Syntax and Sensibility: Exploring Commentary Generation for Functional Code with Graph Neural Networks

1 Problem & Scope

In "Why functional programming matters", it is said that our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming, a language must provide good glue. Functional programming languages provide two kinds of glue - higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways. [7]

Functional programming is renowned for its expressive power, offering features like higher-order functions, immutability, and declarative paradigms that allow developers to write concise and abstract code. However, these same characteristics often make functional programs difficult to read and understand, particularly for those unfamiliar with the paradigms or advanced constructs like monads, lazy evaluation, or recursion patterns. The cognitive load increases as developers attempt to reason about the behaviour and purpose of functions or to onboard to new functional codebases.

This powerful programming paradigm seems to be underrepresented in terms of code intelligence tasks, such as code explainability and code generation. Furthermore, as per [12], LLMs not trained on functional programming languages seem to have trouble when transferring to functional programming languages.

Machine learning techniques, particularly those leveraging the structure of the code, not only the content of the code, have demonstrated success in generating code summaries for imperative and object-oriented languages. However, functional programming presents unique challenges, such as non-linear control flows, extensive use of compositionality, and a strong emphasis on immutability, which are not structurally addressed by current approaches.

The aim of this research is to explore the ability of code-structure aware AI models to generate meaningful summaries of functional code. In order to better represent structural information, correlated with commentary, we use an Abstract Syntax Tree based model, incorporating Convolutional Graph Neural Networks, inspired by the model proposed by Leclair et al. [1].

2 Related Work

As the novelty of this paper consists in the mixture of functional programming commentary generation with graph-based neural networks, there are only indirect ways to compare it to related work. The performance of the model will be measured both according to the performance of similar architectures on imperative languages, in order to validate the methodology, as well as to the performance of different architectures on the same dataset, in order to verify the performance. Due to the embedding of the AST structure into the neural network, an increased awareness of relationships such as recursive or higher-order functions would be expected, which are specific to the functional programming paradigm.

This paper is strongly related to Leclair et al. [1], as both models use the Graph Neural Networks approach, however, this paper is specialised on imperative languages, specifically Java. This model was trained on a data set of 2.1 million Java method-comment pairs and shows improvement over four baseline techniques, two from the software engineering literature, and two from machine learning literature. The key improvement lays in the use of Graph Neural Networks, which shows improved performance over models that process the flatten the AST as a linear structure, not a graph structure (see ast-attendgru [10]), over models that use random pairs of AST paths (see code2seq [2]).

The recent popularity of large language models specialised on coding tasks must be taken into account as well. Most of them were trained disproportionately on imperative languages, which impacts their ability to transfer to functional languages, as was observed in "Investigating the Performance of Language Models for Completing Code in Functional Programming Languages: a Haskell Case Study" [12], where the evaluated models were CodeGPT and UniXcoder [5]. The performance of GPT-4 seems to be generally good on small snippets of Haskell code, for instance, but it seems to be making type errors as well. The largest tech giants released their own versions of LLMs specialised on code, some of them being CodeLlama [11], from Meta, or CodeBERT [4] (with its variations, such as GraphCodeBERT), from Microsoft, or CodeGemma, from Google. The databases used for fine-tuning these models, for instance, CodeSearchNet [8] or BigCode, as well as the benchmarks, such as HumanEval [3], MBPP (Mostly Basic Python Problems) often contain a significant bias towards popular imperative programming languages like Python, Java, and C++.

This skew in the training data limits the exposure of such models to functional languages, leading to suboptimal performance in tasks requiring functional programming paradigms. Consequently, while these models demonstrate remarkable proficiency in generating and understanding code in dominant imperative languages, they may struggle to generalise effectively to less-represented languages. This underlines the importance of diversifying training datasets and tailoring fine-tuning processes to address such disparities, enabling more support for functional programming tasks.

3 Research Questions

- **RQ1:** Does the incorporation of AST structure in neural network models show potential for improving the performance of code summarisation tasks for functional programming languages?
- **RQ2:** What are the challenges faced by existing neural network models in summarising functional programming code?
- **RQ3:** Can Convolutional Graph Neural Networks be adapted to effectively incorporate Abstract Syntax Tree structures for generating meaningful comments on functional programming code?

4 Experimental Part [9]

The experimental part is performed with a focus on the Haskell programming language. Haskell is a general-purpose, purely functional programming language that was first introduced in 1987 by a committee of researchers. The committee aimed to standardize the growing field of functional programming, which had been developed through several earlier languages like LISP, ML, and Miranda. The name "Haskell" was chosen in honor of Haskell Curry, a logician whose work in combinatory logic influenced the development of functional programming. [6]

Haskell was designed with several key features that distinguished it from other programming languages, including lazy evaluation, currying, a strong static type system, and type inference. These features, along with its emphasis on immutability and purity, made Haskell well-suited for academic research and teaching, particularly in fields like programming language theory and formal methods.

4.1 Data Collection

The datasets used for the experimental part are the GitHub Code Haskell Function dataset and PHagenlocher-HaskellYTv0.1, available on Hugging Face.

GitHub Code Haskell Function contains entries with information about function code snippets, extracted from GitHub repositories. The available columns are repo_name, path, license, full_code, full_size, uncommented_code, uncommented_size, function_only_code, function_only_size, is_commented, is_signatured, n_ast_errors, ast_max_depth, n_whitespaces, n_ast_nodes, n_ast_terminals, n_ast_nonterminals, loc, cycloplexity. Details regarding the computation of uncommented_code and function_only_code are available on the Hugging Face page. The dataset is available on 3 partitions: training data (2.29M rows), validation data (327k rows) and test data (654k rows).

The PHagenlocher-HaskellYTv0.1 dataset contains code description and code snippets, the available columns being Description and Statement. There are 712 entries available.

4.2 Quality of Data and Comparison

In the case of GitHub Code Haskell Function, the commentary needs to be extracted by comparing the full_code column with the uncommented_code. For further use, it is assumed that the extracted commentary contains descriptive information about the corresponding function. The quality of the commentary might raise questions, as in practice, the commentary does not always describe what the function does, but rather to-do's or warnings.

In the case of the PHagenlocher-HaskellYTv0.1 dataset, the commentary is already available in a separate column. After manually comparing fragments of these two datasets, the latter seems to be more appropriate for the proposed task.

4.3 Data Preprocessing

The data preprocessing pipeline has supplementary steps for the GitHub Code Haskell Function dataset. An initial filtering is performed, eliminating the rows that do not have commentary or those that have a number of AST nodes that exceeds a certain threshold. For this dataset, extraction of commentary is required, as described in the previous section.

For both datasets, the source code is parsed into an Abstract Syntax Tree (AST) using the tree-sitter library. Out of this tree, an adjacency matrix is built and the type information is retained for each node. The source code and the type of AST nodes are tokenized into sequences of tokens, separated on whitespace. Each token is associated with a number.

4.4 Neural Network Model

The model takes inspiration from the paper "Improved Code Summarization via a Graph Neural Network", Leclair et al. [1], where it was used for imperative programming paradigms (Java code), achieving good results. The key differences between the model proposed by Leclair et al. [1] and this model are the structure of the AST nodes information, consisting of node types and the sharing of vocabulary between code, syntax types and commentary. Experimenting with adding information about the AST node types should help provide more technical context in the generated commentary, by providing insights into the inner workings of the language, such as variable binding. This would prove helpful in the case where the commentary would make reference to the same node types, when describing the behaviour of the code snippet. In this way, the AST becomes related to the commentary as well, not only to the code.

4.4.1 Inputs and Outputs

The model is built using an encoder-decoder architecture, commonly used pattern for sequence-to-sequence tasks such as code-to-comment generation. The model has four inputs:

- **Source Code Sequence:** The sequence of tokens representing the source code of the function. This is the primary input to the encoder part of the model.
- **AST Node Sequence:** The type of the nodes of the Abstract Syntax Tree, passed to the model.
- **AST Adjacency Matrix:** This is a classic adjacency matrix of a graph, where adj[i][j] = 1 if there is a directed edge between the i and j nodes, and false otherwise.
- **Comment Sequence / Decoder Input:** The sequence of tokens representing the comments generated up to that point in the model's decoding process. This is used as input to the decoder during training, allowing the model to generate comments based on the source code and AST

The output of the model is of type categorical, representing a token from the common vocabulary.

4.5 Convolutional Graph Neural Networks

The internal architecture of the model uses convolutional graph neural networks, in order to model the Abstract Syntax Tree (AST) structure information. ConvGNNs take graph data and learn representations of nodes based on the initial node vector and its neighbors in the graph. The process of combining the information from neighbouring nodes is called "aggregation." By aggregating information from neighbouring nodes, a model can learn representations based on arbitrary relationships [1].

4.6 Diagram

The architecture of the layers as proposed by Leclair et al. [1] is shown in the diagram below.

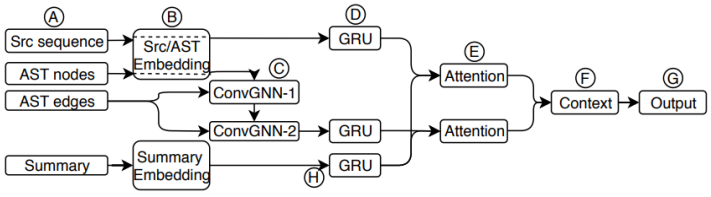

Figure 1: Model Architecture

4.7 General Architecture

The encoder processes the source code and AST simultaneously. Both the source code tokens and AST node tokens share a common embedding layer. The decoder generates the comment sequence based on the information provided by the encoder.

This model architecture builds on the dual-encoder approach proposed by Leclair et al. [1], which utilizes a GRU for source code sequences and a ConvGNN-based AST encoder. The architecture utilizes two attention mechanisms: (1) an attention between the decoder and the source code, and (2) between the decoder and the AST. The attention mechanisms are then concatenated together with the decoder to create a final context vector. Then, a dense layer is applied to each vector in the final context vector, which is then flattened and used to predict the next token in the sequence.

For more details, please refer to the original paper by [1].

5 Case Study on a Small Dataset

5.1 Environment

The platforms used for the experiments are Google Colab and Huggingface, as they provide access to increased RAM. A Python 3 Google Compute Engine backend (TPU) was used for the computations, providing 335 GB RAM (only at most 16 GB were used for training). These platforms impose time limitations for free usage, which impacts the quality and relevance of the experiments. The latest versions are the latest versions available on Google Colab.

To evaluate the outcome of the experiments, a combination of human evaluation and automated scoring was used, using BertScore [13].

5.2 First Experiment

GitHub Code Haskell Function: A small dataset of 10 entries is used for training. The training of the model lasts for approximately 15 minutes, and the dataset was trained for 5 epochs. In most cases, the model outputs only "¡unk¿" (unknown) tokens for code that was not used for training. For the snippets used in training, the model seems to generate random words taken from the commentaries.

5.3 Second Experiment

PHagenlocher-HaskellYTv0.1: A small dataset of 40 entries is used for training. The training of the model lasts for approximately 1 hour, and the dataset was trained for 5 epochs. The model seems to overfit on outputting sequences of "of" words, irrespective of input.

5.4 Third Experiment

PHagenlocher-HaskellYTv0.1: A small dataset of 40 entries is used for training. The training of the model lasts for approximately 20 minutes, and the dataset was trained for 3 epochs to avoid overfitting. Still, the model seems to overfit on outputting sequences of the same word, repeatedly, irrespective of input. However, now the word seems to be different for different code snippets. Sometimes, the word is an AST type.

5.5 Fourth Experiment

PHagenlocher-HaskellYTv0.1: For this experiment, the dimension of the internal embedding layers, GRU layer, and Time Distributed layer was cut in half to increase the speed. The training was done on a small dataset of 50 entries, for 2 epochs, lasting for approximately 13 minutes. This experiment seemed to be the most successful one, as the model is now outputting sentences, such as "this is is is is a function." However, the sentences are not necessarily correlated to the code.

6 Evaluation

For evaluating the performance of the best version of the model, BERTScore was used. BERTScore is a metric that measures the quality of text generation by comparing the similarity between generated and reference text using pre-trained BERT embeddings. It is particularly useful in evaluating models generating natural language text, as it considers semantic similarity between words rather than just exact matches.

In this case, BERTScore compared the generated comments with the reference comments in the dataset. A higher BERTScore indicates that the generated comments are closer in meaning to the original comments, reflecting a more accurate understanding and generation by the model.

The results are as follows:

- Mean BERT Precision: 0.7710
- Mean BERT Recall: 0.8165
- Mean BERT F1 Score: 0.7929

And the distribution of BERTScore can be visualised here:

This score is relatively high, given the small size of the dataset used for training. This can be explained by the fact that the commentary generated by the model consists of plenty of commonly used words (such as 'is', 'this', 'a', 'an'), which are likely to be present in the original commentary.

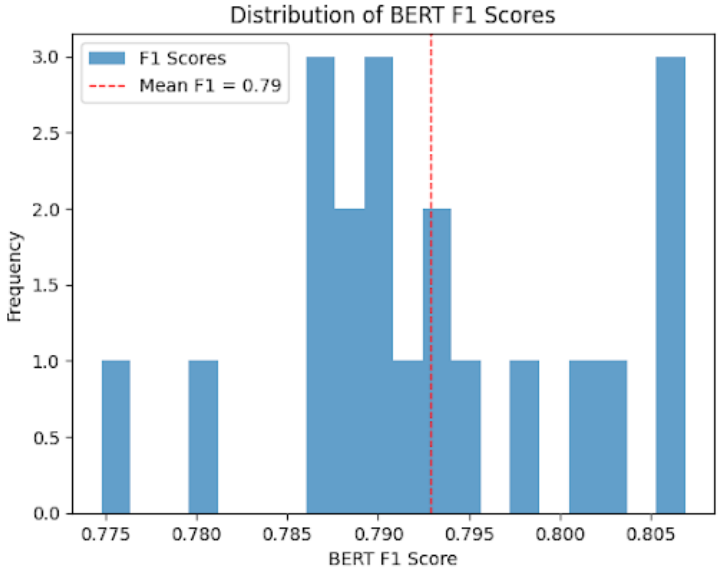

Figure 2: BERTScore Distribution

References

[1] Lingfei Wu Collin McMillan Alex LeClair, Sakib Haque. Improved code summarization via a graph neural network. In *2020 IEEE/ACM International Conference on Program Comprehension*, Oct. 2020.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[5] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[6] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[7] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[8] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[9] Gabor Ioana. https://github.com/IoanaGabor/code_analysis_project.

[10] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2019.

[11] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[12] Tim van Dam, Frank van der Heijden, Philippe de Bekker, Berend Nieuwschepen, Marc Otten, and Maliheh Izadi. Investigating the performance of language models for completing code in functional programming languages: a haskell case study. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 91–102, 2024.

[13] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.