 Background

4.1 Natural Language Processing

Natural Language Processing (NLP) is the subfield of computer science that enables communication between humans and computers [19]. By understanding, interpreting, and generating natural language, NLP supports applications like chatbots, search engines, and machine translation. This subsection introduces fundamental concepts used in NLP to process natural language.

4.1.1 Tokenization

Tokenization is the process of translating natural language text into a sequence of tokens [20]. It is an essential step that bridges communication between humans and computers, as the generated tokens can be processed by subsequent NLP algorithms. Tokenization is performed based on a tokenizer that has learned a mapping

$$tok : W \rightarrow V^* \tag{1}$$

This mapping is learned by applying a tokenization algorithm on a text corpus D, where W is the set of any possible natural language text, V is the vocabulary of learned tokens, and V^* contains all possible token sequences from V [20]. For example,

$$tok(\text{I love Natural Langauge Processing!}) = (\text{I, love, Natural, Language, Processing, !})$$

Often, V contains tokens for special cases that occur during tokenization [20]. For instance, `<bos>` and `<eos>` could be tokens indicating the beginning and ending of a token sequence, respectively. Also, `<unk>` could be a token indicating an unkown mapping that the tokenizer has not learned based on D. Sometimes, it might also be useful to pad all token sequences to a uniform length by appending a token such as `<pad>`.

For learning the mapping in (1), multiple tokenization approaches based on different word granularity levels can be employed [20].

Word-based Tokenization [20, 21]**:**
Word-based tokenization algorithms process natural language text based on words, as they segment text into individual words as the primary unit of tokens. Natural language text is segmented based on spaces and punctuation, resulting in a unique token for each word contained in D. As a result, V contains a relatively large number of tokens (i.e. words), which might deteriorate the performance of subsequent NLP algorithms. Further, words not contained in D will be mapped to the `<unk>` token. Often, this is the case for infrequent words (e.g. a compound word like 'Lebensversicherungsangestellter' in German).

Character-based Tokenization [20, 21]**:**
Character-based tokenization algorithms process natural language text based on characters, as they segment text into individual characters as the primary unit of tokens. As a result, V contains a unique token for each character contained in D. On the one hand, there is no need for the `<unk>` token anymore. A word not contained in D will be segmented into its individual characters and mapped to the corresponding token sequence. On the other hand, resulting token sequences will contain a relatively large number of tokens, which might increase the computational costs for subsequent NLP algorithms.
Subword-based Tokenization [20, 21]**:**
Subword-based tokenization leverages the advantages of word-based and character-based tokenization, while eliminating their disadvantages. One subword-based tokenization algorithm is Byte Pair Encoding (BPE) [22, 23]. In BPE, V is initialized with a unique token for each character contained in D like in character-based tokenization. Iteratively, the most frequent pair of tokens in D is merged into a new token and added to V. BPE terminates based on a stopping criterion such as the maximum number of tokens contained in V. As a result, the size of token sequences and V will be lower compared to character-based and word-based tokenization, respectively.

4.1.2 Language Models

Language models (LMs) are machine learning models that model natural language by assigning a probability to an entire token sequence [21]. More formally, an LM can be described by the probability function

$$P : V^* \to [0, 1], (t_1, t_2, \ldots, t_N) \mapsto P(t_1, t_2, \ldots, t_N) \tag{2}$$

P assigns a probability $P(t_1, t_2, \ldots, t_N)$ to each possible token sequence $(t_1, t_2, \ldots, t_N) \in V^*$ based on a vocabulary of all possible tokens V. The probability function P is derived by training the LM on a text corpus D. Thus, P represents the likelihood of the occurrence of token sequences based on D. By applying the chain rule of probability, the joint probability of a token sequence can be decomposed into conditional probabilities of tokens given previous tokens [21]:

$$P(t_1, t_2, \ldots, t_N) = \prod_{n=1}^{N} P(t_n \mid t_1, t_2, \ldots, t_{n-1}) \tag{3}$$

LMs are fundamental for a variety of NLP tasks including natural language generation (NLG), which will be explained in the following section.

4.1.2.1 Natural Language Generation

By using the ability of assigning probabilities to token sequences, LMs can be employed to predict the next token given a sequence of previous tokens [24]. Thus, LMs can generate natural language text by iteratively generate the next token based on previously generated tokens. Given previously generated tokens $(t_1, t_2, \ldots, t_{n-1}) \in V^*$, the LM computes the probability for each next token $t_i \in V$ for $i \in \{1, 2, \ldots, |V|\}$:

$$P(t_n = t_i \mid t_1, t_2, \ldots t_{n-1}) \tag{4}$$

Effectively, the LM computes a conditional probability distribution over all tokens t_i [24]. Based on this conditional probability distribution, the LM can select the token to be generated next. The generated token is then appended to the previously generated tokens and the process repeats based on this updated token sequence until the `<eos>` token is generated. This generation process is known as autoregressive decoding and the LM can employ different decoding strategies to select the next token based on the computed conditional probability distribution.

Greedy Decoding [21]**:**
Greedy decoding is a deterministic decoding strategy that always selects the most probable next token given previously generated tokens:

$$t_n = \underset{t_i \in V}{\operatorname{argmax}}\ P(t_i \mid t_1, t_2, \ldots, t_{n-1}) \tag{5}$$

Thus, based on greedy decoding, the LM generates locally optimal tokens that might not lead to a global optimal token sequence.

Sampling Decoding [21]**:**
Sampling decoding is a non-deterministic decoding strategy that selects the next token $t_i \in V$, where the inequality

$$\sum_{j=0}^{i-1} P(t_j \mid t_1, t_2, \ldots, t_{n-1}) < r \leq \sum_{j=0}^{i} P(t_j \mid t_1, t_2, \ldots, t_{n-1}) \tag{6}$$

based on a uniform random number $r \sim U(0, 1)$ is fulfilled. Thus, based on sampling decoding, the LM can generate different possible token sequences. However, tokens with a relatively small probability could accumulate to a relatively high probability, such that the probability of generating any token with a small probability is relatively high. This effect can be mitigated by top-k or top-p sampling, where only the k most probable tokens or only the tokens with a probability higher than p, respectively, are taken into account for selecting the next token.

Beam Search Decoding [21]**:**
In beam search decoding, the LM maintains the most probable k token sequences, called beams, at each generation step. For that, the LM computes the conditional probability

distribution based on each beam to compute extended beams with a next token appended. Then, the LM selects the most probable k extended beams to continue with the next generation process. In contrast to greedy decoding, beam search decoding results in a better approximation of the global optimal token sequence in general. However, beam search decoding requires higher computational costs for decoding.

4.1.2.2 Training

4.1.2.3 Architecture

4.1.3 Vector Representations

4.2 Transformer

4.2.1 Encoder-Decoder

4.2.2 Attention Mechanism

4.2.3 Encoder-Only

4.2.4 Decoder-Only

4.3 Code Understanding & Code Generation

4.3.1 Task

4.3.2 Code LLMs

4.4 Transfer Learning

4.4.1 Continued Pretraining

4.4.2 Finetuning

4.5 Abstract Syntax Tree

4.6 Data Flow Graph

5 Preliminary Thesis Structure

References

[1] OpenAI. Introducing ChatGPT, 2022. URL `https://openai.com/index/chatgpt/`. Accessed: 02 February, 2025.

[2] GitHub. GitHub Copilot. URL `https://github.com/features/copilot`. Accessed: 02 February, 2025.

[3] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. *Ieee Access*, 10:82434–82455, 2022.

[4] Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. Structcoder: Structure-aware transformer for code generation. *ACM Transactions on Knowledge Discovery from Data*, 18(3):1–20, 2024.

[5] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

[6] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[7] Amazon EC2 P4 Instances. URL `https://aws.amazon.com/ec2/instance-types/p4/`. Accessed: 23 February, 2025.

[8] Yong Xie, Karan Aggarwal, and Aitzaz Ahmad. Efficient continual pre-training for building domain specific large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 10184–10201, 2024.

[9] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964*, 2020.

[10] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.

[11] 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. IEEE, 2021.

[12] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International conference on machine learning*, pages 245–256. PMLR, 2020.

[13] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

[14] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

[15] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8984–8991, 2020.

[16] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[17] Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. An ast structure enhanced decoder for code generation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:468–476, 2021.

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[19] Sonja Grigoleit. Natural language processing. *Europäische Sicherheit & Technik: ES & T*, 2019.

[20] Hobson Lane and Maria Dyshel. *Natural language processing in action*. Simon and Schuster, 2025.

[21] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2025. URL `https://web.stanford.edu/~jurafsky/slp3/`. Online manuscript released January 12, 2025.

[22] Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2): 23–38, 1994.

[23] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[24] Chenhe Dong, Yinghui Li, Haifan Gong, Miaoxin Chen, Junxin Li, Ying Shen, and Min Yang. A survey of natural language generation. *ACM Computing Surveys*, 55 (8):1–38, 2022.