Investigating the Performance of Language Models for Completing Code in Functional Programming Languages

A Haskell Case Study

van Dam, Tim; van der Heijden, Frank; de Bekker, Philippe; Nieuwschepen, Berend; Otten, Marc; Izadi, Maliheh

DOI
10.1145/3650105.3652289

Publication date
2024

Document Version
Final published version

Published in
FORGE '24

Investigating the Performance of Language Models for Completing Code in Functional Programming Languages: a Haskell Case Study

Tim van Dam
t.o.vandam@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Frank van der Heijden
f.n.m.vanderheijden@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Philippe de Bekker
p.m.debekker@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Berend Nieuwschepen
b.r.nieuwschepen@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Marc Otten
m.j.c.otten@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Maliheh Izadi
m.izadi@tudelft.nl
Delft University of Technology
Delft, Netherlands

ABSTRACT

Language model-based code completion models have quickly grown in use, helping thousands of developers write code in many different programming languages. However, research on code completion models typically focuses on imperative languages such as Python and JavaScript, which results in a lack of representation for functional programming languages. Consequently, these models often perform poorly on functional languages such as Haskell. To investigate whether this can be alleviated, we evaluate the performance of two language models for code, CodeGPT and UniXcoder, on the functional programming language Haskell. We fine-tune and evaluate the models on Haskell functions sourced from a publicly accessible Haskell dataset on HuggingFace. Additionally, we manually evaluate the models using our novel translated HumanEval dataset. Our automatic evaluation shows that knowledge of imperative programming languages in the pre-training of LLMs may not transfer well to functional languages, but that code completion on functional languages is feasible. Consequently, this shows the need for more high-quality Haskell datasets. A manual evaluation on HumanEval-Haskell indicates CodeGPT frequently generates empty predictions and extra comments, while UniXcoder more often produces incomplete or incorrect predictions. Finally, we release HumanEval-Haskell, along with the fine-tuned models and all code required to reproduce our experiments on GitHub [41].

KEYWORDS

Language Models, Automatic Code Completion, Line Completion, Programming Languages, Functional Programming, Haskell, CodeGPT, UniXcoder

ACM Reference Format:

Tim van Dam, Frank van der Heijden, Philippe de Bekker, Berend Nieuwschepen, Marc Otten, and Maliheh Izadi. 2024. Investigating the Performance of Language Models for Completing Code in Functional Programming Languages: a Haskell Case Study. In *AI Foundation Models and Software Engineering (FORGE '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650105.3652289

FORGE '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0609-7/24/04
https://doi.org/10.1145/3650105.3652289

1 INTRODUCTION

Automatic code completion has already proven to be an asset for programmers [18, 19, 27, 35, 38], with many Large Language Models (LLMs) accurately predicting developers' intent given sufficient expert context. GitHub Copilot has been very successful [28, 39], and case studies on automatic code completion such as IntelliCode Compose [38], Tabnine, and Amazon CodeWhispherer [45], prove that developers benefit from these completions.

Most of the training data for LLMs for code consists of imperative and Object-Oriented Programming (OOP) languages, with the most prominent programming languages being Python, JavaScript, and Java [21]. While such languages have embraced many functional programming concepts and features over the past years, they are still mainly object-oriented. In contrast to these popular languages, there is little research on code completion for strongly typed functional languages with advanced type class techniques. As a result, most datasets contain little to no functional code, leading to poor performance in functional languages such as Haskell. These languages are, however, still frequently used in practice, albeit not nearly as often as imperative languages. In this study, we aim to fill this gap by evaluating the performance of line completion in the Haskell programming language. Haskell in particular is interesting, as the syntax and language constructs are more concise and considered more difficult by many programmers. However, whether this also applies to language models has not been investigated.

We fine-tune two multilingual pre-trained code completion models, UniXcoder [15] and CodeGPT [25], using Haskell functions from the Blastwind dataset, a publicly accessible dataset containing Haskell function implementations[1]. We evaluate the models automatically using the aforementioned dataset and introduce a new dataset – a version of HumanEval [10] manually translated to Haskell – that we use to manually evaluate the models. For this, we translate 164 Python functions into their Haskell equivalent and introduce points of interest for our manual evaluation in the Haskell sources for the models to perform code completion on.

[1] https://huggingface.co/datasets/blastwind/github-code-haskell-function

Our results show that models fine-tuned on Blastwind outperform base models by a significant margin, indicating that knowledge of a broad array of imperative programming languages does not transfer well to functional languages. For our newly created HumanEval dataset translated into Haskell, the average performance, as measured by Exact Match (EM) and Edit Similarity (ES), indicates lower performance than on the Blastwind dataset. Our manual evaluation of the performance of the models on the HumanEval dataset indicates several key differentiators in the behavior of the fine-tuned CodeGPT and UniXcoder. CodeGPT often generates empty predictions and unnecessary comments, while UniXcoder has more `incomplete`, `wrong syntax`, or `undefined` predictions. This indicates that CodeGPT is more 'careful' when predicting, while UniXcoder prefers to predict incorrect output. Additionally, our manual evaluation shows no common pitfalls for Haskell code completion based on the distribution of annotations. Both models lack the performance to excel in any particular category, indicating a need for better datasets and models. In short, CodeGPT was found to be relatively reliable and accurate than UniXcoder, resulting in CodeGPT being the safer choice for practical usage. However, the manual analysis indicates that our fine-tuned models are not reliable enough to conclude that any particular aspect of functional programming is more difficult for code completion models.
To summarize, this paper highlights the potential for LLMs to comprehend functional concepts and languages, as shown by the performance increase resulting from fine-tuning. Additionally, the large performance gap between base models and fine-tuned models indicates that code completion models that are proficient in a wide array of imperative programming languages do not necessarily transfer this proficiency to functional languages. Based on the lack of performance in all annotated categories of the manual evaluation, no specific pitfalls are detected for Haskell code completion. Instead, the results indicate a high demand for better overall language support. This paper aims to accelerate this process by releasing a manually translated HumanEval dataset, originally written in Python, for Haskell, forming a great foundation for enhancing the performance of functional programming languages in further research.

The contributions of this paper are as follows:

- An extensive evaluation of the performance of pre-trained and fine-tuned code completion models on Haskell;
- A comprehensive assessment on the primary pitfalls of code completion when applied to Haskell;
- Publicly available manual translation of HumanEval to Haskell for future evaluation, including marked locations of interest for code completion;
- Publicly available source code and fine-tuned code completion models for Haskell.

2 MOTIVATION

Our work investigates the performance of LLM-based code completion, and in particular, line completion. We perform our evaluation using the Haskell programming language, as we find Haskell (and functional languages in general) to be underrepresented within the realm of Natural Language Processing (NLP) for code. Nearly all code completion models in the literature are trained on non-functional languages such as Python, Java, JavaScript, Go, and Ruby [13–16, 25, 38, 43]. Our primary aim is to investigate how well code completion models work on Haskell. Being a functional language, Haskell is syntactically and conceptually very different from the imperative programming languages that code completion models are typically trained on. Some languages such as Java and JavaScript contain functional paradigms, but this is not as close to a purely functional language such as Haskell. A notable difference, for instance, is the lack of control-flow statements such as `for` and `while` loops. Hence, to evaluate the usefulness of code completion models on functional languages, we fine-tune code completion models on Haskell and evaluate their performance relative to other programming languages. Concretely, we investigate code completion performance on a corpus of permissively licensed Haskell functions sourced from HuggingFace.[2] Additionally, we manually evaluate code completions on a newly created dataset from HumanEval containing 164 problems, which have been translated from the original Python source into Haskell function implementations for this purpose.

3 RELATED WORKS

This section will discuss the related works in the field of NLP with regard to the task of code completion. First, we will briefly discuss the model architecture, then we will look into some examples of models that are currently being used. Lastly, we discuss evaluations of models and empirical analysis results.

3.1 Transformer Architecture

LLMs for code completion take code as input and predict a segment of new code related to the input. This can be done using Causal Language Modeling in conjunction with specialized ways of masking inputs to facilitate auto-regressive generation or infilling. State-of-the-art models that are currently used for code completion use the transformer architecture [42]. By making use of attention layers, the transformer models allow for a better understanding of the semantic and syntactic structure of code. This is especially important in code, as code has very strict syntax, unlike natural language. There are three main distinctions in the transformer model:

3.1.1 Encoder-only models. This type of architecture is commonly used for masked token prediction and classification. At each stage, the attention layers can access all the words in the initial sentence (i.e. bi-directional attention). The BERT [12] family of models are implementations of this architecture. It has been studied widely, and has also been adapted for code [11, 13, 20, 26].

3.1.2 Decoder-only models. The decoder architecture is commonly used for code-completion and text generation. Decoder-only tokens have access to the weights of the tokens that come before itself (i.e. the left context). A family that implements this architecture is GPT [31]. Large types of these models have been shown to be effective for dialogue [8] and few-shot learning [24]. Newer variations in this family are GPT-2 [32], GPT-3 [6], and GPT-4 [30]. They have been widely used in the industry [29], and evaluated [7].

[2] https://huggingface.co/datasets/blastwind/github-code-haskell-function

3.1.3 Encoder-Decoder models. Encoder-Decoder models are a combination of the previous two architectures and have been popular in RNNs as "seq2seq", before the Transformer architecture [37], but even in recent years it has still shown to be viable and competitive [36]. The Encoder-Decoder architecture excels in machine translation, as the Encoder processes the source language and the Decoder processes this to the target language.

3.2 Code Completion

Automatic code completion can be used to predict single tokens, complete lines of code, or even complete functions. Recently there has been a surge in developers using better code completion tools, such as Copilot, which has shown improvements in efficiency by using the completions and then slightly tweaking them [27]. Most code completion models follow the decoder-only architecture and are trained using Unidirectional Language Modeling (ULM) on datasets containing natural language and code in many different programming languages. For instance, GPT-C [38] and CodeGPT [25] are code completion models based on GPT-2 [32] that use unidirectional language modeling to learn from examples. Similarly, Codex [10] is a GPT-3-based model [6] that can predict the next token(s) based on the current code. While most models can only use the *left context*, i.e., the source code that is to the left of the cursor, models like InCoder use Causal Masking to be able to handle code from the left and to the right of the cursor despite being auto-regressive [1, 14]. SantaCoder [2] and StarCoder [23] similarly are trained to *infill* code fragments between left and right contexts. As an alternative technique to improve performance, numerous models use modalities aside from source code, such as Abstract Syntax Trees (ASTs). CodeFill [18], for instance, uses source code along with AST token types to enhance the input, leading to improved performance. UniXcoder [15] similarly uses ASTs for a range of code generation and understanding tasks, however, does not benefit from ASTs when used for code completion. While code completion is a widely studied topic, surprisingly little research has been conducted on applying code completion to functional languages. A recent advancement has been made with datasets for OCaml, Racket and Lua [9], where high-quality datasets were made for those underrepresented languages in LLMs. However, almost none of the aforementioned code completion models use functional languages in their base-model training data, as shown in Table 1. This indicates a clear need for more awareness.

Only StarCoder has Haskell in its training data [23]. StarCoder uses the Stack dataset [22], which contains permissively licensed open-source code sourced from GitHub, including 358 unique programming languages. However, Haskell accounts for only 0.291% of StarCoder's training data, and no investigation into the performance of Haskell was performed.

4 APPROACH

Our approach can be divided into three distinct steps: *dataset creation*, *fine-tuning*, and *evaluation*. During dataset creation, we process and split our data for use by the selected models. Next, during fine-tuning, we train the models to perform line completion on Haskell code, using our train set. Finally, during evaluation, we run the fine-tuned models and base models against our test sets. An overview of the full approach pipeline is displayed in Figure 1.

Table 1: Code completion models and the languages they are trained on.

Model	Language(s)
GPT-C [38]	C#, Python, JavaScript, TypeScript
CodeGPT [25]	Python, Java
Codex [10]	Python
InCoder [14]	28 PLs, no significant amount of functional code
SantaCoder [2]	Python, Java, JavaScript
CodeFill	Python [18]
UniXcoder[15]	CodeSearchNet [17], containing Python, Ruby, Java, JavaScript, PHP, and Go
StarCoder	358 PLs, **0.291% Haskell**

4.1 Dataset Creation

During the dataset creation phase, we organize datasets for training and evaluation. We utilize two distinct datasets to achieve this. The Blastwind dataset, sourced from HuggingFace, consists of publicly available Haskell function implementations. This dataset serves as the foundation for the training phase. The second dataset, which we created ourselves, involves translating HumanEval from Python to Haskell. This self-created dataset is solely used for evaluation, which includes a manual evaluation where our objective is to identify the primary pitfalls to Haskell code completion.

4.1.1 Blastwind Dataset. This dataset contains Haskell function implementations. This dataset was selected as it was one of the very few available Haskell datasets and was large enough to be usable. An example input from this dataset is shown in Figure 2.

Before using this dataset, we apply various processing steps to remove low-quality samples. Initially, the dataset has 2 287 379 samples. First, we apply a basic filtering step to remove low-quality samples, ensuring sufficient context for the model. We use the following rules:

- The code must have a comment;
- The code must have a function signature;
- Parsing must not result in any AST errors;
- The code must be at least two lines of code (excluding comments);
- The code must be at least 75 characters long (excluding comments).

This step removes 83.78% of samples. This suggests a lot of the samples in this dataset were uninformative samples. Next, we deduplicate the remaining samples as duplicate samples can lead to data leakage between train and test sets. Such data leakage and consequently result in over-inflated performance scores [3]. Near duplicate deduplication is an expensive $O(n^2)$ operation, especially considering the size of this dataset. Hence, we are limited to performing exact-match deduplication. This is relatively efficient through the use of hash-sets, which allow for duplicate checking in amortized $O(1)$ time per sample. Applying near-duplicate deduplication requires tokenizing all samples and cross-comparing all tokens for

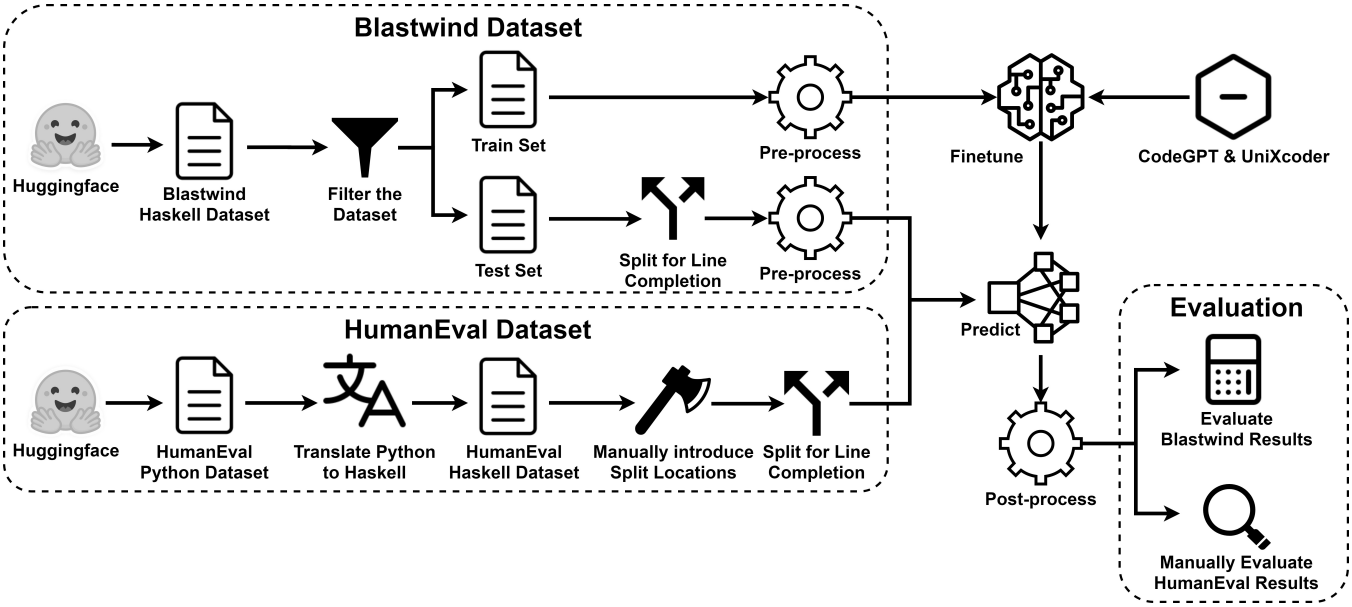

Figure 1: Approach pipeline.

```haskell
-- | Create a pair generator.
pairOf :: Applicative m => m a -> m (a, a)
pairOf m = (,) <$> m <*> m
```

Figure 2: Blastwind sample.

all sample pairs, which adds another layer of complexity of at least $O(n + m)$ time depending on the way the tokens are compared. Despite its limitations, exact-match deduplication removes 25.25% of the remaining samples.

Our final dataset contains 277 337 samples, which is 12.12% of the original dataset size. The processed dataset is subsequently split into a train and a test set using an 80%-20% split. During splitting, we ensure that functions from the same repositories are in the same set, which leads to a 72.81%-27.19% split when counting the number of functions per split, corresponding to 201 921 train samples and 75 416 test samples. This splitting approach serves to further prevent data leakage; a file-based split could lead to situations where a function from the train set is called in a function in the test set. The samples are still full function implementations at this point, so we must convert the test samples into input-output pairs to evaluate our models. Algorithm 1 describes the way these input-output pairs are created. In short, whitespace characters in the sample are candidate split-points if it is not in a comment and has sufficient preceding and succeeding tokens. We choose where to split by pseudo-randomly selecting from the candidate split-points. Note that this splitting method may not accurately capture the use of line completion in practice. One could argue that invoking code completion after specific *trigger-points* such as =, (, and . may be more realistic. However, it has been shown that even this strategy also does not strongly match programmer behavior in practice [19].

Algorithm 1 Generate Input-Output Pair

```
whitespaceIndices ← ∅
for character, index in code do
    if
        character is a whitespace ∧
        previous character is not a whitespace ∧
        has at least five preceding tokens ∧
        has at least one preceding token on the same line ∧
        has at least two following tokens on the same line ∧
        is not on a line starting with "--" ∧
        is not in a multi-line comment block
    then
        whitespaceIndices ← whitespaceIndices ∪ {index}
    end if
end for
chosenIndex ← pseudoRandomSelect(whitespaceIndices, seed)
Input ← code up to chosenIndex
Output ← code after chosenIndex up to first line-break
return (Input, Output)
```

4.1.2 HumanEval-Haskell Dataset. We manually create a new dataset by translating HumanEval[3] from Python to Haskell, a dataset that is often used for evaluating model performance. An example of a translated function is shown in Figure 3. The aim of this dataset is to be able to determine the most common mistakes by the selected models when applied to Haskell code. As Haskell is such a unique language, we expect the models to encounter different pitfalls than in imperative, non-functional languages. The translated dataset

[3]https://huggingface.co/datasets/openai_humaneval

```haskell
-- Check if in given list of numbers, are any two numbers closer to each other than
-- given threshold.
-- >>> has_close_elements [1.0, 2.0, 3.0] 0.5
-- False
-- >>> has_close_elements [1.0, 2.8, 3.0, 4.0, 5.0, 2.0] 0.3
-- True
has_close_elements :: [Float] -> Float -> Bool
has_close_elements numbers threshold = any (\(x, y) -> abs (x - y) < threshold) [(x,y) | x <- numbers, y <- numbers, x /= y]
```

Figure 3: HumanEval-Haskell sample.

contains code that follows an identical structure to the original Python code: the code consists of an instructive comment including test input and output, a function signature, and an implementation. While translating, the functional concepts of Haskell, such as pattern matching, monads, and working without side effects, are properly implemented. Additionally, any Python syntax embedded within the instructive comment is translated to Haskell syntax – e.g., `my_fn([1, 2, 3])` is translated to `my_fn [1,2,3]`. As our aim is to determine the most common pitfalls using line completion, we create equivalent Haskell functions from the one in the original HumanEval and run the new Haskell dataset on these. Each author translated the same number of HumanEval test cases, after which each translated Haskell function was reviewed by two different authors to resolve any issues.

Next, points of interest in the code are manually introduced and marked using a special symbol. We use the following points of interest, as they mark logical invocation points of code completion based on developer interest:

- statements such as `if`/`then`/`else`, generators and guards;
- assignment operators such as `=`, `<-` and `->`;
- logical operators such as `&&`, `||`, `==`, `>`, and more;
- arithmetic operators such as `/`, `*`, and more.

We never place these symbols at the very beginning or end of lines. Multiple splits can be introduced in a single Haskell function.

4.2 Fine-tuning

During the fine-tuning phase, the chosen code completion models are trained on the train set of the Blastwind dataset. The models are fine-tuned to perform line completion. All newline characters (`\n`) in the training data are replaced by end-of-line tokens (**<EOL>**). These tokens can be detected during inference and will cause the model to end its prediction loop.

4.2.1 Selected Models. We perform our experiments using two pre-trained decoder-only models for code completion.

First, *UniXcoder* [15] is a unified pre-trained model that can facilitate numerous code understanding and generation tasks, leveraging code comments and ASTs. UniXcoder was trained using Masked Language Modeling [5, 12] and Denoising [33] to facilitate code understanding, and trained using Unidirectional Language Modeling [31] for auto-regressive tasks such as code completion. UniXcoder was first trained on the C4 dataset [33] for English understanding, after which it was trained on CodeSearchNet [17], which includes function implementations (including comments) written in six programming languages: Python, Java, JavaScript, Go, PHP, and Ruby. While UniXcoder is able to also utilize an encoder-only or encoder-decoder mode, we have chosen specifically for decoder-only as this mode suits code completion best.

Second, *CodeGPT* [25] is a GPT-2-based [32] model for code completion. Lu et al. created four different versions of CodeGPT: for both Python and Java the authors create one version that uses the GPT-2 tokenizer and GPT-2 weights as a starting checkpoint, and one version that uses a newly trained tokenizer and starts with random weights. GPT-2 was trained on the WebText dataset, which Radford et al. introduced as a source of high-quality online text. For our experiments, we use a version of CodeGPT that uses the GPT-2 tokenizer and checkpoint and is further trained on CodeSearchNet [17].[4] This is similar to UniXcoder: both have been trained using a broad range of languages, which may be helpful when predicting new languages. Both models have been trained on imperative languages, which could transfer to improved performance in functional settings.

4.3 Evaluation

The evaluation phase applies the base models and fine-tuned models on the test set of Blastwind and on our newly created HumanEval-Haskell dataset.

4.3.1 Post-Processing. Before evaluation can begin the predictions need to be post processed. This is to ensure the evaluations are accurate and fair. The main steps in this process are stripping trailing spaces and newlines. Additionally the predictions are normalised so the exact match can accurately be calculated.

4.3.2 Blastwind Dataset. We evaluate the performance on the test set using the Exact Match and Levenshtein Edit Similarity metrics. These allow us to concretely compare our results with the results of the baseline models, which have results on other programming languages.

4.3.3 HumanEval-Haskell Dataset. We manually analyze the performance of our models. To prevent domination of the results by a few large functions with a lot of splits, a maximum of five splits per function are pseudo-randomly chosen from the manually introduced splits. Then, on a case-by-case basis, an overview of the performance is created with respect to each point of interest, such that common pitfalls can be detected (RQ2), e.g., the ability to predict list comprehensions. This leads to a total of 603 samples from the HumanEval-Haskell dataset.

4.3.4 Output. The models are applied on the test sets using *beam search*. Beam search is a sampling technique that can result in better predictions in exchange for higher inference costs. Instead of continuously sampling the most likely token (i.e., greedy prediction), beam search tracks the top-k sequences at every inference step. In the end, this results in k unique sequences, of which the one with the highest probability is used. Beam search is a common

[4] https://huggingface.co/AISE-TUDelft/CodeGPT-Multilingual

technique, that has also been widely used in seq2seq modelling [44]. Additionally, since beam search is deterministic we can be sure that our observed metric values are easily reproducible.

5 EXPERIMENTS SETUP

In this section, the research questions are presented, after which the datasets, evaluation settings and metrics, and configuration and implementation details are discussed.

5.1 Research Questions

In this work, we aim to answer the following Research Questions (RQs):

- **RQ1: How well do code completion models perform on Haskell?** This research question aims to answer whether there are quantitative differences in code completion performance between Haskell and imperative languages. Nearly all literature focuses on imperative, non-functional languages, so it is unclear whether Haskell (and functional languages in general) are more difficult for code completion models. As the chosen code completion models are pre-trained on a range of imperative languages, we also aim to gain an insight into whether this knowledge transfers well to Haskell.
- **RQ2: What are the most common pitfalls for code completion on Haskell?** Having considered quantitative differences in code completion, we now aim to understand what aspects of Haskell are difficult to code completion models. This understanding is crucial to improving code completion models.

5.2 Dataset

We use two datasets for our experiments. The first dataset, Blastwind, used for training, consists of permissively licensed Haskell function implementations. This dataset is publicly accessible on HuggingFace and contains a total of 3.26 million functions.[5] We create the second dataset ourselves by translating HumanEval [10] from Python to Haskell, leading to 164 Haskell functions based on HumanEval. This dataset is used for our manual evaluation, where we determine common mistakes made by the code completion models. None of the datasets overlap with the training data of the selected models, as these models were not trained on any Haskell code. Both datasets consist of Haskell function implementations, with the primary difference being that HumanEval always contains large informative comments, while this is rarely the case in real code, as found in Blastwind. This could have an effect on the efficacy of the models, as syntactical elements such as comments have been shown to have a positive impact on code completion performance [40].

5.3 Evaluation Setting and Metrics

We choose the same metrics that were used to evaluate UniXcoder and CodeGPT to compare code completion performance on Haskell to the performance on other programming languages [15, 25]. This allows us to directly compare with their results for Python and Java.

[5]https://huggingface.co/datasets/blastwind/github-code-haskell-function

Table 2: Fine-tuning times.

Model	Time
UniXcoder	19 hours
CodeGPT	12 hours

Table 3: Inference times.

Model	Variant	Blastwind	HumanEval-Haskell
UniXcoder	Base	20 h 26 min	13 min
	Fine-tuned	1 h 50 min	2 min
CodeGPT	Base	8 h 47 min	3 min
	Fine-tuned	4 h 35 min	4 min

The first metric, *exact match*, compares whether the prediction and the ground truth are the exact same. The second metric, *edit similarity*, uses the Levenshtein distance between the ground truth and the prediction to compute how close the prediction is to the ground truth. The distance is determined by the number of insertions, deletions, and substitutions required to make the target match the ground truth. Then, the edit similarity is computed as shown in Equation 1.

$$ES(p, g) = 1 - \frac{Levenshtein(p, g)}{max(|p|, |g|)} \quad (1)$$

The ground truth and prediction are trimmed (i.e., heading and trailing spaces are removed) and spacing is normalized (i.e., sequences of white-spaces of arbitrary length are replaced with a single space) before using them to compute metric values. We present the metric values on a scale of 0 to 100.

5.4 Configuration and Implementation Details

We fine-tune UniXcoder using the default parameters, which were also used by Guo et al. to fine-tune their model on Python code.[6] Similarly, we use the default parameters as used by Lu et al. for training CodeGPT.[7] For both models, we use a batch size of 2. For inference, we apply beam search with a beam size of three for both UniXcoder and CodeGPT. Additionally, we set the maximum number of tokens to predict to 128. This is sufficient for line completion.

Fine-tuning and inference were performed on a server equipped with an NVIDIA Tesla V100S GPU. The time required to complete fine-tuning is reported per model in Table 2. The large discrepancy between the two models is caused by differences in the ways in which the models organize their training inputs. CodeGPT batches as many inputs together during training, whereas UniXcoder uses padding tokens to fill any space left after an input. This leads to CodeGPT needing fewer steps in total, reducing its training time.

The inference time for all models on Blastwind and HumanEval-Haskell are displayed in Table 3. The base models nearly always have a substantially higher processing time than the fine-tuned

[6]https://github.com/microsoft/CodeBERT/tree/0b522a6d7b2e25456e52b1c99a8e9cc6cd2aa6e0/UniXcoder/downstream-tasks/code-completion

[7]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/CodeCompletion-line

Table 4: Blastwind & HumanEval-Haskell Results. EM and ES expressed as number between 0 and 100.

Model		Blastwind		HumanEval-Haskell	
		EM	ES	EM	ES
UniXcoder	Base	1.98	25.93	5.31	27.31
	Fine-tuned	28.00	56.90	13.10	44.16
CodeGPT	Base	2.05	19.51	5.80	23.17
	Fine-tuned	17.40	46.95	15.42	40.01

models, mainly due to the significantly longer predictions they produce. These models have a weak understanding of Haskell, causing them to struggle with identifying where to end lines. As a result, the inference times are increased due to the cost incurred by each generated token. The difference is especially pronounced between the base and fine-tuned UniXcoder models. This is to the fact that the base UniXcoder model was not trained to predict and stop at **<EOL>** tokens at the end of each line. We post-process its outputs such that we only consider the first line it predicts, which ensures that this does not impact our results. However, this does lead to higher inference times for the base model.

6 RESULTS

We conduct a quantitative and qualitative analysis to determine how well UniXcoder and CodeGPT perform on Haskell.

6.1 RQ1: How well do code completion models perform on Haskell?

We quantitatively evaluate the performance on Blastwind and HumanEval-Haskell by running all base and fine-tuned models against the datasets. Then, we compute the average EM and ES scores over all test samples. The results are shown in Table 4. Overall, the base models (i.e., (pre-)trained on six programming languages, not including Haskell) show significantly worse performance than the fine-tuned models. On Blastwind, the two base models are roughly on-par, but the fine-tuned UniXcoder performs much better than the fine-tuned CodeGPT model across metrics. There is not such a clear distinction on HumanEval-Haskell: CodeGPT scores better on Exact Match, whilst UniXcoder scores better on Edit Similarity. Performance on Blastwind is substantially better than performance on HumanEval-Haskell, especially when considering Exact Match. Nevertheless, there is a large overall improvement in model performance when the models are fine-tuned.

6.2 RQ2: What are the most common pitfalls for code completion on Haskell?

We conduct a manual qualitative analysis on HumanEval-Haskell, to further analyse why UniXcoder outscores CodeGPT and the common pitfalls both models have with Haskell linecompletion. Note, only the *fine-tuned* variants of the models are considered in this context. After splitting the data as described in Section 4.3.3, we obtained insights into the common pitfalls by annotating the performance of the splits based on several (sub)categories. By manual inspection of all the predictions, we also found some predictions to be 'valid', viz. semantically equal, resulting in an updated performance overview as illustrated in Table 5.

Table 5: Updated performance of fine-tuned models by manual inspection, where some non-EM predictions have been marked as 'valid'. The correct ratio of predictions (%), denoted by the '%' header, is calculated by dividing the total count of predictions with the *|EM| + Valid* count.

Model	|EM|	Valid	|EM| + Valid	Total	%
UniXcoder	79	18	97	603	16.09
CodeGPT	93	20	113	603	18.74

When looking at the distribution of annotations for both CodeGPT and UniXcoder (see Appendix C [41]), there is no substantial dissimilarity in their general performance with regard to the prediction of certain Haskell (sub)categories such as if/then/else statements, generators, guards, functions, lists, logical operators, arithmetic operators, and case expressions. However, a clear difference in their general performance is shown by the distribution of the other annotations listed in Table 6. This table shows that CodeGPT has significantly more `empty` predictions when compared to UniXcoder, however, UniXcoder has more `incomplete`, `wrong syntax`, and `undefined` as predictions. Furthermore, a few specific predictions of UniXcoder have been separately marked as worth mentioning. This includes:

- using a variable out of scope, mentioned in a previous function within the provided context
- using a variable out of scope, mentioned in a comment within the provided context, yet not defined within the Haskell let context
- line completion of max characters due to getting stuck in a repetitive loop of predicting values for a list

Table 6: Distribution of distinctive annotations between CodeGPT and UniXcoder for predictions that were neither an exact match nor deemed valid.

Annotation	CodeGPT	UniXcoder
Wrong type	12	26
Wrong value	69	66
Wrong function	108	81
Empty prediction	106	1
Incomplete prediction	30	130
Wrong syntax	13	96
'undefined' keyword	1	31

As there seemed to be a correlated overlap in certain annotations for each specific prediction done by UniXcoder and CodeGPT, the commonalities in annotations have also been researched (see Appendix A[9]) in order to get more insight into the different behavior of UniXcoder and CodeGPT. In addition, the commonalities between CodeGPT's and UniXcoder's annotations per prediction itself, hereafter referred to as *overlaps*, have been researched to get more

context for the identification of common pitfalls. The most insightful annotation links that have been discovered, e.g. high overlap or similar annotation type, are illustrated in Table 7. One annotation in particular, i.e., `extra comment`, has been analyzed separately in Appendix B[10], as CodeGPT adds an extra comment to a lot of its predictions, which was found to often be of similar syntax. There is no clear pattern to where CodeGPT adds these comments , however, its content was always in the following format: "*| Creates a value of '<some class name starting with ProjectsLocations>' with the minimum fields required to make a request.*".

Table 7: Most insightful overlaps between annotations of predictions. CodeGPT is denoted by `C` and UniXcoder by `U`.

Annotation	Annotation	Overlap
C: empty	U: incomplete	54.72% (58/106)
C: empty	U: undefined	14.15% (15/106)
C: incomplete	U: incomplete	43.33% (13/30)
U: undefined	C: empty	48.39% (15/31)
U: incomplete	C: empty	44.62% (58/130)
C: complete function	C: wrong function	75.00% (9/12)
C: variable definition	C: wrong value	28.57% (8/28)
C: valid	C: extra comment	15.04% (17/113)
C: arithmetic logic	C: wrong value	14.29% (12/84)
U: undefined	U: case expr. (body)	67.74% (21/31)
U: incomplete	U: case expr. (body)	58.46% (76/130)
U: wrong type	U: case expr. (body)	38.46% (10/26)
U: wrong function	U: case expr. (body)	38.27% (31/81)
U: variable definition	U: incomplete	31.58% (6/19
U: arithmetic logic	U: wrong value	17.24% (15/87)

7 DISCUSSION

This section discusses the results and what they imply. Also the validity and what conclusions can be drawn from the results.

7.1 RQ1: How well do code completion models perform on Haskell?

The performance of UniXcoder and CodeGPT on Haskell improved drastically after fine-tuning. When considering the Blastwind dataset, our fine-tuned models exhibit worse performance compared to the models when fine-tuned on Python and Java [15, 25], as displayed in Table 8. The relative improvements to the base models affirm that language models can become sufficient in Haskell, despite its stark differences to typical programming languages. The poor performance relative to Python and Java could indicate differences in difficulty between the languages, but may also be explained by differences in dataset size and quality. Our Haskell train set consists of 30 349 824 tokens, whilst PY150 [34] contains 154 241 924 tokens for training, and JavaCorpus [4] contains 24 029 629 tokens for training.[8] Based on the number of tokens used for training and the observed metric values, we believe that Haskell is in fact more challenging than Python and Java overall. Nevertheless, further investigations with larger high-quality Haskell training sets (both in terms of tokens and in terms of prompt length) are necessary to validate these claims.

Table 8: PY150 and JavaCorpus results.

Dataset	Model	EM	ES
PY150	UniXcoder	43.12	72.00
	CodeGPT	39.11	69.69
JavaCorpus	UniXcoder	32.90	65.78
	CodeGPT	25.30	61.54

The results for the HumanEval-Haskell dataset are different from the results on the Blastwind dataset. Overall, the base models score better on this dataset, whereas both fine-tuned models score substantially lower. The superior performance of the base models could be explained by the format and context of HumanEval-Haskell. First off, each sample contains a relatively large informative comment. As both UniXcoder and CodeGPT are pre-trained on a large corpus of natural language, this naturally enhances their ability to infer the appropriate following tokens. In contrast, the Blastwind dataset contains smaller comments, leaving out details that are unimportant or obvious to humans. Secondly, samples in the HumanEval-Haskell dataset are far more self-contained than the samples in the Blastwind dataset. Samples in HumanEval-Haskell are typically single functions, but may also have numerous helper functions located in the same file. Functions in the Blastwind dataset are not guaranteed to be self-contained. The samples in this dataset are solely individual function implementations, without including any external helper functions that the model may need. This subsequently makes it more difficult for the models to accurately predict the following code: helper functions, despite being in scope, are not known to the model. The fine-tuned models may suffer from this to a lesser extent due to the training data resembling the test set more.

Despite the HumanEval-Haskell dataset including more informative comments, the fine-tuned models perform better on the Blastwind dataset. This could similarly be explained by the dissimilarity between the data in the two datasets: the fine-tuned models were trained on the Blastwind dataset, and therefore are trained to use little additional data when predicting Haskell. While the models have natural language understanding, they have not been trained to relate natural language to Haskell code. This could explain the difficulty in using the information embedded in the informative comments. Training approaches data that combines natural language with code, such as commented functions, or Q&A-style conversations about code (as seen on StackOverflow) could alleviate this.

Overall, for both datasets the fine-tuned models outperform the base models by a substantial margin, indicating that fine-tuning code completion models on Haskell is crucial to achieving optimal performance. Our results indicate that Haskell is harder to predict than Python and Java, but further experiments are required to verify this.

[8] All tokens counts are using the UniXcoder tokenizer

7.2 RQ2: What are the most common pitfalls for code completion on Haskell?

The distribution of the annotations for the HumanEval-Haskell predictions in combination with the general performance indicates a difference in the capabilities of the models. The distinctive factor in the capabilities can be explained by their differences in behavior. While CodeGPT is more cautious as evidenced by often predicting empty lines, UniXcoder tends to not only predict `undefined` in such cases but also shows a more aggressive prediction behavior. This behavior of UniXcoder leads to incomplete predictions (also often wrong functions in such cases) that would require a new line to complete in order to successfully continue. Furthermore, UniXcoder spits out a troublesome number of predictions featuring fundamental issues such as wrong syntax (e.g., mismatched brackets or capitalized function names), scope issues, or getting stuck in repetition. Ultimately, this makes CodeGPT a safer choice for practical usage, purely based on the behavior of the models. In addition, CodeGPT demonstrated to be the better choice after manually evaluating the correctness of HumanEval-Haskell predictions. It is worth mentioning that splits are introduced manually on points of interest (based on developer experience) for the HumanEval-Haskell dataset, while for the Blastwind dataset, the splits are introduced pseudo-randomly. This variation in the method might well influence the resulting predictions, as the manually introduced splits could be above averagely complex for the models to predict. The exact influence of this variation remains unclear, but it should be noted that the results of this deviation are included in the EM and ES values. Regarding the common pitfalls of Haskell code completion, none of the annotated categories in the manual evaluation yield a significant performance disparity. In fact, the overall performance indicates a high demand for improved support to excel in any particular category, which should be the primary focus of future research.

7.3 Implications

The results of our process of fine-tuning LLMs such as CodeGPT and UniXcoder on Haskell datasets have several theoretical and practical implications for the field of functional programming and code-LLMs.

7.3.1 Theoretical Implications. Fine-tuning LLMs on Haskell has shown to be effective for these models to grasp functional programming concepts. Furthermore, the nuanced trade-off between empty and faulty completions during manual evaluations underlines the complexity of optimal decision-making in AI, suggesting the need for more advanced metrics in model performance assessment. Additionally, the fact that multilingual LLMs underperform significantly compared to their fine-tuned counterparts shows the need for diverse high-quality Haskell datasets in the pre-training of these LLMs. Adding a deep understanding of functional programming might elevate the predictions for OOP-based languages as well, but further research is needed to accurately determine the effect of functional programming pre-training on OOP languages. These datasets could then be included in the pre-training of LLMs for a thorough understanding of this functional programming language.

7.3.2 Practical Implications. Practically, the fine-tuned LLMs on Haskell show to be promising for sophisticated developer tools in functional programming, for instance for code search, code repair, or code summarization. These models could serve as a helper tool for developers new to the complex world of functional programming. For instance, insights gained from AI interactions with Haskell might inspire new language features or paradigms. Furthermore, these models could enable more advanced code suggestion and debugging features, significantly reducing the time and effort required for developers to write and maintain Haskell code. Training LLMs on Haskell can also be beneficial for a deeper understanding of OOP languages that implement functional programming constructs, such as higher-order functions, the notion of pure functions, and recursion.

7.4 Future Work

Future work could strengthen our findings by repeating our experiments on other models, other functional languages, or other datasets. Constructing new high-quality Haskell datasets would be particularly interesting for future LM-based tools for Haskell. At present, there are no high-quality curated Haskell datasets, which can lead to sub-par models resulting from issues such as duplication, small dataset size, and uninformative samples. The fact that our filtering process eliminated nearly 90% of samples further emphasizes the necessity of high-quality Haskell datasets. Moreover, the significant number of discarded samples suggests that conducting experiments with larger datasets could yield valuable insights. Alternatively, online data from e.g., StackOverflow could be used to create an understanding of natural language when related to Haskell code. Other research could also investigate the effect of including Haskell datasets in its pre-training steps on mainstream languages used for evaluating LLMs.

Furthermore, it is worth considering alternative training inputs. Our chosen dataset only includes one function implementation per sample. However, utilizing complete files as training inputs may yield more precise results. Function implementation often depends on contextual information located in the same file. Hence, training on full files instead of single-function implementations would also make our training data align more closely with real-life scenarios.

Finally, determining whether understanding of different programming languages transfers to Haskell would provide an interesting insight. In this study, the pre-trained models were trained on six diverse programming languages before being fine-tuned for predicting Haskell code. The underlying assumption was that this wide-ranging knowledge base would yield better results in Haskell. However, determining whether this approach was indeed beneficial is beyond the scope of this paper. Hence, future research could explore whether this assumption is correct by comparing our findings with results from models that were solely trained on natural language or a single programming language.

7.5 Threats to the Validity

Due to the experimental method of this paper, the validity of the results in the context of the 'real world' must also be considered. These threads can be divided into three categories: threats to internal validity, external validity, and construct validity.

7.5.1 Internal Validity. This section discusses the elements that impact the model's performance, external factors that accidentally affect the outcomes, and mistakes made during the implementation process. Translation quality in the HumanEval-Haskell dataset is important because translation errors could lead to incorrect measurements of performance, compromising our ability to gauge which aspects of Haskell are most difficult to language models. Additionally, any data leakage between train and test sets must be mitigated to prevent unrealistic perceptions of performance.

We have considered these factors and implemented steps to minimize or mitigate the effects of these threats. For instance, the translated HumanEval functions were reviewed by numerous authors, and the Blastwind dataset was deduplicated to promote diversity in the training data as well as to prevent leakage between train and test sets. Moreover, we split the Blastwind dataset into train and test sets on a repository basis, meaning that all code from the same repository will be in either the train or the test set, but never both. This method further serves to prevent data leakage. To allow further examination regarding these concerns, we have made all resources, including datasets and models, publicly available for scrutiny and further research.

7.5.2 External Validity. This section discusses the elements that may influence the applicability or broader relevance of our research results. The usage of line completion in a realistic programming environment may differ from the experimental setup designed to answer *RQ1*. We introduce splits in pseudo-random locations, given that there is a sufficient number of surrounding tokens. A programmer using the line completion functionality would likely invoke code completion in different places, such as trigger points. Examples of trigger points are property accesses on variables (e.g., a `.`, or `->`), or for example an opening bracket (`(`). This limitation is, however, partly mitigated in *RQ2*, in which we manually introduce the splits in logical places. To prevent skewed results, the datasets are deduplicated. There is no overlap in pre-train data and test data, as the pre-train data for both UniXcoder and CodeGPT do not include Haskell code [15, 17, 25].

In the case of HumanEval-Haskell, the model input includes a detailed comment on the functionality of the desired function and example input and output. In reality, however, this context will often not be available for the model, which means that the prediction has to be made using less context, resulting in worse performance.

Furthermore, the setting of this evaluation is relatively pure in HumanEval-Haskell, each time the full comment is available, no surrounding other functions (which could influence the model's predictions), and it only has to predict a single line. In a real-world setting, such as code completions within a developer's project in an IDE, a lot more context is presented which might distract the model from giving the proper completions.

Additionally, It is essential that the Blastwind dataset broadly represents Haskell code patterns to prevent training biases. Insufficient diversity can lead to a model performing poorly on real-world Haskell problems.

7.5.3 Construct Validity. This section discusses the validity of the measurements performed. Metrics must accurately reflect the model's ability to generate functional Haskell code – misaligned evaluation can invalidate the perceived effectiveness of the fine-tuning process. As described in Section 5.3, the metrics used are commonly used in literature [15, 25, 38]. While it is known that these metrics do not capture all nuances of code, such as semantics, they are still widely used and thus essential to be able to compare models with one another. Hence, the chosen combination of metrics ensures a sufficient evaluation of the output of the model. Additionally, small differences in interpretation of the metrics can result in different conclusions. It is therefore essential to properly elaborate on the usage and interpretation of the metrics, as done in Section 5.3 and Section 6. The same holds for the process of data processing, for which a detailed description is given in Section 4.

8 CONCLUSION

LLMs for code completion are often trained only on imperative or Object-Oriented Programming languages. As functional languages are severely underrepresented in training data, the performance of code completion on these languages is substantially worse than on other languages. In this work, we explore the performance of two multilingual auto-regressive language models, UniXcoder and CodeGPT, when tasked to perform line completion on Haskell function implementations. Results show that both models perform significantly better after being fine-tuned, indicating that knowledge of imperative languages does not necessarily transfer to functional languages. Base models perform considerably worse, suggesting that prior knowledge of imperative programming languages may not transfer well to functional languages, and indicating a need for future datasets to include high-quality Haskell code. Additionally, our manual analysis shows that CodeGPT tends to generate more empty predictions and unnecessary comments, while UniXcoder generates incomplete, wrong syntax, and 'undefined' predictions. Since the behavior of UniXcoder shows more fundamental completion issues than CodeGPT during manual evaluation, CodeGPT is a safer choice for practical usage. Regarding the primary pitfalls of Haskell code completion, no specific categories annotated in the manual evaluation indicate a substantial difference in performance such that a general statement can be made about pivotal focus areas for the improvement of Haskell code completion – the language in general requires more support to perform well in any category as of yet. Lastly, the community should take into consideration the implementation of FP languages when training LLMs, especially since many modern OOP languages are integrating more functional concepts as time goes on. Giving LLMs a wider understanding of these fundamental concepts may help models understand such concepts across many different languages.

9 DATA AVAILABILITY

To ensure reproducibility and replicability, we publish all data used for our training and evaluation, including all source code used to conduct our evaluation and analysis. We publish our manually translated Haskell-HumanEval dataset, and additionally provide a supplemental detailing overlapping annotations between UniXcoder and CodeGPT, and highlighting the extra comments predicted by CodeGPT.

REFERENCES

[1] Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, et al. 2022. Cm3: A causal masked multimodal model of the internet. *arXiv preprint arXiv:2201.07520* (2022).

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars!. In *Deep Learning for Code Workshop (DL4C)*.

[3] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[4] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 207–216.

[5] Alexei Baevski, Sergey Edunov, Yinhan Liu, Luke Zettlemoyer, and Michael Auli. 2019. Cloze-driven Pretraining of Self-attention Networks. *CoRR* abs/1903.07785 (2019). arXiv:1903.07785 http://arxiv.org/abs/1903.07785

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

[7] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]

[8] Paweł Budzianowski and Ivan Vulić. 2019. Hello, It's GPT-2 – How Can I Help You? Towards the Use of Pretrained Language Models for Task-Oriented Dialogue Systems. arXiv:1907.05774 [cs.CL]

[9] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. arXiv:2308.09895 [cs.PL]

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3128234

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL

[15] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

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

[18] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-Token Code Completion by Jointly Learning from Structure and Naming Sequences. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 401–412. https://doi.org/10.1145/3510003.3510172

[19] Maliheh Izadi, Jonathan Katzy, Tim van Dam, Marc Otten, Razvan Mihai Popescu, and Arie van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In *Proceedings of the 46th International Conference on Software Engineering*.

[20] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1332–1336. https://doi.org/10.1109/ASE51524.2021.9678927

[21] Jonathan Katzy, Maliheh Izadi, and Arie van Deursen. 2023. On the Impact of Language Selection for Training and Evaluating Programming Language Models. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 271–276.

[22] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[24] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. GPT Understands, Too. arXiv:2103.10385 [cs.CL]

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=6lE4dQXaUcb

[26] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 505–509. https://doi.org/10.1109/MSR52588.2021.00063

[27] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023). https://doi.org/10.1016/j.jss.2023.111734 Cited by: 2; All Open Access, Green Open Access.

[28] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.

[29] OpenAI. [n. d.]. Introducing chatgpt. https://openai.com/blog/chatgpt

[30] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[31] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[32] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21, 1, Article 140 (jan 2020), 67 pages.

[34] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51, 10 (2016), 731–747.

[35] Agnia Sergeyuk, Sergey Titov, and Maliheh Izadi. 2024. In-IDE Human-AI Experience in the Era of Large Language Models; A Literature Review. In *First Workshop on IDEs*.

[36] Saleh Soltan, Shankar Ananthakrishnan, Jack G. M. FitzGerald, Rahul Gupta, Wael Hamza, Haidar Khan, Charith Peris, Stephen Rawls, Andy Rosenbaum, Anna Rumshisky, Chandana Satya Prakash, Mukund Sridhar, Fabian Triefenbach, Apurv Verma, Gokhan Tur, and Prem Natarajan. 2022. AlexaTM 20B: Few-shot learning using a large-scale multilingual seq2seq model. *arXiv* (2022). https://www.amazon.science/publications/alexatm-20b-few-shot-learning-using-a-large-scale-multilingual-seq2seq-model

[37] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. arXiv:1409.3215 [cs.CL]

[38] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[39] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[40] T. van Dam, M. Izadi, and A. van Deursen. 2023. Enriching Source Code with Contextual Data for Code Completion Models: An Empirical Study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 170–182. https://doi.org/10.1109/MSR59073.2023.00035

[41] Tim van Dam, Frank van der Heijden, Philippe de Bekker, Berend Nieuwschepen, and Maliheh Izadi. [n. d.]. Study Material Including Source Code and Data. https://github.com/AISE-TUDelft/HaskellCCEval [Accessed: 2024].

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[43] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 14015–14023.

[44] Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-Sequence Learning as Beam-Search Optimization. arXiv:1606.02960 [cs.CL]

[45] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023).