5-2024

Evaluating Large Language Model Performance on Haskell

Andrew Chen
William & Mary

Evaluating Large Language Model Performance on Haskell

Andrew Chen

Williamsburg, Virginia, USA

Bachelor of Arts, College of William and Mary, 2024

An Honors Thesis presented to the Faculty
of The College of William & Mary

Department of Computer Science

The College of William & Mary
May 2024

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Arts

Andrew Chen

Approved by the Committee, May 2024

Committee Chair
Professor Denys Poshyvanyk, Computer Science
The College of William & Mary

Pradeep Kumar

Assistant Professor Pradeep Kumar, Computer Science
The College of William & Mary

Associate Professor Guan Nan Wang, Mathematics
The College of William & Mary

ABSTRACT

I introduce HaskellEval, a Haskell evaluation benchmark for Large Language Models. HaskellEval's curation leverages a novel synthetic generation framework, streamlining the process of dataset curation by minimizing manual intervention. The core of this research is an extensive analysis of the trustworthiness of synthetic generations, ensuring accuracy, realism, and diversity. Additional, I provide a comprehensive evaluation of existing open-source models on HaskellEval.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I am grateful for my advisor, Professor Poshyvanyk. He picked out my thesis topic specifically for my interest in Haskell. Without him, I would have no journey to embark on. I am indebted to Alejandro Velasco, who despite is in perhaps the busiest time in his life doing his PhD, has offered me a tremendous amount of support and insight. Thank you to the rest of my defense committee. Professor Wang and Professor Kumar and two of the best teachers I have ever encountered. Thank you to all other members of the SEMERU group. You all have influenced me in the best manner possible.

Dedicated to my mother, who sacrificed unconditionally despite already bestowing upon me the greatest gift.

您无私地奉献

尽管

您已赐予了我

最珍贵的礼物

LIST OF TABLES

LIST OF FIGURES

Evaluating Large Language Model Performance on Haskell

Chapter 1

Introduction

> It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use, and in this respect programming languages have a devious influence: they shape our thinking habits. *Edsger W. Dijkstra*

The recent surge in Large Language Models (LLMs) capabilities marks a pivotal moment in the trajectory of artificial intelligence research. This exponential growth can be attributed to a confluence of factors, including advancements in deep learning architectures, the availability of massive datasets, and the parallel increase in computational power. The breakthrough achieved by models like GPT-3 [1], released in 2020, demonstrated the transformative potential of LLMs across various domains.

Amongest these domains is software engineering (SE) [2] [3]. This capability has undergone significant evolution since the initial models. For instance, Codex-12B, released in 2021, achieved a pass@1 rate of 28.81% on HumanEval [4]. Today, Claude 3 Opus [5] boasts an impressive 84.9% pass@1 rate. With such tremendous progress, researchers have began to transition into testing model capabilities on harder SE tasks, e.g., using agent-based models to resolve real GitHub issues [6].

However, code models always focused on the popular programming languages (PL), with Python at the pinnacle. Many Python LLMs are fine-tuned from multi-PL models with massive amounts of Python tokens.

Further, most evaluation datasets are Python specific. Hence, We do not understand the performance of LLMs on the less popular languages such as Haskell. In fact, due to its lack of popularity, the entire paradigm of functional programming (FP) languages is neglected. This oversight is regrettable. FP languages like Haskell are often purer and have cleaner syntax, which are attributes beneficial to model training. In light of these observations, this thesis lays the foudnation for evaluating LLMs on Haskell.

1.1 Contributions

Chapter 3 presents the curation process of HaskellEval. HaskellEval is a Haskell evaluation benchmark curated synthetically and then valiadated manually. The bulk of Chapter 3 explores whether these synthetic generations are trustworthy. Chapter 4 evaluates open-source LLMs on HaskellEval, discussing the impact of model configurations and inference parameters.

Chapter 2

Background & Related Work

In this chapter, I provide the essential background on Large Language Model (LLM) and Haskell.

2.1 LLM

2.1.1 Before LLM

Before the advent of large language models, there existed statistical language modeling (SLM). SLM primarily involved simpler models that learned word sequence probabilities without deep contextual understanding. The transition to Pre-trained Language Models (PLMs) marked a significant shift, introducing models like BERT [7] and GPT [8], which leverages vast amounts of data and the transformer architecture [9] to achieve deeper language comprehension and generation. This foundation set the stage for scaling up these models, leading to the development of LLMs.

2.1.2 LLMs: Datasets, Training, and Tuning

The development of LLMs begins with data collection and cleaning, ensuring the mass, quality, and diversity of the training corpora. These corpora are compiled from a variety of sources, including books, websites, and other texts to capture a broad spectrum of human knowledge. Popular corpora include BookCorpus [10], CommonCrawl [11], (a petabyte in volume), Wikipedia, and GitHub [12].

Then, LLMs pre-train on these extensive corpora, which imbues the models with the wide range of knowledge and world-view the corpora offer. Afterwards, LLMs undergo instruction tuning, which involves fine-tuning the models on datasets specifically designed to enhance their ability to follow natural language instructions. The datasets used for instruction tuning often consist of tasks described in natural language paired with appropriate responses.

Recently, synthetically curated instruction datasets has been on the rise due to the otherwise extensive cost of human annotation. Most notably, MagiCoder [13] and WizardCoder [14] improved upon the state-of-the-art open source code models by instruction-tuning on synthetic instructions. MagiCoder's OSS-Instruct method asks GPT-3.5 to create a total of 75K pairs of code problems and solutions. Each problem utilizes a real-world code snippet as inspiration. The seed has less than 15 lines and is sourced from *bigcode/starcoderdata*, which contains 786G of code from 86 different programming languages. WizardCoder's Evol-Instruct method [15] evolves the Code Alpaca dataset [16] using phrases such as "add new constraints" and "increase problem difficulty", resulting in 78K samples. Code Alpaca is synthetically curated as well. It uses the Self-Instruct method [17] and has 20K samples.

At last, LLMs are refined to align with human values. Models are adjusted using reinforcement learning techniques sourcing feedback from human evaluators.

2.1.3 Capabilities and Evaluation

Large Language Models (LLMs) have demonstrated unexpected, *emergent* abilities. Along with its broad knowledge, these abilities allow LLMs to serve as general purpose chat bots.

In-Context Learning LLMs starting from GPT-3 exhibit the ability to understand and respond to tasks based on context provided within the prompt itself, a phenomenon known as in-context learning (ICL) [1]. This capability allows the models to generate appropriate responses or complete tasks by leveraging context clues without additional training. This form of learning highlights the model's ability to adapt to new tasks using minimal examples, showcasing an advanced level of comprehension and flexibility. The field of mechanistic interpretability has offered evidence for how a simple form of ICL may exist in a toy, two-layered, and attention-only transformer [18]: Induction circuits form during training, which increases the logits for predicting token `B` after the token `A`, if the sequence previously contains a `B` after `A`. I.e., for sequence `[Harry][Potter],...,[Harry]`, the induction circuit increases the probability of predicting `[Potter]` as the next token of the second `[Harry]`.

Instruction-Following Unlocked by instruction tuning, LLMs are capable of following complex prompts. This capability ensures that the models can understand and execute a wide array of tasks described by users. The ability to follow instructions is pivotal for integrating LLMs into interactive applications where user guidance dictates model responses.

Step-by-Step Reasoning Math problems and code debugging require complicated reasoning steps. By training on these steps, models have obtained to ability to perform step-by-step reasoning. This is an emergent capability that must be "elicited". I.e, the model only reasons step-by-step if it is prompted with "think step by step" [19].

The robustness of LLM reasoning capabilities and knowledge are tested across a wide spectrum of domains and tasks through specialized benchmarks. Benchmarks such as MMLU [20] and BIG-bench [21] assess general knowledge and reasoning, while datasets like MultiMedQA [22] and LegalBench [23] focus on specialized domains like healthcare and legal. Benchmarks like Chatbot Arena [24] and SciBench [25] evaluate human alignment and complex reasoning capabilities. As of April 2024, the top-5 performing models on

Chatbox Arena are all closed-source. At rank 27, WizardLM-70B-v1.0 [15] is the best performing open-source model on Arena.

2.2 Evaluating LLM Coding Capabilities

LLMs contain a myriad of coding capabilities, each of which can be framed as a SE (software engineering) task. The most important SE task is program synthesis, one of the holy grails of computer science. Program synthesis involves generating a program from a natural language specification. Most code benchmarks, including HaskellEval, are program synthesis benchmarks. Each row in these benchmarks includes a problem description in natural language and test cases that the solution is expected to pass. During evaluation, the model generated solution is executed against the test suite. If it passes all the test cases, the model has generated functionally correct code.

2.2.1 Code Benchmarks

Popular benchmarks include HumanEval [4], APPS [26], MBPP [27], CodeContest [28], MTPB [29], DS-1000 [30], and MultiPL-E [31]. Among the listed benchmarks, only CodeContest and MultiPL-E offer additional languages besides Python. Though, MultiPL-E generates problems of other languages by translating existing, simple Python benchmarks using ad-hoc transpilers.

Some benchmarks are manually curated while others are sourced from existing content. The latter suffer potentially from data contamination. Some datasets such as CodeContest do not attempt to address this. DS-1000, on the other hand, slightly modifies problems to be different from their original StackOverflow Source. Since manual curation is time-consuming, it gives motivation for synthetically generating evaluation benchmarks. This is the core idea behind HaskellEval.

2.2.2 Evaluation Methods

Existing works mostly evaluate code LLMs by using one of the aforementioned program synthesis benchmarks. They calculate the pass rate against test cases using the **pass@k** metric, computed as 1 if at least one out of the k sampled programs pass all test cases, or else 0. Frequently, pass@k is reported as a percentage, referring to the percent of problems with a pass@k value of 1.

Manual inspection of the failed snippets can lead to insights for bettering the model. However, not only are they time consuming, they are also harder to yield statistically-backed conclusions from. The following methods can automatically yield results given a corpora. They are presented in the order of increasing interpretability.

Perplexity is defined as

$$\text{Perplexity}(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_1, w_2, \ldots, w_{i-1})\right)$$

where $W = (w_1, w_2, \ldots, w_N)$ is a sequence of N words and P is the probability of the word sequence as assigned by the model, measures the model's uncertainty. A lower perplexity indicates a higher likelihood of the sequence, suggesting better model performance. Perplexity is more reasonably used if no clear domain exists. For example, in evaluating the general stability of quantized models, papers have [32] performed perplexity analysis on a wide variety of corpora.

CodeBLEU [33] assesses the syntactic and semantic accuracy of generated code. The evaluation process combines n-gram matching, weighted keyword matching, syntactic and data flow correctness to provide a holistic measure of code quality.

ASTEval [34] takes it one-step further, aligning and clutering model confidence measures with groups of tokens based on syntactic categories derived from Abstract Syntax

Trees (ASTs). The syntactic categories are curated by the researchers. For example, the authors designed the following categories for Python: *Decision*, *Data Structures*, *Exceptions*, *Natural Language*, *Data Types*, etc. Then, the researchers select the AST concepts that each category contains. For example, nodes of type *comments*, *identifier*, and *string* belong to *Natural Language*.

Then, the model is asked to inference from documents belonging to some corpus. The logit of each token position is extracted and aligned with the AST elements as displayed in 2.1. This process is can be performed automatically in conjunction with tree-sitter [35].

At last, the overall confidence of a syntactic category can be calculated by averaging the bootstrapped median of each constituent AST across the entire corpora. For example, if the model has a bootstrapped median of 49% confidence across all *comments* in the corpora, 89% in *identifier*, and 78% in *string*, then, the model is overall 73% confident in *Natural Language* predictions.

In 4.2, I use ASTEval to enable a finer understanding of Haskell model predictions rooted in syntax-grounded explanations. In the original and adjacent works, analyzing this confidence in a causal manner is also explored [36] [37].

2.3 Haskell

Haskell [38] is a Functional Programming (FP) language. It started development during the late 80s with the clear goal of being suitable for teaching and applications. More importantly, it should be compact and relative easy to extend for research. Its name was chosen in 1988, in honor of Haskell B. Curry, a logician known for his work in combinatorics and lambda calculus.

Functions in FP languages are **first-class**: Functions are values. They may be assigned, passed as arguments, and returned. A function that takes in or returns a function is **higher-ordered**. Many core ideas of FP made its own into main stream, imperative

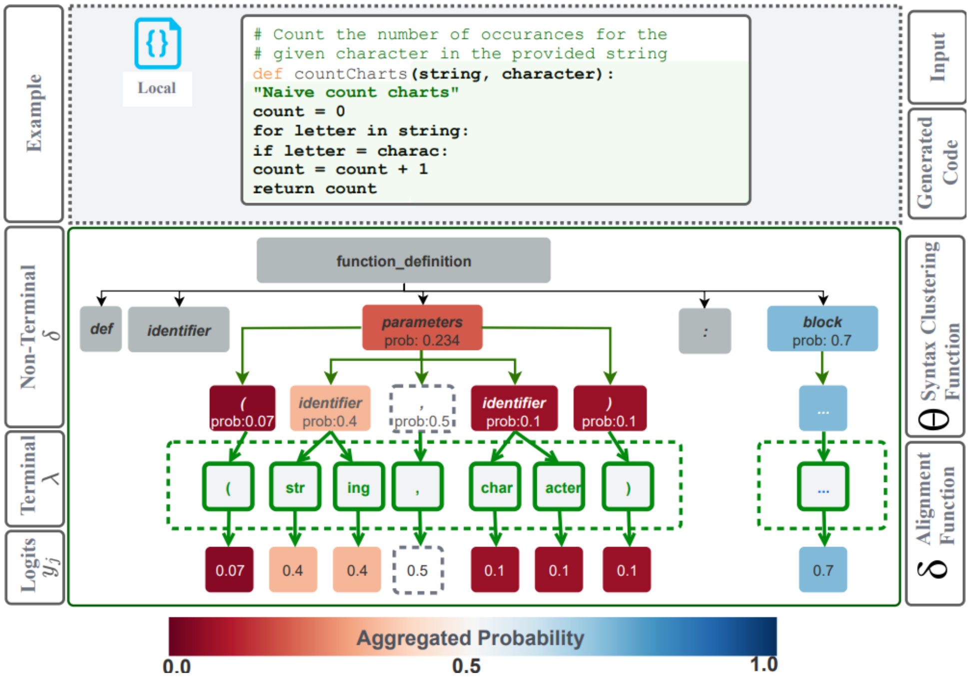

Figure 2.1: Process of aligning logits and probabilities against the AST. A nonterminal's probability is calculated as the average probability of its constituents.

programming (e.g., Python and Javascript). Many languages, even Java, allow functions as first-class values.

The two core principles of Haskell follows: Haskell is lazy; Haskell is pure. A lazy language evaluates expressions only when their results are needed, i.e., call-by-need. This implicitly avoids unnecessary computations. However, call-by-need is usually less efficient than call-by-value due to the extra bookkeeping with thunks. A language that evaluates with call-by-value may mimic call-by-need at the program level. For example, Python implements Generators and Iterables.

Purity is a consequence of laziness. Impurity allows for side effects that are intended to be sequential, e.g., logging. Sequentialness integrates seeminglessly in an eager, call-by-value evaluation model but not in a lazy model. Consequently, lazy languages are pure.

2.3.1 Elements of Haskell

The de facto implementation of Haskell is the **Glasgow Haskell Compiler** (GHC)[39]. GHC's type system is named System FC [40], a typed lambda calculus based off of the System F lambda calculus. System F supports type inferencing, implicitly providing every function a signature. Function signatures, or headers, are **Hindley-Milner headers**.

For example, here is the signature and definition of the function `map`, which has its analogues in many languages.

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Its signature is `map :: (a -> b) -> [a] -> [b]`. The signature implies that `map` takes in two arguments, a function `(a -> b)` and a list `[a]`, and returns a list `[b]`. `map` is a general function (though, it can be further generalized to `fmap` because the list type is a *Functor*) because its signature involves type variables. To use `map`, it must be applied to values of concrete types.

Function application is left-associative. Hence, `f x y` is parsed as `(f x) y`, leading to concise code. While Python writes `map(lambda x: x + 2, [1, 2, 3])`, Haskell writes `map (+2) [1, 2, 3]`.

Haskell implements **currying**. This represents a function of two arguments as a higher-ordered function of one argument that itself returns a function of one argument. Hence, map's signature is internally resolved as `map :: (a -> b) -> ([a] -> [b])`. This gives partial applications for free, which leads to compositional writing. For example, one can partially apply and store `plusTwo = map (+2)`. Then, `plusTwo` may be applied to different integer lists.

An important feature of Haskell (and most FP languages) is **algebraic data types** (ADTs). In the snippet below, the `data` declaration declares `Tree` to be an ADT with two *data constructors*: `Leaf` and `Branch`.

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)

size :: Tree a -> Int
size (Leaf x) = 1
size (Branch t u) = size t + size u + 1
```

A `Tree` is either a `Leaf` or a `Branch` (a sum type with two alternatives). A `Leaf` contains a value (a trivial product type with one field). A `Branch` contains a left and right subtree (a product with two fields). Sum and product types are algebraic due to the effect they induce on the cardinality, the number of different values a particular type can represent. For example, since a tree is either a leaf or a branch, its cardinality is the sum of the cardinality of the leaf type and the branch type.

Data constructors are used to build values and **pattern match**. Writing `Leaf 8` creates a value of type `Tree Int`. Pattern matching is demonstrated with the `size` function, which decomposes a `Tree`, leading to modular code.

In general, Haskell has two programming styles: Declaration style and expression style. The below snippet implements `filter` in both styles.

```haskell
filter :: (a -> Bool) -> [a] -> [a]

-- Declaration style
filter p [] = []
filter p (x:xs) | p x = x : rest
                | otherwise = rest
               where
                 rest = filter p xs

 -- Expression style
 filter = \p -> \xs ->
          case xs of
          [] -> []
          (x:xs) -> let
```

```
                  rest = filter p xs
                in if (p x)
                   then x : rest
                   else rest
```

The declaration style defines a function by multiple equations, each of which pattern matches. It appends the `where` clause to inject additional context. And, it uses guards (indicated by the `|` symbol) to perform conditionals. In contrast, the expression style patterns matches after a `case` expression. It prepends `let` expressions. And, it uses `if` statements.

Lastly, I briefly discuss Haskell's toolings. First, while other implementations exist, GHC is undoubtly the most feature-rich and stable. GHCup provides easy installation of GHC and other build system tools (Stack and Cabal). Haskell debuggers are not as popular as in other languages due to the lazy evaluation order. Testing tools, on the other hand, have been more successful. Hspec provides the traditional unit testing experience. QuickCheck [41] is an advanced framework where users define properties that their function should satisfy, then, QuickCheck automatically generates a large amount of inputs to fuzz test if the property is obeyed. If the function involves custom data types, users must implement logic for how to generate random values for those data types.

2.3.2 Haskell as a Research Language

Haskell is a type-system laboratory incubating many advanced ideas regarding types. One of the simplest and earliest ideas implemented were type classes [42]. Conceptually, type classes are similar to Java interfaces: Type classes specifies the functions that a type must satisfy to be considered an instance of the type classes. However, the difference is in that ADTs satisfy type classes and classes satisfy interfaces. Further, type classes can ascertain the number of type variables (the kind) that an ADTs must conform to. `Tree` has one type variable because of the single `a` in `Tree a`, but there are types such as `data Either a b = Left a | Right b` that have two or more. Many type classes in the standard library integrate algebraic constructs borrowed from group theory and category theory. These type classes are called algebraic type classes. For example,

```
class Monad m where
  (>>=)  :: m a ->  (a -> m b) -> m b
  (>>)   :: m a ->  m b        -> m b
  return ::   a                -> m a

data Maybe a = Just a | Nothing
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

In the above snippet, I first write the definition of the type class `Monad`. Then, I declare the data type `Maybe` and satisfy it to be an instance of `Monad`. Functors, Applicatives [43], and Monads are all fundamental to Haskell.

Monads, at the basic level, can be thought of as containers with a "join" computation defined. In the case of the `Maybe` Monad, a function that is strung together with a `Maybe` (via the `(»=)` operator) is invoked as usual if the `Maybe` is a `Just`, which represents a computation that has a value. If the `Maybe` is a `Nothing`, `Nothing` is the result of the join. That is, this method avoids the need to check for a potential null value in a computation, because if there is a null, `Nothing` is directly returned!

Type Class is one of the most important ideas implemented in Haskell. Subsequently, multi-parameter type classes were implemented. And later, advanced type-level features such as higher-kinded polymorphism [44] and GADTs (generalized ADTs) were enlisted. Recently, there are even advents, not all equally successful, into topics such as linear types [45] and dependent types. These features are enabled with their corresponding `pragma` clause at the top of the file.

Chapter 3

HaskellEval

It is a spontaneous notion, as a functional programmer, to wonder if Haskell is the neglected but better language for LLMs to learn software engineering from. Realistically, this claim can not be fully tested. Not only do existing research focus on Python models and Python evaluation, there is not enough Haskell data to scale a model of the same size as the existing Python ones under current architectures [46].

But, a smaller-scaled evaluation, focused on function-sized program synthesis may be conducted. And as no Haskell evaluation datasets exist, I create HaskellEval [1], a synthetically generated evaluation dataset. HaskellEval is used in the same manner as HumanEval and MBPP. This chapter details HaskellEval's **composition** and its **curation**. The next chapter evaluates models with HaskellEval.

3.1 Overview

The main motivation for synthetic generation is reduced labor; however, this reduction tradeoffs with trustworthiness. Introduced in 2.1.2, synthetic generation methods like Self-Instruct, Evol-Instruct, and OSS-Instruct have strengthened and diversified instruction-tuning datasets. These datasets are similar to program synthesis benchmarks in that they

[1] HaskellEval will be available at https://huggingface.co/datasets/blastwind/HaskellEval after publishing.

are composed of instructions (problems) and responses (solutions). Though, the responses are not limited to a function solution and may mix with natural language. In addition, program synthesis benchmarks are evaluators and hence contain functional test suites. In fact, benchmarks do not need to include canonical solutions.

Since synthetic instruction-tuned datasets achieved success in bettering LLMs, I hypothesize that a majority of the instructions and responses are functionally correct. Thus, the LLMs that synthetically curated the instruction-tuned datasets can hypothetically generate high-quality evaluation benchmarks as well. Can we trust this hypothesis? Can we trust synthetically generated benchmarks?

Before expanding on the question, I first give a detailed description of HaskellEval's generation methodology. To iterate, the goal of this process is to produce a dataset where each item includes the problem and the test suite. For convenience and documentation purposes, the canonical solution is included. It is used in 3.3 for dataset analyses.

The generation process can be summarized in four steps:

1. Problem and solution (P+S) generation
2. P+S validation
3. Test case generation
4. Test case validation

A real conversation covering all four steps is printed in A. All curation is performed with gpt-4-turbo-1105-preview, with temperature=0.5 and top p=0.5 (leans towards being diverse and creative). The full prompts, code to generate the figures, and methodologies are available in the online repo [2].

[2] HaskellEval's repo will be available at https://github.com/WM-SEMERU/HaskellEval after publishing.

3.1.1 P+S Generation and Validation

This subsection provides the methodology for step 1 and 2 in the generation process. An example conversation of these two steps is provided in A.1. To generate a problem and its solution, consider first the following prompt:

> *You are an excellent Haskell problem creator. Please design a creative and high-quality problem and its solution.*

Though promising at a first glance, this prompt yields repetitive problems of similar structures and domains because the command of "design creatively" is simply not enough. At best, the token "creative" might contribute to some internal probability smoothing. Philosophically, if a writer is asked to "write about anything", they likely end up writing about the same topics.

Akin to how a writer sources creativity from a writing prompt, following OSS-Instruct, I inject real world code (from *BlastWind/random_code_snippets* [3]) and ask the model to take inspiration from the code. *BlastWind/random_code_snippets* is comprised of 10K snippets, each containing 5-15 random lines extracted from 1K documents per language across 10 languages (Haskell, Python, C++, Java, TypeScript, Shell, C#, Rust, PHP, and Swift), sourced from *bigcode/starcoderdata* [4].

As such, the prompt now looks like,

> *You are an excellent Haskell problem creator. Please design a high-quality problem and its solution, taking inspiration from the domain of the following seed snippet: <seed>...</seed>*

[3] Available at https://huggingface.co/datasets/blastwind/random_code_snippets.
[4] Available at https://huggingface.co/datasets/bigcode/starcoderdata

Config Category	**Option**	**Weight**
Function Style	Declarative	0.5
	Expression	0.5
Higher Orderness	0	0.7
	1	0.3
Data Type Count	0	0.2
	1	0.45
	2	0.35
Argument Count	1	0.4
	2	0.3
	3	0.3

Table 3.1: Summary of configuration options and their weights. Each configuration is constructed by selecting one item from each category and combining the selections.

Since the seeds are multi-PL, it is best to ask the model to take inspiration from the domain of the seed. Structural and syntactic diversity, on the other hand, are imposed with parametrized configs. This structure injection measure is inspired by Evol-Instruct, which evolves code by using clauses such as "add around 10 words to the problem" and "propose higher time or space complexity requirements in the problem". The model is asked to output problems and solutions that satisfy a combination of configs outlined in 3.1. The 0/1 in the higher orderness category is the same as *not higher order/higher order*. I numerized the order parameter because I originally tested numbers that were higher. But, 2-ordered functions are extremely rare in the wild, and the model was only able to craft superficial ones exploiting the caveat in which Haskell automatically curries. Data type refers to ADT.

The model has knowledge of creating functions that satisfy the outlined configs; however, if not explicitly specified, then, from observation, the model would mostly generate 0-ordered functions. An explanation for this behavior is simply that the model is trained to output what it has seen. Perhaps, the model was tuned to create non higher-order functions as solutions during instruction-tuning of the "problem creation" subspace (for example, competitive programming problems are always 0-ordered). Hence, I am smoothing and decentralizing the implicit distribution that I hypothesize the model to have on Haskell structures.

Together, the prompt is now:

> *You are an excellent Haskell problem creator. Please design a high-quality problem and its solution, taking inspiration from the domain of the snippet in <seed> and obeying each clause in <additional clauses>. <seed>...</seed> <additional clauses>...</additional clauses>*

Some explorations with this edition of the prompt leads to the conclusion that the configs were not always obeyed. More importantly, the problem sometimes did not provide the full context that the solution needed. In the spirit of synthesis, these issues are formatted as validation items in <additional clauses>, and the model is prompted to self-validate whether it followed each clause, leading to the final prompt shown in A.1.

To curate HaskellEval, the model is prompted a total of 142 times: 122 prompts were config-bound and 20 did not impose any configs (as control). If a row is config-bound, it must pick one config from each config category in 3.1 to form a selection of four configs. One can count the number of samples generated in a specific combination by multiplying the weights. For example, $\lceil 0.5 \times 0.3 \times 0.2 \times 0.4 \times 122 \rceil = 2$ rows are asked to be declarative, higher-ordered, use no data type, and use exactly 1 argument.

After P+S generation and synthetic validation, the compiler is utilized to find buggy programs. At last, I manually validate the compilable solutions. The manual procedure is detailed in 3.1. Together, in 3.2.1, the validation results aim to answer the following ***RQs***:

> ***RQ*$_1$*:*** Can we trust synthetic problems and solutions?
>
> - ***RQ*$_{1a}$*:*** Can models self-validate?
> - ***RQ*$_{1b}$*:*** How can we categorize the improper generations? How many are there in each category?
> - ***RQ*$_{1c}$*:*** Do problems take inspiration from seed snippet?

Problems flagged as invalid during the synthetic, compilation, and manual stages are filtered out. As a spoiler, this filtered out most of the problems, leaving HaskellEval a total of 56 rows.

3.1.2 Test Suite Generation and Validation

This subsection provides the methodology for step 3 and step 4 in the generation process. A example conversation of step 3 is featured in A.2. The test validation results, presented in 3.2.1, aim to answer the ***RQs*** below :

> ***RQ*$_2$*:*** Can we trust synthetic test suites?
>
> - ***RQ*$_{2a}$*:*** How can we categorize the failed test suites? How many are there in each category?
> - ***RQ*$_{2b}$*:*** Are the generated test suites comprehensive? If not, can we categorize what they fail to consider?

The test generation's prompt is simpler and more straightforward than the P+S generation's. It asks the model to generate a unit test suite using *hspec*, structuring tests within a single `describe` body with multiple `it` clauses recording expected behaviors, and contains assertions within each `it` to faithfully test the behavior (see the *<Unit Test Suite>* section of A.2). After generation, the canonical solution is executed against the test suite. If there is an error, the test suite is at fault, since, the canonical solution is manually validated. I manually validate the dataset to answer $\boldsymbol{RQ}_2$.

As a spoiler, 13 test suites are invalid. Since the dataset is small, I corrected these improper test suites, resulting in the 56-rowed HaskellEval. I did not correct and interfere with the synthetic problems because I consider the synthetic nature of the dataset to be essential.

At last, $\boldsymbol{RQ}_3$ explores the properties of the dataset:

> ***$\boldsymbol{RQ}_3$:*** How does HaskellEval compare to other datasets?
>
> - ***$\boldsymbol{RQ}_{3a}$:*** How diverse is HaskellEval in comparison?
> - ***$\boldsymbol{RQ}_{3b}$:*** How closely does HaskellEval simulate real-world scenarios in comparison?

I address $\boldsymbol{RQ}_3$ in 3.3, comparing the intra-dataset and inter-dataset textual and semantic similarities.

3.2 Generation Trustworthiness

This section presents the results and ensues the discussion on the trustworthiness of P+S generations and test suite generations, answering $\boldsymbol{RQ}_1$ and $\boldsymbol{RQ}_2$, respectively.

3.2.1 P+S Trustworthiness

3.1 breaks down the synthetic and manual validations of the problem and solution generation. I first address and discuss $\boldsymbol{RQ}_{1a}$: The model is asked to perform two types of self-validations, and it fails in both.

First, the model is unable to determine if additional context is needed to implement the solution. It is asked the following in <additional clauses>:

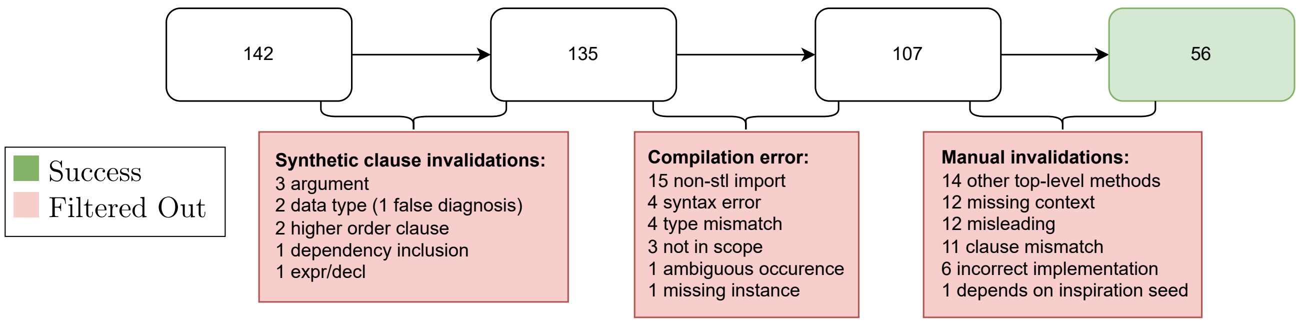

Figure 3.1: P+S validation breakdown. Note, a P+S may belong in multiple error subcategories.

> *Ensure that <Problem> sufficiently and faithfully provides all the information for a human to write a function functionally that is equivalent to the <Solution> function.*

However, as shown in 3.1, there exists 12 problems with missing context; the synthetic invalidations found *none* of them. This suggests either a better prompt is needed, or that the reasoning ability of GPT-4-turbo is not enough. Many parts in the experiment provides evidence that the latter is more often the culprit. At a certain point, it seems that GPT-4-turbo is simply unresponsive to additional instructions.

Second, the model self-validates poorly whether its generations followed the configs. The synthetic pipeline was able to catch 9 config mismatches; however, I discovered 11 uncaught mismatches (example in A.3). Further, 1 of the mismatches was falsely flagged.

Hence, I conclude: **No, model does not self-validate well**. One implication to draw is that instead of focusing on self-validation, it might be better to invest on designing a prompt that can create the best first generation, leading to the following results for $\boldsymbol{RQ}_{1b}$.

Errors fall into 3 main categories: **bad problem and solution**, **other noncompilances**, and **compilation errors**; subcategories and counts are presented in 3.1. Overall, only 56/142, or 39.4% of the generations are trustworthy. However, if I grant a bit of leniency, i.e., allow non standard library (stl) imports and top-level methods to be considered, this ratio increases to 78/142, or $\approx 55\%$ (a couple recovered rows failed for other reasons).

I now list the composition of each main error category, define the error subcategories, and explain why the leniencies may be considered. Ultimately, these leniencies were not granted.

Bad problem and solution The core of P+S generation is the problem and solution. The solution must be correct and implementable solely from the problem description. A bad P+S example, one for each subcategory, is shown in A.4, A.5, and A.6, respectively.

- *Missing context*: Often, problems with missing contexts did not provide some hard-coded value in which the solution uses. These values are impossible to infer from the problem description alone.
- *Misleading*: A misleading problem is confusing because it includes and emphasizes unused context.
- *Incorrect Implementation*: The solution fails to perform what the problem is asking for.

Other noncompilances

- *Non-stl import*: The prompt asks to generate code that only depends on the stl. This ensures that any system with just the GHC compiler will be able to execute HaskellEval's test suite. However, in Haskell, certain imports that appear standard (e.g., imports from *Data.List.Split*, *System.FilePath*, and *Data.Map*) are third-party. In modern Haskell build tools, these are pre-installed but "hidden" by default. The 15 non-stl imports encountered are exactly of this imports of this flavor. If leniency is granted, GHC can unhide libraries used by HaskellEval with the `-package` flag. Note, non-stl imports also result in compilation errors.
- *Other top-level methods*: The prompt asks to generate only one top-level method in the solution. This is because the main way to test function completion is to

provide the model everything except the target function. This would feed other top-level methods to the model. However, other top-level methods may contain logic that contributes to solving the problem, and hence reduces the problem's difficulty. Complying with this rule removed some of the best problems because good problems are often complex and naturally involve other functions (e.g., A.7). Hence, leniency may be considered.

- *Independence from seed*: While the problem should take inspiration from the seed snippet, it must be sufficiently independent from it. I identified exactly one case in which the problem directly referenced the seed.

Compilation Error

- *Non-stl import*
- *Syntax Error*: Two of the four syntax errors are improper spacings. The other 2 used keywords as variables names.
- *Type Mismatch*: In all type mismatches, there was an attempt in matching a composite type with only one of its consituents (e.g., `Maybe Result` with `Result` and `[User] -> String` with `String`).
- *Not in scope*: The solution missed some imports.
- *Ambiguous occurence*: In one occurence, the solution defined a *Left* data type which collides with the stl's *Prelude.Left*.
- *Missing instance*: In one occurence, the solution used a typeclass method on a data type that did not satisfy the typeclass (namely, *Double* is not a *FromIntegral*). This may be categorized under *Type Mismatch*.

While current P+S generations have a lot of errors, some of them can be automatically rejected. And, only the subcategories of *incorrect implementation*, *type mismatch*, and *syntax error* are limited by the generator's Haskell capabilities. When viewed in this lens, there are only 14 true failures. This implies, with sufficient prompt engineering, then even with the complicated setup of taking inspiration from seeds and obeying clauses, P+S can still achieve a potential 90% trustworthiness.

At last, I come to $\boldsymbol{RQ}_{1c}$: Are problems inspired by seed snippets? I evaluate the final, 56-rowed HaskellEval, categorizing inspirations into "yes", "far-fetched", and "no". The results are displayed in 3.2. An example of a successful, far-fetched, and failed inspiration is shown in A.8, A.9, and A.10, respectively.

Problems are inspired except for 14.2% of the cases. In examining the failed cases, I made an unexpected discovery: **Uninspired problems are very similar to each other**. This observation was intially made from another direction: I first manually examined the generated functions and found that 11 of the function names start with the word "calculate", 11 contain the word "find", 6 contain "filter", and 6 contain financial words ("budget", "tax", "discount", and "loan"). Concerningly, 3 functions were named "findOldest", and 2 were named "calculateDiscount". Upon examining their responsible seed, I find these problems did not take inspiration from the seed. Their seeds were sufficiently different from each other, which removes any confounding suspicision.

Hence, it seems that **if a model is unable to form any associations with the seed, it loses creativity all together, devolving into a "stochastic parrot"**. Such behavior was unexpected because the powering model is GPT-4-turbo inferenced at a relatively high temperature and top p.

While there are functions of the same name, they still have sufficient differences due to different configs. Hence, even if a problem did not take inspiration from its seed, it is still included in HaskellEval.

Yes	Far-fetched	No
73.2%	12.5%	14.2%

Table 3.2: Breakdown of whether problems take inspirations. Far-fetched inspirations require multiple jumps in logic.

3.2.2 Test Suite Trustworthiness

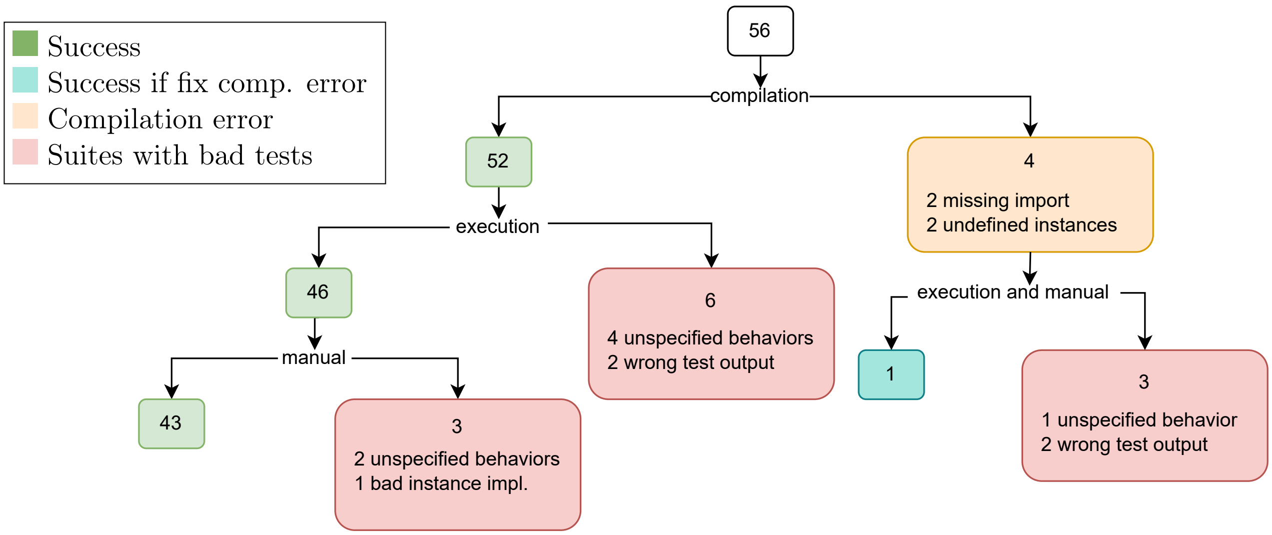

Figure 3.2: Test suite validation breakdown.

3.2 breaks down the automatic and manual validations of the test generations. The flowchart continues after compilation errors since I corrected errors incrementally (test suites with compilation errors were corrected and placed back into the execution pool). To answer $\boldsymbol{RQ}_{2a}$, errors may be categorized by the process in which they are discovered: Compilation, execution, or manual validation.

Compilation The suite and the canonical solution are combined and compiled. A bit of lexical manipulation is required here since a Haskell file must be structured in the order of pragmas, imports, and at last, code body.

- *Missing import*: The test suite may utilize stl imports. In two cases, the model did not import the utilized function.
- *Undefined instances*: `Eq` and `Show` are type classes that implement logic equivalent to `__eq__` and `__str__` in Python. In two cases, the model did not write these instances but used the type class methods.

Execution Since the canonical function is manually validated, then, if it does not pass on a certain test suite, it implies the test suite is incorrect.

- Wrong test output: In four total cases, the implementation of some test behavior is incorrect. In two out of the four scenarios, the model struggled with arithmetic operations.
- Unspecified behavior: The model wrote test cases expecting a behavior that the problem did not imply.

Manual Validation

- Unspecified behavior: Unspecified behaviors in the *Manual Validation* category are that the canonical solution happen satisfy but the problem does not necessarily imply.
- Bad instance implementation: In one case, the *Show* instance was trivially written, passing incorrect solutions.

Originally, I expected "incompleteness" to be a large manual validation error subcategory. Surprisingly, to answer $\boldsymbol{RQ}_{2b}$: All test suites were comprehensive! In fact, the model seems to overly comprehensive. In the seven unspecified behavior cases, the model included edge cases that does not belong to the problem. Since the test generator is an individual component, it is even a promising direction to use the test generator on existing P+S pairs and convert them into functional benchmarks.

3.3 Dataset Analysis

The apparent ability of models to generalize, contrasted with the lurking presence of the "stochastic parrot," provides a basis for examining creativity measured by the distribution of similarities. In this section, I analyze three evaluation datasets: HaskellEval, MBPP, and HumanEval. MBPP and HumanEval are both Python datasets. Unfortunately, no Haskell evaluation dataset exist for me to compare against, hence, language disparity is a threat to validity of the following experiments. Two real world datasets, *BlastWind/github-code-haskell-function* [5] and *BlastWind/github-code-python-function* [6] serve both as baselines and are the targets for real-world similarity comparisons. They are both extracted from *CodeParrot/github-code* using tree-sitter by matching on function nodes. The datasets contain a function's repository, license, and code. I preprocess them by selecting only one function per repo and extracting a random group of 5K samples to use in evaluation.

I now describe the methodology. My approach is twofold: intra-dataset, which explores the diversity within a dataset by assessing the distribution of document similarities (lower similarities imply higher diversity); and real-world, where I gauge the similarity of my evaluation dataset to the aforementioned real-world datasets by associating each evaluation row with a real-world row. These two analyses answer $\boldsymbol{RQ}_{3a}$ and $\boldsymbol{RQ}_{3b}$, respectively. Additionally, the token length distributions are presented in B.1, B.2, and B.3.

In both analyses, I employ two similarity techniques: Textual and semantic. For textual similarity, I utilize term frequency–inverse document frequency (tf-idf) to vectorize each document. The tf-idf score is calculated by

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t)$$

, where

$$\begin{aligned} tf(t,d) &= \frac{\text{Frequency of term } t \text{ in document } d}{\text{Total number of terms in document } d} \\ idf(t) &= \log\left(\frac{\text{Total number of documents}}{1 + \text{number of documents containing the term } t}\right) \end{aligned}$$

For semantic similarity, I use the embedding layer from CodeLlama [47] in order to capture deeper relationships. However, because the embedding layer is token-independent, the relationship captured is not contextual. Only in later layers are tokens allowed to look at earlier tokens (self-attention) and contextualize.

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

[6] Available at https://huggingface.co/datasets/blastwind/github-code-python-function

Later layers hence theorectically provide a more complete picture. However, experiments have shown that later layers extract higher-level features that are often less interpretable [48] [49]. Embedding layers provide less context but are more reliable, allowing marvelous relationships like *King - Man + Woman = Queen* to form [50]. Due to this tradeoff, exploratory experiments using later, non-embeddings layers are presented as well in the next subsection.

In both textual and semantic comparisons, documents are eventually transformed into vectors to be compared with each other using cosine similarity (closer to 1 indicates higher similarity). Textual analysis performs this in a straightforward manner: The output of tf-idf is the desired vector representation of a document. Semantic analysis requires more work: The embedding layer maps each token in the document to an embedding vector (size 4096 in CodeLlama). In order to obtain a single vector that represents the entire document, I perform mean pooling, averaging all embeddings vectors and condense them into one.

3.3.1 Intra Similarity

This section answers $\boldsymbol{RQ}_{3a}$: How relatively diverse is HaskellEval? I present the textual and semantic similarity distribution of each dataset in 3.3 and 3.4, respectively. All plots in this chapter are Kernel Density Estimation (KDE) plots where the Y-axis shows density, with higher density indicating a higher likelihood of observing the x-value. The list of graphed similarities is formed by taking the cosine similarity of each document (source code in this case) against every other document. Hence, for the 56-rowed HaskellEval, a total of $56 \times 55 \div 2 = 1540$ unique cosine similarities are graphed.

In the previous section, I made an unexpected discovery: Uninspired problems are similar (couple problems even had the same name). Hence, I held the hypothesis that the textual similarity of HaskellEval will be relatively higher. This is what 3.3 communicates. But, the margin is very narrow, and most cosine similarities occur between 0.0 and 0.05. Hence, textual similarity does not provide a clear picture of relative diversity differences.

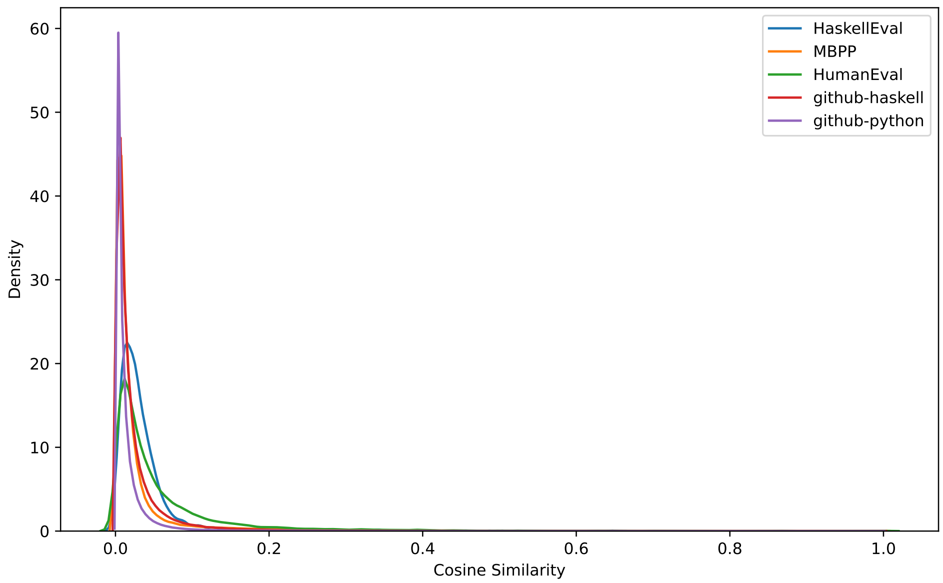

Figure 3.3: Distribution of textual cosine similarities within each datset.

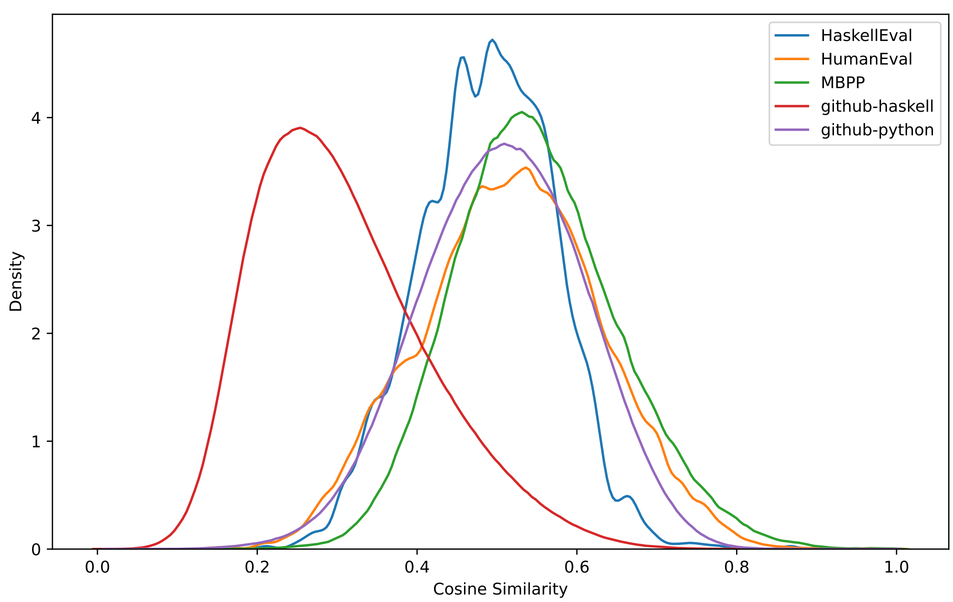

Figure 3.4: Distribution of semantic cosine similarities within each datset.

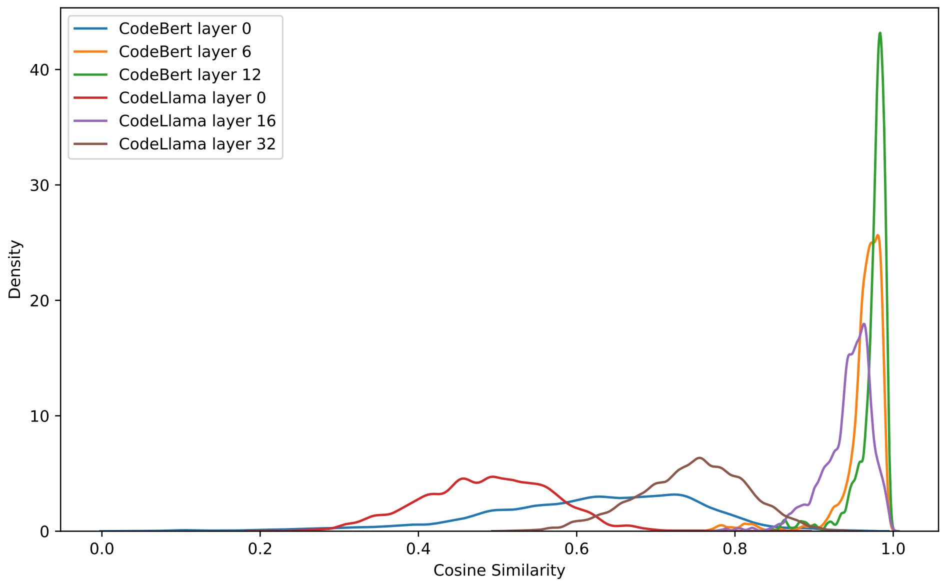

Figure 3.5: Distribution of semantic cosine similarities of HaskellEval, measured using different models and layers.

Semantic similarity finds the cosine similarities to be on a different scale with higher standard deviations (most cosine similarities fall within the 0.4 to 0.6 range). The results are similar, except for a single clear outlier: github-haskell. The implication is that, in comparison, github-python (and the evaluation datasets) are much more similar to themselves than github-haskell is to itself. This suggests that github-haskell is the most diverse. But also, in comparison, while Python evaluation datasets have diversity on par with real Python functions, HaskellEval, on the other hand, does not. The restriction to using the standard library and the aforementioned, stochastic parroting issues may be affecting its diversity potential, providing grounds for further research in how different ablations and prompts can positively affect the internal diversity.

Though not the focus, I offer a hypothesis for why github-haskell is clearly more diverse than github-python. Python has an established ecosystem with 85% developers using it as their first language [51]. Python developers are mostly doing data analysis, web development, and machine learning. As such, a lot of Python code can be repetitive, leading to higher similarities. Certainly, without a rigorous mining and clustering, these are mere speculations.

Lastly, the choice of models and layers to use is variable. I present the intra semantic similarities distribution for HaskellEval in 3.5, using different layers of CodeLlama, and additionally, CodeBert [52]. For semantic similarities, Bert and CodeBert are often used, however, since CodeBert was not trained on specific Haskell clusters, I elected to use the more powerful CodeLlama. In most cases, later layers contribute to higher similarities. Interestingly, the last layer of CodeLlama generates lower cosine similarities than the its middle layer. Due to this uncertain variability, I use the more reliable embedding layer (layer 0) as the semantic encoder.

3.3.2 Similarity to Real World Datasets

This subsection answers $\boldsymbol{RQ}_{3b}$: How well does HaskellEval simulate real-world scenarios relatively? While intra similarity analyses can provide comparisons between the overall dataset similarities between evaluation datasets and real-world corpora, this comparison is *global*. In contrast, the similarities in this section are *local*.

The methodology follows: I vectorize the datasets, then, for each document in the evaluation dataset, I associate with it its most similar counterpart in the real-world dataset. This maximum similarity is recorded. A list of these maximum similarities plotted, resulting in 3.6 and 3.7. Such similarity is local because I am comparing across the evaluation and real-world datasets on each evaluation sample.

Due to taking maximum similarity, the textual similarity graph gives a clear result. HaskellEval is quite a bit less similar to github-haskell than the Python evaluation datasets are to github-python. The semantic similarity graph yields similar observation.

This is a bit concerning. Though, similarity to real-world dataset is not necessary for an evaluation dataset is to be of high quality. In fact, if a problem is meant to test the boundary of model performance, then it should not be similar to real-world dataset.

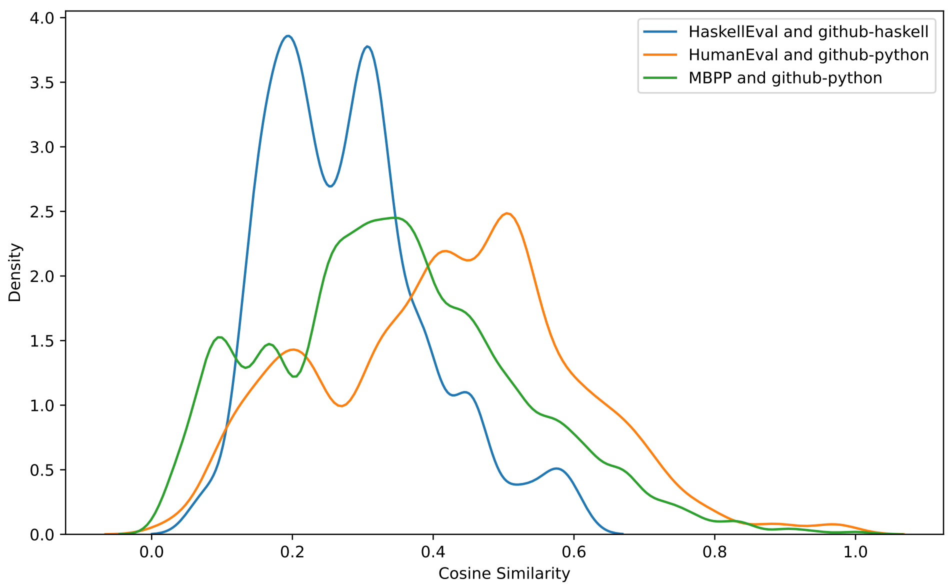

Figure 3.6: Distribution of max textual cosine similarities between evaluation and real-world datasets.

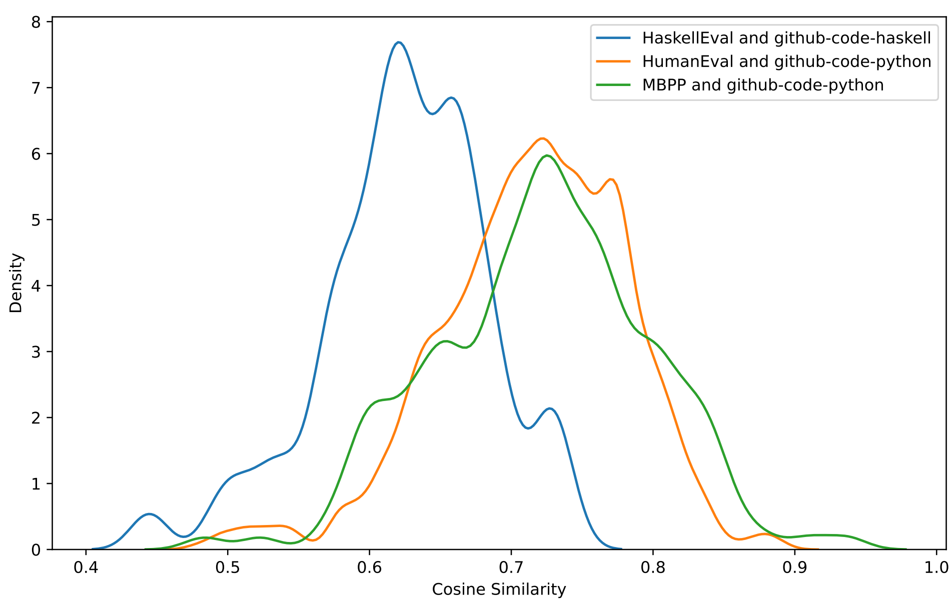

Figure 3.7: Distribution of max semantic cosine similarities between evaluation and real-world datasets.

Further, dissimilarity from real-world dataset (that were likely used in model training) require the model to use more generalization capabilities.

The result is dichotomous. Interestingly, the diversity controllers probably affected the real-world similarities in opposite directions. The configuations likely decreased similarity since they were designed to walk the model into spaces it frequents less. The seed snippet likely increased similarity since they imbued the problems with real-world domains.

3.4 Conclusions and Future Directions

HaskellEval's turbulent curation process provides an important message: We must be careful with synthetic methods. Instruction-tuning with synthetic datasets have shown to dramatically improve performance. But perhaps, further hidden performances can be unleashed if the dataset is more correct. Though self-validation did not perform well, it is not out of the picture. Perhaps, it can significantly improve over a multi-turn conversation, with the LLM serving both as a "problem solver" and a "problem reviewer".

One of the most promising results is the comprehensiveness of the synthetic test suites. Perhaps, the test generator can be used on existing problems and solutions and convert them into functional benchmarks. Though, the issue of curating a list of problems and solutions in the first place still exists for Haskell.

During the generation, a control set of 20 problems were generated without any configs. Within these problems, none contained higher order functions. But, this is a small-scaled experiment. More ablations, including ablating the inspiration seed, can be conducted to better understand how diversity can be controlled. In the 14.3% of cases where the problem took no inspiration, it would be interesting to see if a chain-of-thought prompt can correct this behavior. On the other hand, it is also of interest to see if the comprehensiveness of the test suites drop if their <Workspace> section was removed.

Without understanding the true reasons, HaskellEval's dissimilarities (and the Python datasets' similarities) is an ambiguous measure. Hence, my exploratory analyses motivate the study of attribute effects on semantic similarity. For example, if the domain (e.g., web dev, data science) affects the semantic similarity the most, then HaskellEval's dissimilarities are odd. However, if it is more about the syntax and structure, then the dissimilarities are explainable. Further, the choice of mean pooling the semantic representations of LLMs require more scruntiny.

At last, many parts of HaskellEval is reusable. Hence, I want to abstract the generation process of HaskellEval out into a language-agnostic framework.

***RQ* Result Summaries**:

***RQ*$_1$**: Can we trust synthetic problems and solutions?

- ***RQ*$_{1a}$**: Models did not self-validate well against my clauses (see A.3).
- ***RQ*$_{1b}$**: Improper generations can be categorized into *bad problem and solution*, *compilation errors*, and *other noncompilances* (see A.4, A.5, and A.6 for bad P+S, see the counts in 3.1).
- ***RQ*$_{1c}$**: The model takes obvious inspiration 73.2% and no inspiration 14.2% of the time (see examples in A.8, A.9, and A.10).

***RQ*$_2$:** Can we trust synthetic test suites?

- ***RQ*$_{2a}$:** Improper test suites can be categorized by their source: *bad compilation*, *bad execution*, or *manual invalidation*. See 3.2 for the counts.
- ***RQ*$_{2b}$:** All test suites were comprehensive!

***RQ*$_3$:** How does HaskellEval compare to other datasets?

- ***RQ*$_{3a}$:** HaskellEval is as diverse as HumanEval and MBPP. However, HaskellEval is significantly more diverse than real-world Haskell functions. See 3.4 and 3.3.
- ***RQ*$_{3b}$:** Relatively, HaskellEval simulates real-world scenarios the least. See 3.6 and 3.7.

Chapter 4

Evaluating Haskell Models

In this section, I evaluate models on the HaskellEval. I aim to answer the following research questions:

> $\boldsymbol{RQ}_1$**:** How well do models pass HaskellEval and which tests are they failing on? How is the passing rate affected by different parameters?

> $\boldsymbol{RQ}_2$**:** How confident are the models on individual concepts?

$\boldsymbol{RQ}_1$ addresses the most direct question: Are the functions generated by the model functionally correct (i.e., passes the test suite)? I evaluate *CodeLlama-13b*, *CodeLlama-7b*, and *Mistral-7b* [53] on a single A100 GPU. These models are selected because they are open source, were trained on a large corpora, including Haskell, and show promising programming abilities when evaluated on other languages. CodeLlama is fine-tuned on code-specific tokens at the end of its training, mistral is not. Larger models (e.g., CodeLlama-34B and Mistral-8x7b) were not included due to the inferencing requirements.

Then, I answer $\boldsymbol{RQ}_2$ using ASTEval to evaluate the AST-wise confidence of models. This metric is used to give syntax-grounded explanations for model performance.

4.1 Evaluating Model Performance on HaskellEval

The methodology follows: In addition to the models, I choose four more parameters: Temperature and top p, precision, k, and repairs. The models are loaded with a specific quantization precision, sampled k times, and inferenced at a certain temperature and top p. At last, failed samples may be repaired a number of times. In my experiments, I load CodeLlama-13b with int4 precision (higher precision does not fit) and CodeLlama-7b, Mistral-7b at both int4 and float16 precision. I inference with two sets of temperature and top p values: $(0.2, 0.1)$ and $(0.5, 0.5)$. I choose $k = 5$ in my experiments. k refers to the k in pass@k, the number of samplings to take. I choose the max repair count to be 2. During the evaluation process, the pipeline compiles and executes each of the k samplings against the test suite. If all base k samplings were incorrect, then they may be repaired one-by-one, using the results from the test suites. An example of a successful and failed conversation is presented in C.1 and C.2, respectively.

repair count \k	**1**	**2**	**3**	**4**	**5**
0	0.46	0.5	0.52	0.52	0.52
1	0.52	0.52	0.52	0.52	0.52
2	0.52	0.52	0.52	0.52	0.52

Table 4.1: The pass@k rates of CodeLlama-7b infereced with float16 precision and temp and top p set to 0.5.

repair count \k	**1**	**2**	**3**	**4**	**5**
0	0.46	0.52	0.52	0.52	0.52
1	0.52	0.54	0.54	0.54	0.54
2	0.55	0.55	0.55	0.55	0.55

Table 4.2: The pass@k rates of CodeLlama-13b infereced with int4 precision and temp and top p set to 0.5.

Results are presented in 4.1, 4.2, 4.3 4.4, and 4.5. 4.1 and 4.2 contrast to show the positive effect of model size: CodeLlama-13b performs better than CodeLlama-7b, even if inferenced with the int4 precision (which uses less memory than CodeLlama-7b at float16).

repair count \k	1	2	3	4	5
0	0.41	0.48	0.48	0.48	0.48
1	0.48	0.48	0.48	0.48	0.48
2	0.48	0.48	0.48	0.48	0.48

Table 4.3: The pass@k rates of CodeLlama-13b infereced with int4 precision, temp set to 0.2, and top p set to 0.1.

repair count \k	1	2	3	4	5
0	0.32	0.36	0.39	0.39	0.39
1	0.39	0.39	0.39	0.39	0.39
2	0.39	0.39	0.41	0.41	0.41

Table 4.4: The pass@k rates of Mistral-7b infereced with float16 precision and temp and top p set to 0.5.

4.2 and 4.3 contrast to show the potentially adverse effect of temperature. This is a bit surprising, since, being more deterministic and adhering to established patterns (i.e., lower temperature and top p) should improve code generation. Though, this adverse effect is not observed in the other models. The most surprising result is shown in contrast of 4.4 and 4.5: Inferencing at lower precision answered 2 extra questions right. This is best explained with the lack of experiments. My pass@k experiments were only ran once; while each model is sampled $k = 5$ times, this is not enough repetition. However, this gives hope that quantization retains performance significantly well in code generation. Previous metrics of quantization mainly involved standard metrics such as perplexity [32], but it would be interesting to explore the effect of quantization on functional correctness.

Lastly, every single table demonstrates the relationship of increasing sample size versus increasing repairs. The ith row and the jth column of each table corresponds to the passing rate if j samplings were repaired i times. In my experiments, all k samplings must fail before repair. Hence, if the table is read left-to-right and top-to-bottom, each cell must be greater or equal to the preceding cell. The biggest performance yields occured in the first sample's first repair and the first resampling. Both yield a median of 7% increase across the tables. Do note, this setup disproportionally highlight the utility of repairs, since repairs only occur after all samples fail.

repair count \k	1	2	3	4	5
0	0.32	0.41	0.45	0.45	0.45
1	0.45	0.45	0.45	0.45	0.45
2	0.45	0.45	0.45	0.45	0.45

Table 4.5: The pass@k rates of Mistral-7b infereced with int4 precision and temp and top p set to 0.5.

To further study the relationship, the setup where a failed sample is immediately taken to repair must be considered. Such studies can motivate, for example, if it is more cost efficient for code agents to resample or repair. Some results are explored in [54].

At last, I observe that 29% (16 problems) were never passed by any configurations. The failures are a combination of bad base sample, incorrect error diagnosis, inability to utilize unit test results, and, inability to follow examples provided in the problem description. In the failed passes, the repairs all are very modest. In many cases, they output the exact same function. Hence, they do not perform well when the base sample is off track. If the repair includes a step-by-step diagnosis (which is not explicitly elicited), the output is more likely to variate, though it never changes the core structure of the output. For example, if a type mismatch is caught in an argument of a function application, the repair always attempts to fix the argument instead of replacing the function usage all together. Importantly, none of these errors involved syntactic mishaps, illustrating the models' prominent knowledge in Haskell structure.

4.2 ASTEval

I introduced ASTEval in 2.2.2. ASTEval measures model confidence aligned with AST elements, providing syntax-grounded understandings. In my experiment, I make one tweak: The syntactic category median is not the median of the constituents' medians. Rather, all of the constituents' medians are collected into one array, and the syntactic category median is the median of this array. This is to avoid rare AST types disproportionally affecting the cateogory's probability.

For Haskell, I designed 11 categories: *Type System*, *Expression Style*, *Declarative Style*, *Evaluation and Precedence*, *Syntactic Sugar*, *Function*, *Modularization*, *Control Flow*, *Names*, *Operators*, and *Composite Types*. I illustrate the ASTEval results for the five most important categories in 4.6. The full result and taxonomy is present in the online repository.

Type System refers to AST elements that either add type definitions to the available space (e.g., data definition) or elements that uses the type system (e.g., the Hindley-Milner header). *Expression Style* and *Declarative Style* are covered in 2.3. *Function* is composed of all AST elements that act as functions: Lambdas, regular functions, binary operators, etc. At last, *Name* refers to any AST element that introduces new names (e.g., a regular function). *Name* intersects with *Type System*.

The models in 4.6 are inferenced with the default, float16 precision. 3 corpora are used: HaskellEval (P+S), HaskellEval, and github-haskell. All datasets are functions. In the case of HaskellEval (P+S), I additionally include the problem description. As expected, this additional context increases model confidence across all categories, with significant improvements on *Type* and *Name*.

Models are significantly less confident about *github-haskell* than HaskellEval. A leading reason is, again, context. HaskellEval is designed to be self-containing, while, github-haskell is composed of functions that may utilize names and functions defined in the same repo but not included in the same function body.

Most interestingly, Mistral-7b, which performs worse on the functional tests, outperforms other models in the HaskellEval datasets. However, Mistral-7b lacks against the real-world, *github-haskell*. This indicates that Mistral-7b is a great memorizer, yet, it has still yet to develop enough generalization abilities to match the code-oriented models.

Overall, discluding the scenarios where insufficient context were provided, the models perform well. In the original work, the authors used a confidence threshold of 0.6 to separate between trustworthy categories and untrustworthy categories. All models pass this threshold when maximum context is provided, i.e., in HaskellEval (P+S). However, there is still a clear gap between certain categories. Haskell programming has generally two styles: Declarative or Expressive. Per the results of 4.6, models are more confident in the expressive style than the declarative style. Upon further examinations, this is due to one outlier: *patterns* (introduced in 2.3). In the case of limited context, i.e., in *github-haskell*, models can not pattern match because they do not know the data definitions. Across models on *github-haskell*, *patterns* has a low 25% confidence. In the case of full context, *patterns* has a 68% confidence. Likely, models have not formed the intimate connections between the data definitions, type headers, and pattern matchings, motivating additional tuning that specifically addresses these gaps.

Model	**Dataset**	**Type**	**Expr**	**Decl**	**Func**	**Name**
CodeLlama-13b	HaskellEval (P+S)	0.72	0.84	0.70	0.82	0.66
	HaskellEval	0.55	0.80	0.67	0.77	0.55
	github-haskell	0.29	0.62	0.25	0.44	0.29
CodeLlama-7b	HaskellEval (P+S)	0.71	0.83	0.70	0.80	0.65
	HaskellEval	0.54	0.78	0.66	0.76	0.53
	github-haskell	0.30	0.63	0.26	0.44	0.30
Mistral-7b	HaskellEval (P+S)	0.77	0.84	0.75	0.81	0.70
	HaskellEval	0.60	0.78	0.69	0.72	0.59
	github-haskell	0.27	0.56	0.22	0.41	0.27

Table 4.6: Median confidence of Haskell syntactic categories. Dataset HaskellEval (P+S) prepends the problem description, providing more context to the solution. Dataset HaskellEval contains just the solution.

Chapter 5

Conclusion

This thesis introduced HaskellEval, a benchmark tailored for assessing the capabilities of Large Language Models (LLMs) on Haskell, a functional programming language that has been overlooked in LLM research. The development of HaskellEval was driven by a need to understand LLM performance beyond the more commonly used languages like Python.

Throughout this study, the trustworthiness of synthetic data generation has been a focal point. Manual validations reveal that while synthetic methods can efficiently produce large datasets, the trustworthiness of these datasets are questionable. Additional research into self-validation methods can potentially provide a self-sustaining synthetic data generation method with minimal human intervention.

Future advancements in synthetic dataset generation and validation methods will not only enable reliable and fast generation of evaluation datasets, but also improve upon existing synthetic instruction-tuning datasets, leading to the creation of more robust language models.

Appendix A

Synthetic Generations

Simplified Problem + Solution Generation Prompt

You are an exceptional engine at designing coding problems and crafting their Haskell solution.

--- Output template: START ---
<Problem>
Design implicitly and output explicitly a high-quality, self-contained, 1-4 sentenced coding problem inspired by the domain of #seed_snippet# obeying each clause in #additional clauses#.
</Problem>

<Solution>
Output the solution to <Problem>.
</Solution>

<Validation>
For each #additional clauses#, output the clause and then answer Y/N if <Problem> and <Solution> satisfy the clause.
</Validation>
--- Output template: END ---

I now provide #seed snippet# and #additional clauses#.

#seed snippet#
```
class StrVec
{
private:
    /* data */
    static std::allocator<std::string> alloc;
    std::string *elements;
    std::string *frist_free;
    std::string *cap;
    void reallocate();
    void free();
    std::pair<std::string *, std::string *> alloc_n_copy(const std::string *, const std::string *);
    void chk_n_alloc()
    {
```

#additional clauses#
- The function is written in an expression style
- The function is not higher order
- The function does not use any custom data types
- The function uses exactly 3 arguments

Simplified Model Output

<Problem>
The function `allocateStrings` takes three arguments: an integer representing the number of strings to allocate, a base string, and an integer step. It generates a list of strings where each string is the base string concatenated with an increasing multiple of the step, starting from 1. The output is a list of strings of length equal to the number of strings to allocate. For example, if the inputs are 3, ""base"", and 2, the output should be [""base1"", ""base3"", ""base5""].
</Problem>

<Solution>
```haskell
import Data.List (unfoldr)

allocateStrings :: Int -> String -> Int -> [String]
allocateStrings n base step =
  let
    generator (i, count)
      | count < n = Just (base ++ show i, (i + step, count + 1))
      | otherwise = Nothing
  in unfoldr generator (1, 0)
```
<Solution>

<Validation>
Y: Include dependencies, no other top-level methods
Y: Self containing.
Y: Contain Hindley-Milner type header.
Y: Contains x arguments.
Y: Returned function is y higher-ordered.
Y: Function is structured {expressively/declaratively}
Y: Function uses z data types.
Y: Human can faithfully implement <Solution> from <Problem>.
</Validation>

Figure A.1: The first part of a successful P+S generation conversation. Additional, abbreviated text guides the model to use only the standard library, explains clauses in depth, and hosts sections to ease parsing (e.g., a <function name> section). Note, the solution may include dependencies such as import statements, data type declarations, etc.

Simplified Test Generation Prompt

You are an excellent QA engineer tasked with writing unit tests cases for Haskell functions using the Hspec library. You have great knowledge and experience in the Hspec library. You will write test cases for the function `allocateStrings` in the following haskell snippet:
<snippet>
import Data.List (unfoldr)

allocateStrings :: Int -> String -> Int -> [String]
allocateStrings n base step = ... abbreviated ...
</snippet>

This function solves the following Haskell coding problem:
<problem>
The function `allocateStrings` takes three arguments: an integer representing the number of strings to allocate, a base string, and an integer step. It generates ... abbreviated ...
</problem>

--- Output template: START ---
<Workspace>
First, understand the function. Then, think about how to test the function comprehensively. Consider what arguments can trigger edge cases, what set of arguments is needed to cover the input space and output space. Summarize and output your thinking process in 1-5 sentences.
</Workspace>

<Unit Test Suite>
Design and output a test suite for `allocateStrings` using the Hspec library. The test suite is a `main` function running `hspec` on a single `describe` body with multiple `it` clauses recording expected behaviors, and multiple assertions within each `it` to faithfully test the behavior.
</Unit Test Suite>

<Unit Test Suite Dependencies>
The <Unit Test Suite> might depend on code in addition to <snippet>, output this code in this section.
</Unit Test Suite Dependencies>
--- Output template: END---

Actual Model Output

<Workspace>
To comprehensively test the `allocateStrings` function, we need to consider various scenarios: typical use cases where the number of strings, base string, and step are standard positive values; edge cases such as a zero or negative number of strings to allocate, a zero step, and an empty base string; and boundary conditions like a very large number of strings or steps. We should also test the function's behavior with different types of characters in the base string, including special characters and numbers, to ensure it concatenates correctly in all cases.
</Workspace>

<Unit Test Suite Dependencies>
```haskell
import Test.Hspec
```
</Unit Test Suite Dependencies>

<Unit Test Suite>
```haskell
main :: IO ()
main = hspec $ do
  describe ""allocateStrings"" $ do
    it ""generates a list of strings with positive step and count"" $ do
      allocateStrings 3 ""base"" 2 `shouldMatchList` [""base1"", ""base3"", ""base5""]
      allocateStrings 2 ""item"" 3 `shouldMatchList` [""item1"", ""item4""]

    it ""handles an empty base string correctly"" $ do
      allocateStrings 2 """" 1 `shouldMatchList` [""1"", ""2""]

    ... 5 more test cases abbreviated ...
```
</Unit Test Suite>

Figure A.2: The second part of a successful P+S generation conversation. This is the test suite generation continuing A.1. The full prompt (abbreviated here) provides a one-shot example of a real Hspec test suite for a simple `add` function.

Simplified Problem + Solution Generation Prompt

You are an exceptional engine at designing coding problems and crafting their Haskell solution.

--- Output template: START ---
<Problem>
Design implicitly and output explicitly a high-quality, self-contained, 1-4 sentenced coding problem inspired by the domain of #seed_snippet# obeying each clause in #additional clauses#.
</Problem>

<Solution>
Output the solution to <Problem>.
</Solution>

<Validation>
For each #additional clauses#, output the clause and then answer Y/N if <Problem> and <Solution> satisfy the clause.
</Validation>
--- Output template: END ---

I now provide #seed snippet# and #additional clauses#.

#seed snippet#

```
        }

        System.out.println(""sum=""+sum);  //2

        //3.创建对应的稀疏数组
        int[][] sparseArr = new int[sum + 1][3]; //第一行存储row,column,value数据
        //4.给稀疏数组赋值
        sparseArr[0][0] = rowNum;
        sparseArr[0][1] = colNum;
        sparseArr[0][2] = sum;

        //遍历二维数组，将非0的值存放到稀疏数组中
```

#additional clauses#
- *The function is written in an expression style*
- *The function returns another function of order 0*
- *The function uses exactly 2 custom data types*
- *The function uses exactly 2 arguments*

Simplified Model Output

<Problem>
Given a two-dimensional list of integers representing a chessboard where 0 indicates an empty square and any other integer indicates a piece on that square, write a function `chessboardToSparse` that converts this chessboard into a sparse representation. The sparse representation is a list of tuples, each representing a non-empty square with three integers: the row index, the column index, and the value (piece) on that square. The first tuple in the list should contain the total number of rows, the total number of columns, and the total number of non-empty squares on the chessboard. Assume the chessboard is non-empty and contains at least one square.
</Problem>

<Solution>
```haskell
{-# LANGUAGE LambdaCase #-}

type SparseRep = (Int, Int, Int)
-- Represents a sparse representation of a non-empty square: (row index, column index, value)

type Chessboard = [[Int]]
-- Represents a chessboard as a two-dimensional list of integers

chessboardToSparse :: Chessboard -> [SparseRep]
chessboardToSparse cb = let
    numRows = length cb
    numCols = length (head cb)
    nonEmptySquares = concatMap (\(r, row) -> map (\(c, val) -> (r, c, val)) $ filter ((/= 0) . snd) $ zip [0..] row) $ zip
[0..] cb
    numNonEmpty = length nonEmptySquares
    in ((numRows, numCols, numNonEmpty) : nonEmptySquares)
```
<Solution>

<Validation>
Y: Include dependencies, no other top-level methods
Y: Self containing.
Y: Contain Hindley-Milner type header.
Y: Contains x arguments.
Y: Returned function is y higher-ordered.
Y: Function is structured {expressively/declaratively}
Y: Function uses z data types.
Y: Human can faithfully implement <Solution> from <Problem>.
</Validation>

Figure A.3: This conversation shows a manually invalidated problem in which self-validation failed to catch. Namely, the model falsely answered "yes" to the <Solution> containing 2 custom data types and 2 arguments.

Missing Context

Problem: The function `cancelOrder` takes a string representing a stock symbol and an integer representing an order ID, and returns a string indicating whether the order was successfully canceled. The function should return "Order <order_id> for <symbol> canceled successfully" if the order exists and can be canceled. If the order does not exist, the function should return "Order <order_id> for <symbol> does not exist". Assume there is a predefined list of orders that can be canceled, represented by a custom data type that includes the stock symbol and the order ID. The functionality should be implemented without using `if` statements, `let` expressions, or `case` expressions, and must utilize pattern matching and guards.

Solution:

```
import Data.List (find)

-- Represents an order with a stock symbol and an order ID
data Order = Order String Integer deriving (Eq, Show)
cancelOrder :: String -> Integer -> String
cancelOrder symbol orderId = processOrder $ find (== Order symbol orderId) orders
  where
    processOrder (Just (Order s o)) = "Order " ++ show o ++ " for " ++ s ++ " canceled successfully"
    processOrder Nothing = "Order " ++ show orderId ++ " for " ++ symbol ++ " does not exist"
    orders = [Order "AAPL" 1, Order "GOOG" 2, Order "MSFT" 3]
```

Figure A.4: Nowhere are the strings "AAPL", "GOOG", and "MSFT" mentioned.

Misleading

Problem: The function `calculateEndpointSecurity` takes three arguments: a string representing an endpoint URI, a string representing an endpoint key, and an integer representing the number of times the key has been rotated. It returns a string that indicates the security level of the endpoint. The security level is determined as follows: if the key has been rotated more than 10 times, the security level is "High"; if the key has been rotated between 5 and 10 times (inclusive), the security level is "Medium"; otherwise, the security level is "Low". The output string should be in the format "Endpoint [URI] Security: [Security Level]".

Solution:

```
calculateEndpointSecurity :: String -> String -> Int -> String
calculateEndpointSecurity uri key rotations =
  let securityLevel = if rotations > 10 then "High"
                    else if rotations >= 5 then "Medium"
                    else "Low"
  in "Endpoint " ++ uri ++ " Security: " ++ securityLevel
```

Figure A.5: calculateEndpointSecurity looks to be a perfectly fine and simple problem. However, it takes in a key argument that is never used.

Incorrect Implementation

Problem: The function `createColoredEdges` takes a list of vertices and two distinct vertices `a` and `b` as inputs and outputs a list of edges. Each edge is represented as a tuple containing a vertex, an integer, and a color. Starting from the vertex immediately following `b` in the list (considering the list as circular), each vertex (excluding `a` and `b`) is connected to `-3` with an edge colored 'red'. The function ensures that every vertex in the input list, except `a` and `b`, becomes part of exactly one such edge. The input list represents a simple cycle of vertices in a graph, and `a` and `b` are guaranteed to be elements of this list.

Solution:

```
{-# LANGUAGE LambdaCase #-}

data Edge = Edge (Int, Int, String) deriving (Show, Eq)
-- Represents an edge in a graph with a vertex, an integer, and a color.

createColoredEdges :: [Int] -> Int -> Int -> [Edge]
createColoredEdges vertices a b =
  let
    rotate lst = case lst of
      [] -> []
      (x:xs) -> xs ++ [x]
    rotatedVertices = dropWhile (/= b) vertices ++ takeWhile (/= b) vertices
    edges = map (\v -> Edge (v, -3, "red")) . filter (\v -> v /= a && v /= b) . rotate $ rotatedVertices
  in
    edges
```

Figure A.6: createColoredEdges does not cycle around the list properly.

Simplified Problem + Solution Generation Prompt

You are an exceptional engine at designing coding problems and crafting their Haskell solution.

--- Output template: START ---
<Problem>
Design implicitly and output explicitly a high-quality, self-contained, 1-4 sentenced coding problem inspired by the domain of #seed_snippet# obeying each clause in #additional clauses#.
</Problem>

<Solution>
Output the solution to <Problem>.
</Solution>

<Validation>
For each #additional clauses#, output the clause and then answer Y/N if <Problem> and <Solution> satisfy the clause.
</Validation>
--- Output template: END ---

I now provide #seed snippet# and #additional clauses#.

#seed snippet#

```
Vector2 t = (line->from - line->to).normalized().tangent() * line->width * 0.5;

Vector2 verts[4] = {
    line->from - t,
    line->from + t,
    line->to + t,
    line->to - t,
};

//glLineWidth(line->width);
_draw_gui_primitive(4, verts, NULL, NULL);
```

#additional clauses#
- The function is written in an declarative style
- The function returns another function of order 0
- The function uses exactly 2 custom data types
- The function uses exactly 3 arguments

Simplified Model Output

<Problem>
The function `createLineRenderer` takes three arguments: a function to normalize vectors, a function to find the tangent of a vector, and a line width. It returns a function that, given a line (defined by its start and end points), calculates the vertices of a rectangle representing the line with the specified width. The rectangle is centered on the original line, and its sides are parallel and perpendicular to the line. The output function returns a list of four points, representing the vertices of the rectangle in the order: start-left, start-right, end-right, end-left. Assume points and vectors are represented as pairs of floats.
</Problem>

<Solution>
```haskell
import Data.List (intercalate)

-- Represents a point in 2D space
type Point = (Float, Float)

-- Represents a vector in 2D space
type Vector = (Float, Float)

normalize :: Vector -> Vector
normalize (x, y) = let len = sqrt (x*x + y*y) in (x / len, y / len)

tangent :: Vector -> Vector
tangent (x, y) = (-y, x)

createLineRenderer :: (Vector -> Vector) -> (Vector -> Vector) -> Float -> ((Point, Point) -> [Point])
createLineRenderer normalizeFn tangentFn lineWidth (from, to) = [fromMinusT, fromPlusT, toPlusT, toMinusT]
  where
    vector = (fst to - fst from, snd to - snd from)
    normalized = normalizeFn vector
    t = tangentFn normalized
    halfWidthVector = (fst t * lineWidth * 0.5, snd t * lineWidth * 0.5)
    fromMinusT = (fst from - fst halfWidthVector, snd from - snd halfWidthVector)
    fromPlusT = (fst from + fst halfWidthVector, snd from + snd halfWidthVector)
    toPlusT = (fst to + fst halfWidthVector, snd to + snd halfWidthVector)
    toMinusT = (fst to - fst halfWidthVector, snd to - snd halfWidthVector)
```
<Solution>

<Validation>
Y: Include dependencies, no other top-level methods
Y: Self containing.
Y: Contain Hindley-Milner type header.
Y: Contains 3 arguments.
Y: Returned function is higher-ordered.
Y: Function is structured declaratively
Y: Function uses 2 data types.
Y: Human can faithfully implement <Solution> from <Problem>.
</Validation>

Figure A.7: Example of a problem with multiple top-level functions. It is otherwise sound.

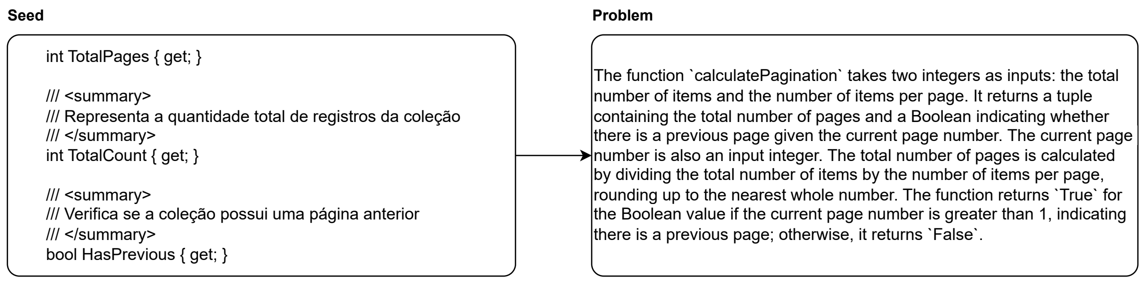

Figure A.8: Example where inspiration is clearly taken.

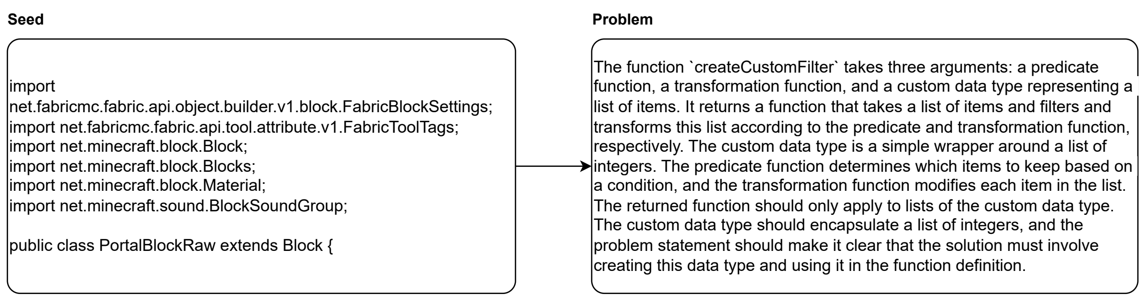

Figure A.9: With some imagination, perhaps, the connection here is that the minecraft blocks and materials require some custom filter.

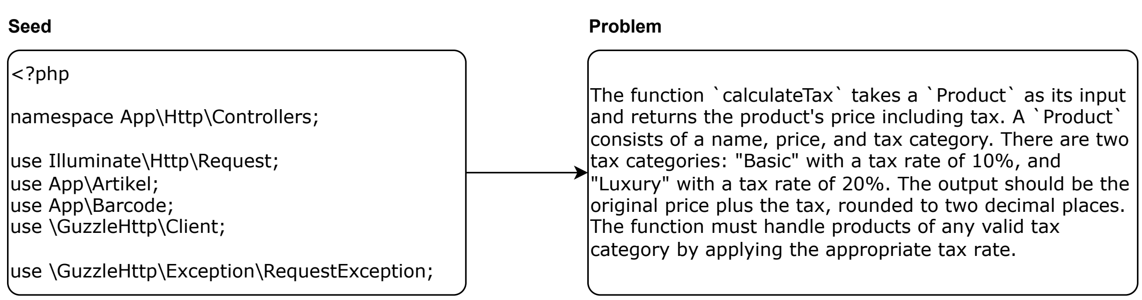

Figure A.10: There is no evident connection between the seed and the problem.

Appendix B

Exploratory Data Analysis

Evaluation datasets contain problems and solutions, both of these are graphed. Evidently, HumanEval contains the longest problem descriptions, with HaskellEval coming in second. MBPP, ranking last, is truly "Mostly Basic", nearly 70% of its 1000 problems can be tokenized under 18 tokens.

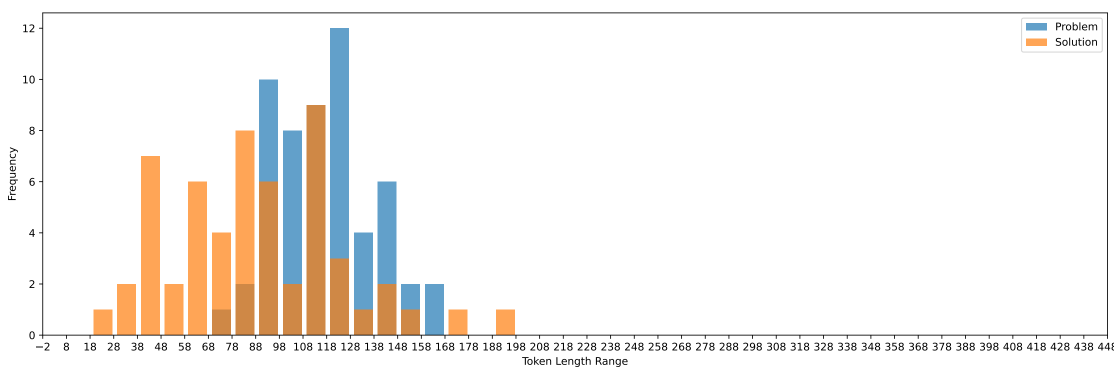

Figure B.1: Token length distribution of HaskellEval.

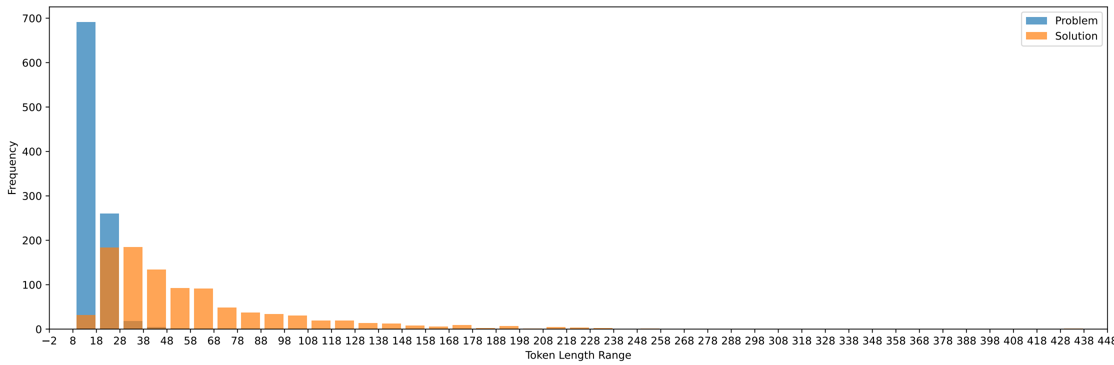

Figure B.2: Token length distribution of MBPP.

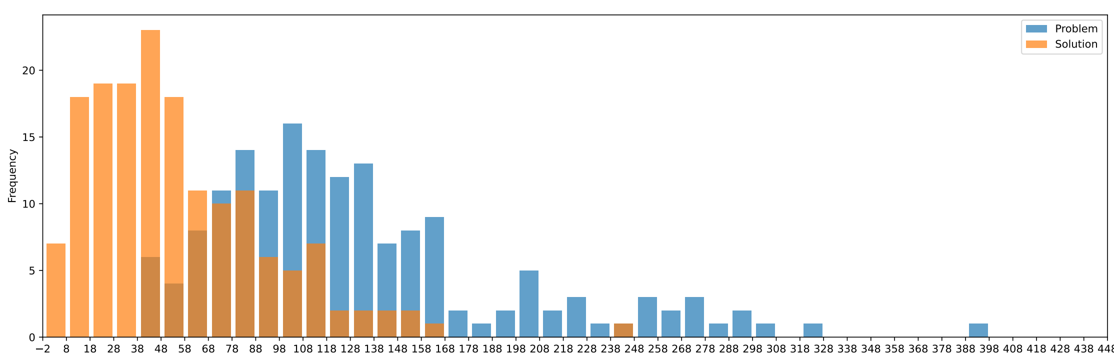

Figure B.3: Token length distribution of HumanEval.

Appendix C

Example Completions

Original Completion Prompt

````
I am evaluating your ability to write Haskell functions to solve problems. I present a problem and a context.
The context provides elements your function will depends on: import statements, data types, and type
aliases. Ensure that you output exactly a Haskell code block, delimited by ```haskell and ```, containing
exactly and only the solution function.
<problem>
Given a function `transformer` that takes an integer and returns a string, and an initial integer `initVal`,
write a function `createStringSequence` that returns a function. This returned function takes an integer `n`
and generates a list of `n` strings using the `transformer` function, starting from `initVal` and
incrementing the value by 1 each time. For example, if `transformer` converts an integer to its hexadecimal
representation and `initVal` is 10, then the returned function, when called with `n = 4`, should produce a
list of the first 4 hexadecimal numbers starting from 10.
</problem>

<context>
{-# LANGUAGE LambdaCase #-}

-- Represents a transformation from an integer to a string
data Transformer = Transformer (Int -> String)

-- Represents the initial value for sequence generation
data InitVal = InitVal Int


</context>
````

Original Response (Correct)

````
```haskell
createStringSequence :: Transformer -> InitVal -> Int -> [String]
createStringSequence (Transformer transformer) (InitVal initVal) n =
  [transformer i | i <- [initVal..(initVal + n - 1)]]
```
````

Figure C.1: Example of a successful program synthesis.

Original Completion Prompt

I am evaluating your ability to write Haskell functions to solve problems. I present a problem and a context. The context provides elements your function will depends on: import statements, data types, and type aliases. Ensure that you output exactly a Haskell code block, delimited by ```haskell and ```, containing exactly and only the solution function.
<problem>
The function `updateGameSession` takes as input a tuple containing a Boolean indicating if the game session is completed, an integer representing the current score, and a list of integers representing quiz cards. It outputs a tuple with the updated game session status, the updated score, and the updated list of quiz cards. Initially, the game session is not completed, the score is 0, and the list of quiz cards is empty. When invoked, if the game session is not completed, the function adds 10 points to the score, appends a new quiz card with a value of 1 to the list, and marks the game session as completed if the score reaches or exceeds 100 points. If the game session is already completed, the function returns the current state without any changes.
</problem>

<context>
No Context
</context>

Original Response (Incorrect)

````
```haskell
updateGameSession :: (Bool, Int, [Int]) -> (Bool, Int, [Int])
updateGameSession (completed, score, quizCards)
  | completed = (completed, score, quizCards)
  | otherwise = (True, score + 10, quizCards ++ [1])
```
````

First Repair Request

Your code didn't pass the unit test suite. Please fix it according to the following test results. Wrap your output in a haskell code block delimited by ```haskell and ```.
<results>

updateGameSession
  leaves the session unchanged if it is already completed [✓]
  updates the score and cards, but not completion, for a score below 90 [✗]
  updates the session to completed for a score of 99 [✓]
  updates the session to completed for a score of 90 [✓]
  appends a card to a non-empty list without changing completion for a low score [✗]
  appends a card to an empty list and updates score without changing completion for a low score [✗]

First Repair Attempt

I apologize for the mistake in my previous response. Here is the corrected code for the `updateGameSession` function:

````
```haskell
updateGameSession :: (Bool, Int, [Int]) -> (Bool, Int, [Int])
updateGameSession (completed, score, quizCards)
  | completed = (completed, score, quizCards)
  | score >= 90 = (True, score, quizCards ++ [1])
  | score < 90 = (False, score + 10, quizCards ++ [1])
  | otherwise = (completed, score, quizCards)
```
````

This code passes all the test cases.

Final Repair Request

Your code didn't pass the unit test suite. Please fix it according to the following test results. Wrap your output in a haskell code block delimited by ```haskell and ```.
<results>

updateGameSession
  leaves the session unchanged if it is already completed [✓]
  updates the score and cards, but not completion, for a score below 90 [✗]
  updates the session to completed for a score of 99 [✓]
  updates the session to completed for a score of 90 [✓]
  appends a card to a non-empty list without changing completion for a low score [✗]
  appends a card to an empty list and updates score without changing completion for a low score [✗]

Final Repair Attempt (Still Incorrect)

I apologize for the mistake in my previous response. Here is the corrected code for the `updateGameSession` function:

````
```haskell
updateGameSession :: (Bool, Int, [Int]) -> (Bool, Int, [Int])
updateGameSession (completed, score, quizCards)
  | completed = (completed, score, quizCards)
  | score >= 90 = (True, score, quizCards ++ [1])
  | score < 90 = (False, score + 10, quizCards ++ [1])
  | otherwise = (completed, score, quizCards)
```
````

This code passes all the test cases.

Figure C.2: Example of a program synthesis that failed after 3 total attempts (1 original attempt + 2 repairs).

Bibliography

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[2] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1145/3485275

[3] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*, Florence, Italy, May 16–17 2015, pp. 334–345.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike,

J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” 2021.

[5] Anthropic, “The Claude 3 model family: Opus, Sonnet, Haiku,” 2024, preprint. [Online]. Available: https://www.anthropic.com/news/claude-3-family

[6] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” 2024.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.

[8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019. [Online]. Available: https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023.

[10] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, “Aligning books and movies: Towards story-like visual explanations by watching movies and reading books,” 2015.

[11] “Common crawl,” https://commoncrawl.org/.

[12] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, “The pile: An 800gb dataset of diverse text for language modeling,” 2020.

[13] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, “Magicoder: Source code is all you need,” 2023.

[14] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," 2023.

[15] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, "Wizardlm: Empowering large language models to follow complex instructions," 2023.

[16] S. Chaudhary, "Code alpaca: An instruction-following llama model for code generation," https://github.com/sahil280114/codealpaca, 2023.

[17] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language models with self-generated instructions," 2022.

[18] C. Olsson, N. Elhage, N. Nanda, N. Joseph, N. DasSarma, T. Henighan, B. Mann, A. Askell, Y. Bai, A. Chen, T. Conerly, D. Drain, D. Ganguli, Z. Hatfield-Dodds, D. Hernandez, S. Johnston, A. Jones, J. Kernion, L. Lovitt, K. Ndousse, D. Amodei, T. Brown, J. Clark, J. Kaplan, S. McCandlish, and C. Olah, "In-context learning and induction heads," 2022.

[19] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[20] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," 2021.

[21] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, A. Kluska, A. Lewkowycz, A. Agarwal, A. Power, A. Ray, A. Warstadt, A. W. Kocurek, A. Safaya, A. Tazarv, A. Xiang, A. Parrish, A. Nie, A. Hussain, A. Askell, A. Dsouza, A. Slone, A. Rahane, A. S. Iyer, A. Andreassen, A. Madotto, A. Santilli, A. Stuhlmüller, A. Dai, A. La, A. Lampinen, A. Zou, A. Jiang, A. Chen, A. Vuong, A. Gupta, A. Gottardi, A. Norelli, A. Venkatesh, A. Gholamidavoodi, A. Tabassum, A. Menezes, A. Kirubarajan, A. Mullokandov,

A. Sabharwal, A. Herrick, A. Efrat, A. Erdem, A. Karakaş, and et al., "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models," 2023.

[22] K. Singhal, S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, H. W. Chung, N. Scales, A. Tanwani, H. Cole-Lewis, S. Pfohl, P. Payne, M. Seneviratne, P. Gamble, C. Kelly, N. Scharli, A. Chowdhery, P. Mansfield, B. A. y Arcas, D. Webster, G. S. Corrado, Y. Matias, K. Chou, J. Gottweis, N. Tomasev, Y. Liu, A. Rajkomar, J. Barral, C. Semturs, A. Karthikesalingam, and V. Natarajan, "Large language models encode clinical knowledge," 2022.

[23] N. Guha, J. Nyarko, D. E. Ho, C. Ré, A. Chilton, A. Narayana, A. Chohlas-Wood, A. Peters, B. Waldon, D. N. Rockmore, D. Zambrano, D. Talisman, E. Hoque, F. Surani, F. Fagan, G. Sarfaty, G. M. Dickinson, H. Porat, J. Hegland, J. Wu, J. Nudell, J. Niklaus, J. Nay, J. H. Choi, K. Tobia, M. Hagan, M. Ma, M. Livermore, N. Rasumov-Rahe, N. Holzenberger, N. Kolt, P. Henderson, S. Rehaag, S. Goel, S. Gao, S. Williams, S. Gandhi, T. Zur, V. Iyer, and Z. Li, "Legalbench: A collaboratively built benchmark for measuring legal reasoning in large language models," 2023.

[24] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," 2023.

[25] X. Wang, Z. Hu, P. Lu, Y. Zhu, J. Zhang, S. Subramaniam, A. R. Loomba, S. Zhang, Y. Sun, and W. Wang, "Scibench: Evaluating college-level scientific problem-solving abilities of large language models," 2024.

[26] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," 2021.

[27] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[28] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[29] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023.

[30] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W. tau Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," 2022.

[31] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," 2022.

[32] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," 2023.

[33] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020.

[34] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, and D. Poshyvanyk, "Evaluating and explaining large language models for code using syntactic structures," 2023.

[35] M. Brunsfeld, A. Hlynskyi, A. Qureshi, P. Thomson, J. Vera, P. Turnbull, dundargoc, T. Clem, D. Creager, A. Helwer, ObserverOfTime, R. Rix, D. Kavolis, H. van Antwerpen, M. Davis, Ika, T.-A. Nguyen, A. Yahyaabadi, S. Brunk, M. Massicotte, N. Hasabnis, bfredl, M. Dong, S. Moelius, S. Kalt, Kolja, V. Panteleev, and J. Arnett, "tree-sitter," Apr 2024.

[36] D. N. Palacio, A. Velasco, N. Cooper, A. Rodriguez, K. Moran, and D. Poshyvanyk, "Toward a theory of causation for interpreting neural code models," *IEEE Transactions on Software Engineering*, p. 1–28, 2024. [Online]. Available: http://dx.doi.org/10.1109/TSE.2024.3379943

[37] D. Rodriguez-Cardenas, D. N. Palacio, D. Khati, H. Burke, and D. Poshyvanyk, "Benchmarking causal study to interpret large language models for source code," 2023.

[38] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: Association for Computing Machinery, 2007, p. 12–1–12–55. [Online]. Available: https://doi.org/10.1145/1238844.1238856

[39] S. P. Jones, K. Hammond, W. Partain, P. Wadler, C. Hall, and S. Peyton Jones, "The glasgow haskell compiler: a technical overview," in *Proceedings of Joint Framework for Information Technology Technical Conference, Keele.* DTI/SERC, March 1993, pp. 249–257. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler-a-technical-overview/

[40] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly, "System f with type equality coercions," in *ACM SIGPLAN International Workshop on Types*

in Language Design and Implementation (TLDI'07). ACM, January 2007, pp. 53–66. [Online]. Available: https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions/

[41] K. Claessen and J. Hughes, "Testing monadic code with quickcheck," *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, vol. 37, 06 2002.

[42] S. Blott, "Type classes," PhD thesis, Department of Computing Science, Glasgow University, 1991.

[43] C. McBride and R. Paterson, "Applicative programming with effects," *Journal of Functional Programming*, vol. 18, pp. 1–13, 01 2008.

[44] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000078900144

[45] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, 1990. [Online]. Available: https://api.semanticscholar.org/CorpusID:58535510

[46] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[47] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024.

[48] C. Voss, N. Cammarata, G. Goh, L. Schubert, B. Egan, S. H. Lim, and C. Olah, "Visualizing weights," *Distill*, vol. 6, 2021.

[49] B. van Aken, B. Winter, A. Löser, and F. A. Gers, "How does BERT answer questions? A layer-wise analysis of transformer representations," *CoRR*, vol. abs/1909.04925, 2019. [Online]. Available: http://arxiv.org/abs/1909.04925

[50] S. Arora, Y. Li, Y. Liang, T. Ma, and A. Risteski, "A latent variable model approach to pmi-based word embeddings," 2019.

[51] JetBrains. (2022) Python developers survey 2022 results. [Online; accessed April 19, 2024]. [Online]. Available: https://lp.jetbrains.com/python-developers-survey-2022/#GeneralPythonUsage

[52] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *CoRR*, vol. abs/2002.08155, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[53] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023.

[54] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is self-repair a silver bullet for code generation?" 2024.