Benchmark-F: Towards Sound Evaluation of Program Logic Understanding with System F

Anonymous ACL submission

Abstract

Large language models (LLMs) are increasingly integrated into software engineering. There is limited discussion on addressing the semantic gap between natural languages and programming languages when evaluating LLMs for code. To address this, we introduce Benchmark-F, a novel benchmark designed to evaluate LLMs' capabilities in software engineering by focusing on type inference. Type inference is a task that requires understanding code semantics purely from program logic. By predicting type signatures from function implementations without natural language cues, Benchmark-F assesses models' intrinsic code comprehension and logical reasoning essential for software engineering and security applications. Natural language elements in Benchmark-F can be removed through verified transformations, making it purely a programming language evaluation platform to help mitigate the semantic gap. Our evaluation reveals that even state-of-the-art LLMs face significant limitations in this context, with the best closed-source model (Claude-3.5-Sonnet) achieving 69.05% accuracy and the top open-source model (Llama3.1-405B) reaching 60.71%, highlighting a critical gap in current models' abilities to comprehend and reason about code. By isolating code understanding from natural language elements, Benchmark-F_{pure} advances the assessment of LLMs in practical software development tasks, promoting the development of more robust models for software-related tasks. Our code and benchmark will be publicly available.

1 Introduction

The ability to understand and write programs has become an important factor to evaluate the intelligence of large language models (LLMs) (Achiam et al., 2024; Jiang et al., 2023; Llama Team, 2024; Team, 2024c). However, the popular tasks in code understanding lack the ability to reveal the prowess of LLMs fully in understanding program flows and logic. Code-to-text, a common task category for evaluating program understanding abilities of language models (Alon et al., 2019b; Lu et al., 2021; Ahmed and Devanbu, 2022), includes code summarization and function name prediction. These tasks involve generating natural language (NL) descriptions from programming language (PL) code, and predicting function names based on their implementations. Despite the complexity of these benchmarks, they fail to evaluate how well LLMs understand the semantics of code.

NL elements within code, such as comments and the name of identifiers, assist programmers in reading code. However, they do not affect the semantics of the code. For example, removing comments and normalizing identifier names do not change the semantics of the program. Ahmed and Devanbu (2022) showed that the performance of language models on these benchmarks can be heavily influenced by NL elements within code snippets. AdvTest (Lu et al., 2021) proposed normalizing function and variable names to enhance the assessment of understanding and generalization capabilities of models, though it is only for text-to-code tasks such as code generation or code search.

Traditional semantics (e.g., denotational, operational, and axiomatic) of programming languages describe program behavior, as viewed by the compiler. They are invariant to the NL components of programs. In contrast, a programmer's understanding of a program's behavior is influenced considerably by its NL components. We call this the *cognitive semantics* of a program. The gap between these two types of semantics often leads to bugs and security vulnerabilities. We are unaware of any benchmark for evaluating LLMs' understanding of traditional semantics.

Understanding traditional semantics is important in software engineering and security. For example, clone detection can be used to detect plagiarized

programs (Crussell et al., 2014) or to find programs that contain a library that was just revealed to be vulnerable or malicious. However, often the target programs are obfuscated either by developers to protect intellectual properties or by attackers to deter security analysis (Crussell et al., 2015). Program obfuscation transforms a program into a new version that preserves its original functionality while appearing significantly different (Barak et al., 2012). Recent work (Zhao et al., 2023; Huang et al., 2024) detects clones but overlooks code obfuscation. Since LLMs that can understand traditional semantics are naturally immune to obfuscation, they can be used for robust clone detection. Therefore, we need a benchmark for evaluating LLMs' abilities to understand traditional semantics.

As a first step, we use type inference, or predicting the type signature from function implementations, as a task in understanding traditional semantics for several reasons. First, the ability to perform type inference indicates the intrinsic understanding of the program. The type signature of a function acts as a succinct summary of its functionality (Pierce, 2002). Therefore, predicting type signatures is close to, and perhaps surpasses, the task of predicting function names, which was introduced as "extreme summarization of functions" by Allamanis et al. (2016). Pierce further notes that type declarations serve as a form of documentation that hints at the function's behavior, underscoring the usage of generating type signatures as code summarization. Therefore, using type inference as a code understanding task is natural and intuitive for evaluating LLMs.

Moreover, the results of removing NL elements from type inference tasks can be easily verified. Type inference is based on symbolic logic, where substituting one type name for another does not alter the semantics. In fact, type synonyms (Chakravarty et al., 2005) serve the exact purpose. The compiler serves as an oracle to verify the NL-free pure-PL tasks are semantically correct. To assess the semantics of the program is not altered, we can compile the candidate tasks into abstract syntax trees, and compare the structure of these trees. As a result, we argue that type inference serves as a robust task towards the soundness of evaluating code understanding.

In addition to the advantages in code understanding and verifiability, type inference also serves as a valuable benchmark for assessing language models' understanding of symbolic logic, particularly in their deductive reasoning abilities. Unlike simple arithmetic or geometry problems found in popular math datasets (Hendrycks et al., 2021; Cobbe et al., 2021), real mathematical reasoning involves making deductive claims from axioms (Paseau and Pregel, 2023). Mathematicians and logicians (Russell, 2020; Hilbert, 2019; Frege, 1976), view mathematics fundamentally as a logical deduction. This deductive nature is also recognized in the philosophy. Curry (1951) argued that mathematics are to be understood deductively from axioms and inference rules. These insights led to the development of the Curry-Howard Isomorphism (Curry, 1934; Howard et al., 1980), also called propositions-as-types (Wadler, 2015), linking mathematical reasoning with type systems in programming languages (known as the Hindley-Milner type system (Hindley, 1969; Milner, 1978)). Therefore, we argue that type inference, as an instance of natural deduction (Prawitz, 2006), is also suited to evaluate LLMs' reasoning capabilities.

We present Benchmark-F, a novel evaluation benchmark for program understanding and reasoning. Benchmark-F is constructed from function-level type inference in Haskell (Jones, 2003), whose core language's type system is based on System F (Girard, 1986), (also called polymorphic λ-calculus (Reynolds, 1974) from Curry-Howard Isomorphism). System F is parametric polymorphic, meaning that our Benchmark-F contains not only atomic types (for example, `Int`, `Bool`, etc) but also type variables, ensuring the diversity of tasks in the benchmark. Taking advantage of type synonyms in Haskell, we can easily remove NL-related pieces from the benchmark, making it sound to evaluating program logic and structure understanding without the influence from non-code data. Type inference is a form of deductive mathematical reasoning, which makes Benchmark-F also applicable to evaluating LLM's intelligence in reasoning. To sum up, we make the following contributions

1. We introduce Benchmark-F, a novel benchmark containing 84 tasks, which leverage type inference to assess LLM capabilities in code understanding and reasoning, advancing the field of NLP methods for program analysis.

2. We conduct a comprehensive evaluation of state-of-the-art LLMs. Our findings indicate that the leading API-based LLM (Claude-3.5-Sonnet) achieves 69.05% accuracy, while

the best open-source model (Llama3.1-405B) reaches 60.71%.

3. We develop Benchmark-F_{pure}, a variant of Benchmark-F which is devoid of natural language components, to isolate LLM evaluation based solely on code and logical reasoning. We observe an approximate 65% relative performance drop in state-of-the-art LLMs, Claude-3.5-Sonnet and GPT-4o.

2 Design of Benchmark-F

Benchmark-F is a benchmark embodying *type inference* in System F (Girard, 1986). The task is to generate the final type signature of a function given on its implementation and the type signatures of related functions.

2.1 Benchmark Construction

Haskell is a function programming language with higher-order functions and static polymorphic typing (Marlow et al., 2010). Type inference in Haskell exemplifies the Hindley-Milner type system (Hudak et al., 2007), which was discovered independently by Hindley and Milner to introduce parametric polymorphism into combinatory logic and programming. Type inference in Haskell is sound and complete (Hudak et al., 2007), but inferring the results are context-sensitive. The implementation of a function always depends on other functions via composition. Therefore, construction of this benchmark at the single-function level, similar to other prevalent benchmarks for code Understanding (Chen et al., 2021; Austin et al., 2021; Husain et al., 2020; Lu et al., 2021), is not sensible. Furthermore, the entire type system in Haskell is too complex even for programmers to understand without special training. Haskell supports ad-hoc polymorphism via type classes (Wadler and Blott, 1989) where the polymorphic type variable can be constrained. Haskell also supports higher-rank types (Jones et al., 2007). For evaluation purposes, we will not include these complex inference rules in our benchmark. Putting it together, to build self-contained type inference tasks, we build a three-stage pipeline as described in Figure 1.

Package selection and function extraction. We select the standard Haskell Prelude (Jones, 2003) to build Benchmark-F. All functions in Prelude are defined in the ghc-internal (Committee, 2024a) library, so there are no external dependencies. Therefore, all tasks in Benchmark-F are complete and self-contained, i.e. all tasks contain the type signatures needed to infer the correct result.

We then parse all the code files into abstract syntax trees (ASTs) using tree-sitter (Brunsfeld, 2024) and extract all type signatures and function definitions from the ASTs. Since the Haskell function's type signature and implementation are not tied to one AST, we match the available type signatures and corresponding implementation together. Since declaration style pattern matching is common in Haskell, each case pattern is parsed individually. We concatenate all extracted function definitions together as the whole implementation if they have the same function name and are from the same file. Since Haskell evaluates pattern matching top-to-bottom and left-to-right, we perform this concatenation by the order of post-order tree traversal from the root of the AST. This stage ensures the completeness of all tasks in Benchmark-F to have a full valid implementation and checked ground truth.

Type Filtering. Since some of the advanced type rules are overly complicated for language models, we set up a filter to keep only monomorphic or parametric polymorphic functions. To be clearer, we only allow atomic types, structured atomic types, and type variables. We provide an example for each major polymorphic type in Figure 5. From the logic perspective, Benchmark-F only focuses on first-order logic.

Dependencies solving. For each candidate task, we perform a dependency search. This concept of type dependencies is similar to assumptions in proof theory and logic. For instance, in proving the proposition $A \wedge B$, we require the assumption that A is true and B is true. To address this, we first extract all function application nodes from the AST and then search for their corresponding type signatures using Hoogle, the Haskell API search engine (Mitchell, 2024). The type signatures returned by Hoogle are prepended to the task, constructing the desired prompt. This step ensures that all tasks are self-contained and form a closed logical chain to evaluate the LLMs.

Removing natural language from tasks. The Haskell Prelude was introduced by Jones before the knowledge cutoff of all models. The `ghc-internal` (Committee, 2024a) package is part of the Glasgow Haskell Compiler GitLab repository(Committee, 2024b), which has been open-

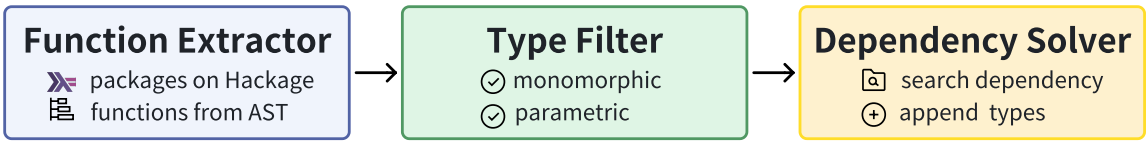

Figure 1: Pipeline to construct Benchmark-F. Task instances in Benchmark-F are created from monomorphic or parametric functions with the required type dependencies provided.

source since 2018. Therefore, we assume that all models have been trained on source data relevant to our benchmark tasks. To address potential contamination, we developed a type rewrite algorithm, as outlined in Algorithm 1. This algorithm operates on a per-task basis, replacing types that contain natural language with their NL-free counterparts.

In summary, all type variables are rewritten as letters tokens from 'a' to 'z', while type names are mapped to uppercase letters 'A' to 'Z'. Although the possibility exists of encountering more than 26 types in a given task, such a case has not arisen in Benchmark-F. Function names and variable names are standardized as f_i and v_i, respectively, where i denotes their order of occurrence.

By applying this algorithm to all tasks, we construct Benchmark-F_{pure}, allowing us to evaluate the LLMs based solely on logical reasoning, free from natural language interference and data leakage. We demonstrate examples of tasks before and after applying this rewrite in Figure 2. For the threats to validity, we verify all the rewritten tasks in Benchmark-F_{pure} by compiling. After applying our rewrite algorithm, every task in Benchmark-F_{pure} compiles into an AST that has the same structure as the corresponding tasks in Benchmark-F.

2.2 Task Formulation

Model input. The input prompt is divided into three components: the system prompt, the instruction prompt, and the task prompt. We use the same system and instruction prompts, as depicted in Figure 6, to guide the models toward a successful generation. These prompts remain consistent across all models. For GPT and Ollama-based models, the system prompt is sent using the 'system' role. However, since the Anthropic API for Claude models does not support the 'system' role, we use the 'assistant' role instead.

The task prompt is similar to the examples in Figure 4, but consists of real functions defined in the Haskell Prelude (Jones, 2003). As outlined

```
not :: Bool -> Bool
(.) :: (b -> c) -> (a -> b) -> a -> c
span :: (a -> Bool) -> [a] -> ([a], [a])

break p =  span (not . p)
- complete the following type signature for
    `break`
break :: (a -> Bool) -> [a] -> ([a],[a])
```

Listing (1) Before type-rewrite

```
f1 :: A -> A
(.) :: (b -> c) -> (a -> b) -> a -> c
f2 :: (a -> A) -> [a] -> ([a], [a])

f3 v1 =  f2 (f1 . v1)
- complete the following type signature for `f3`
f3 :: (a -> A) -> [a] -> ([a],[a])
```

Listing (2) After type-rewrite

Figure 2: Example of the `break` task before and after type-rewrite.

in subsection 2.1, we extract monomorphic and parametric polymorphic functions from Prelude to construct the tasks, and use Hoogle to resolve their type dependencies. For each extracted function, we concatenate its dependencies, separated by new line symbols, and prepend this to the function definition. Following established work (Mishra et al., 2022; Chen et al., 2023; He et al., 2024), we add an instructive comment for the models to predict the type of the implemented function. Finally, we append a task-specific hook in the form of `'function_name' ::` to the end of the input prompt, providing a clear starting point for the next token prediction, improving the likelihood of successful code completion.

Result evaluation. Since all ad-hoc polymorphisms have been removed, the argument for checking whether the generated type matches up to parametric by Peng et al. (2023) no longer applies. The result of type inference is always deterministic, meaning that the same input consistently produces the same output. Thus, we can evaluate the predictions through simple string matching. Therefore, we assess all results based on accuracy by compar-

ing the exact match with the ground truth.

3 Experiments and Results

3.1 Experimental Setup

Compared models. Our objective is to evaluate the performance of both open-source and API-based models across different sizes. For open-source models, we further categorize them based on whether they have been fine-tuned for code-related tasks. Therefore, the models are grouped into three main categories: API-based commercial LLMs, open-source (OSS) LLMs, and code-specific LLMs. For API-based models, we select the GPT series from OpenAI and Claude models from Anthropic. In the OSS category, we include several widely used models of varying sizes, such as Microsoft's Phi-3 (Abdin et al., 2024), Mistral (Jiang et al., 2023) and its mixture-of-experts variant Mixtral (Jiang et al., 2024), Google's Gemma (Team, 2024c,b), Alibaba's Qwen (Yang et al., 2024), DeepSeek (DeepSeek-AI, 2024), and Meta's Llama (Llama Team, 2024). When multiple versions of a model exist (e.g., Llama3 and Llama3.1), we include both versions as possible, ensuring consistent comparison across similar parameter sizes. For code-specific models, we focus on the code versions of the aforementioned OSS models. These include CodeGemma (Team, 2024a), Codestral (Mistral-AI, 2024), CodeLlama (Roziere et al., 2023), and DeepSeek-Coder (Zhu et al., 2024). Additionally, we evaluate Granite-Coder (Mishra et al., 2024), which is specifically designed for code-related tasks. All these models are available in various parameter sizes and versions, as detailed in Table 4. For inference on OSS models, we utilize Ollama (Ollama, 2024) to set up the environment.

We evaluate all models in a zero-shot setting. To maintain the consistency and determinism in the generation results, we set the inference hyperparameters to temperature = 0 and top_p = 1 for all models, except for the Claude models. Unlike OpenAI's and Ollama's APIs, Anthropic's APIs do not allow explicit configuration of these hyperparameters. To address potential threats to validity, we repeated the Claude experiments ten times and observed identical results across all runs.

Post-processing. We apply straightforward post-processing strategies to the generation results. First, since the `String` type in Haskell is an alias for `[Char]`, we replace occurrences of `[Char]` with `String` in both the generated responses and the ground truth. Second, some model outputs are enclosed in markdown code blocks. so we remove the top and bottom markdown delimiters for consistency. Third, we observe that when a hook (for example, `xs ::` as in the last line in Figure 4) is provided in the prompt, some models continue from the hook and generate only the type signature, while others repeat the hook in their response. To standardize the outputs, we remove all instances of the hook from both the ground truth and the model outputs. In general, we avoid complex post-processing and focus on resolving basic formatting issues. We also notice that some models, such as CodeLlama-13B, ignore our instructions and produce explanations of the input prompt instead of performing type inference. In these cases, we take the generated results directly for evaluation.

Model	Version	#Correct	Acc (%)
gpt-3.5-turbo	0125	42	50.00
gpt-4o-mini	2024-07-18	43	51.19
gpt-4-turbo	2024-04-09	48	57.14
gpt-4o	2024-05-13	**50**	**59.52**
claude-3-haiku	2024-03-07	29	34.52
claude-3-sonnet	2024-02-29	30	35.71
claude-3-opus	2024-02-29	40	47.62
claude-3.5-sonnet	2024-06-20	**58**	**69.05**

Table 1: Evaluation results of API-based LLMs on Benchmark-F. "Acc" is the abbreviation of accuracy.

3.2 Evaluation on Benchmark-F

We evaluated 45 models on the Benchmark-F benchmark, ranging from closed-source API-based models to open-source models and their code generation versions. Table 1 and Table 2 present the performance of API-based models and selected OSS models. For the full evaluation results, please refer to Table 4. In this section, we discuss our key findings and provide further insights.

Who are the best performers on Benchmark-F? In our evaluation, the highest-performing closed-source API model is Claude-3.5-Sonnet with 69.05% accuracy, outperforming its closest competitor, GPT-4o with 59.52% accuracy, by an absolute margin of 9.53%. Among open-source models on Benchmark-F, Llama3.1-405B Llama Team achieves the best performance, correctly solving 60.71% of the deductive type inference tasks. Notably, we find that as the most advanced OSS model, Llama3.1-405B delivers comparable, if

not surpassing, quality to leading API-based models such as GPT-4o and GPT-4-turbo, following similar trends as the evaluation results on other code and math datasets released by Llama Team. The nearest open-source competitor is DeepSeek-V2.5-236B (DeepSeek-AI, 2024) with 54.76% accuracy. In the code generation category, DeepSeek-Coder-V2-236B (Zhu et al., 2024) demonstrates superior performance with 50% accuracy, followed closely by Granite-Code-34B (Mishra et al., 2024), which achieves 47.62% accuracy, despite having a significantly smaller parameter size. Surprisingly, Qwen2.5-Coder-7B achieves 46.43% accuracy, which is comparable to that of much larger models, such as Granite-Code-34B, DeepSeek-Coder-V2-236B, and the foundation models GPT-4o and GPT-4-turbo.

Do newer versions of the same model perform better? During our experiments, the existence of "sub-versions" of models drew our attention. Therefore, we selected open-source models with different sub-versions to assess their improvements on Benchmark-F. For Qwen2 (Yang et al., 2024), Llama3 (Llama Team, 2024), DeepSeek-V2 (DeepSeek-AI, 2024), and their respective sub-versions, we observed consistent improvements on Benchmark-F, similar to their performance on other benchmarks. The major advancement from Llama3 to Llama3.1 is the increase in context length from 8K to 128K (Llama Team, 2024). While our type inference tasks are token-efficient, we find that the extended pre-training context length significantly enhances the model's deductive reasoning capabilities. In the case of Qwen2, the transition to Qwen2.5 involves a substantial increase in the volume of code and math data used for pre-training, which results in a notable performance improvement on Benchmark-F. Similarly, DeepSeek indicates that the difference between DeepSeek-V2 and V2.5 is also the increase in context length to 128K, supporting our observation that a longer pre-training context benefits logical reasoning in token-efficient tasks, particularly those involving code and math.

Can fine-tuning on code improve accuracy on Benchmark-F? To this end, we investigate the performance differences between Gemma (Team, 2024c) and CodeGemma (Team, 2024a), as well as between DeepSeek-V2 (DeepSeek-AI, 2024) and DeepSeek-Coder-V2 (Zhu et al., 2024). Our findings show that both the 16B and 236B versions of DeepSeek-Coder-V2 significantly outperform their respective base models. In particular, the improvement is more pronounced in smaller models. For example, DeepSeek-Coder-V2-16B achieves twice the accuracy of its base model, while the 236B version shows a relatively modest gain of 10.52% in accuracy. The same trend is also observed on the Qwen2.5-Coder, where the smaller code model improved substantially compared to the large model. In the case of CodeGemma, the smaller 2B version also doubles its performance, improving from an accuracy of 2.38% to 4.76%. However, the larger CodeGemma-7B, with an accuracy of 22.61%, performs worse compared to its base model Gemma-7B. The experiments show that while fine-tuning on code-specific datasets can enhance the performance of some LLMs, this effect is not consistent across all models. Fine-tuning on code does not reliably guarantee improved performance on Benchmark-F.

Model	Size	#Correct	Acc (%)
Gemma	2B	2	2.38
	7B	22	26.19
Qwen2	1.5B	5	5.95
	7B	21	25.00
	72B	34	40.48
Qwen2.5	1.5B	18	21.43
	7B	35	41.67
	72B	43	51.19
DeepSeek-V2	16B	16	19.05
	236B	38	45.24
DeepSeek-V2.5	236B	46	54.76
Llama3	8B	22	26.19
	70B	31	36.90
Llama3.1	8B	28	33.33
	70B	40	47.62
	405B	**51**	**60.71**
CodeGemma	2B	4	4.76
	7B	19	22.61
Granite-Code	2B	6	7.14
	8B	15	17.86
	20B	23	27.38
	34B	40	47.62
DeepSeek-Coder-V2	16B	34	40.48
	236B	**42**	**50.00**

Table 2: Evaluation results of selected OSS LLMs on Benchmark-F. The complete evaluation is at Table 4.

3.3 Evaluation on Benchmark-F_{pure}

Since the Benchmark-F_{pure} version replaces the natural language elements with meaningless identifiers, it poses a significantly greater challenge

Model	Version	#Correct	Acc (%)
gpt-3.5-turbo	0125	10	11.90
gpt-4o-mini	2024-07-18	11	13.10
gpt-4o	2024-05-13	18	21.43
gpt-4-turbo	2024-04-09	**21**	**25.00**
claude-3-haiku	2024-03-07	8	9.52
claude-3-sonnet	2024-02-29	2	2.38
claude-3-opus	2024-02-29	10	11.90
claude-3.5-sonnet	2024-06-20	**20**	**23.81**

Table 3: Evaluation results of API-based LLMs on Benchmark-F_{pure}.

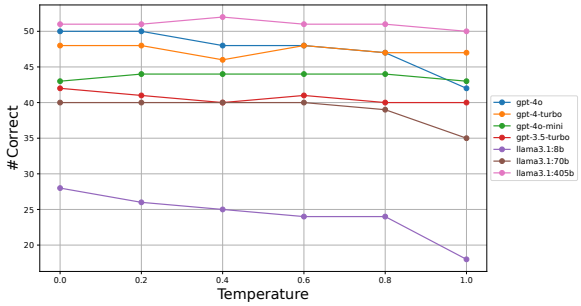

Figure 3: Performance changes of GPT and Llama3.1 models on different generation temperatures. "#Correct" is the number of correct generations out of 84 tasks in Benchmark-F.

for LLMs compared to the base version of Benchmark-F. Therefore, we limited our evaluation to the more advanced API-based models, specifically GPT and Claude. The evaluation results for Benchmark-F_{pure} are presented in Table 3. The top-performing model for Benchmark-F_{pure}, GPT-4-turbo, achieved an accuracy of only 25%. Claude-3.5-Sonnet, which was the best-performing model on the base version, ranked second on Benchmark-F_{pure}, with a 23.81% accuracy.

The substantial drop in performance highlights that LLMs' understanding of programs has become overly reliant on natural language elements. These natural language elements, such as variable and function names, are inherently meaningless to program execution, as the program's behavior remains unchanged regardless of how variables and functions are named. This finding is consistent with previous studies by Ahmed and Devanbu (2022), further emphasizing the necessity of Benchmark-F and Benchmark-F_{pure} to effectively and robustly assess the true reasoning capabilities of LLMs on code understanding.

3.4 Effects on Generation Temperature

We acknowledge potential biases in performance evaluation arising from the temperature hyperparameter of the models at inference time. To explore these effects, we conduct ablation studies on the GPT models and Llama3.1 models, as they are the most robust models in our evaluation that allow us to adjust the temperature.

We investigate the impact of generation temperature on type inference tasks in Benchmark-F by evaluating model performance across a range of temperatures $t = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$. Results for $t = 0.0$ are also provided in Table 1. As illustrated in Figure 3, the majority of models exhibit a decline in accuracy as the generation temperature increases. Among the models compared, the largest model Llama3.1-405B demonstrates the most consistent performance across different temperatures, maintaining the highest accuracy at all six levels of temperature. This suggests that reasoning tasks on programs demand less creative generation. Additionally, larger models appear to exhibit greater resilience to variations in generation temperature.

4 Related Work

4.1 Learning to Understand Programs

Large Language Models (LLMs) have made remarkable advances in the field of natural language processing (Vaswani et al., 2017; Devlin et al., 2019). To leverage the power of LLMs in software engineering (SE), earlier work has addressed the LLM's ability to learn code representation for a better understanding of programs. Code-related tasks for LLMs can be categorized into generation or synthesis tasks and understanding tasks. Code generation tasks (Roziere et al., 2023; Mishra et al., 2024; Team, 2024a; Mistral-AI, 2024; Zhu et al., 2024; He et al., 2024), let the model generate functioning code or testing code that can be executed. Understanding tasks include code classification (Zhao et al., 2023; Huang et al., 2024). code summarization (Iyer et al., 2016; LeClair et al., 2019; Ahmad et al., 2020), and method name prediction (Allamanis et al., 2016; Alon et al., 2019a,b; Ahmed and Devanbu, 2022), which tests the model's ability to comprehend complex program logic without writing code.

In software engineering, code summarization and method name prediction are particularly prevalent tasks for demonstrating the effectiveness of models. This is because both doc-strings and

method names are considered to provide a high-level summary of a function's purpose (Alon et al., 2019b; Ahmed and Devanbu, 2022). In practice, choosing good names for API functions and writing clear documentation are the essential steps of software refactoring (Fowler, 2018). However, using these tasks to evaluate code understanding cannot fully reveal the real intelligence of the model, as the results are easily influenced by the non-code natural language components in the data (Ahmed and Devanbu, 2022). In fact, programs behave the same even if all comments are removed and identifier names are normalized. In real case program analysis, for example, control flow analysis (Allen, 1970), data flow analysis (Fosdick and Osterweil, 1976), and compiler optimization (Lattner and Adve, 2004; Zhao et al., 2012), tools built by human experts analyze programs on sound representations of the same program with no natural language components.

4.2 Propositions as Types

The connection between logic and programming can be traced back to Curry (1934) and Howard et al. (1980), commonly known as the Curry-Howard Isomorphism, or *propositions as types*. As summarized by Wadler, the core idea is that each proposition in logic corresponds to a type in a programming language, and vice versa. For instance, proposition A implies a proposition B (denoted as $A \implies B$) corresponding to a function mapping from type A to type B (denoted as $A \to B$). Since implementing a function involves invoking other defined functions to transform inputs of specific types into an output of another (or the same) type, we can interpret these defined functions as logical assumptions. Then the process of implementing the function itself can be viewed as proof of the proposition. Thus, the concept of *proofs as programs* emerges. This paradigm has been widely applied in security areas like formal verification (Leroy, 2009) and automatic proving (Huet et al., 1997; Nipkow et al., 2002; Bertot and Castéran, 2013).

4.3 Neural Type Inference

With the advancement of ML models, the software engineering and programming languages communities have been attempting to utilize them for predicting type hints in DPLs to complement the limitations of rule-based type inference. There are three main approaches to neural type inference: supervised, cloze-style, and generative. Supervised type inference utilizes type annotations as labels to train models such as recurrent neural networks (Pradel et al., 2020), graph neural networks (Wei et al., 2020), BERT-based models (Jesse et al., 2021), and clustering methods (Mir et al., 2022), but it relies on high-quality labeled datasets. Cloze-style type inference use pre-trained fill-in-the-middle models like CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022), which learn from unlabeled code by filling masked type annotations (Huang et al., 2023). Generative type inference (Peng et al., 2023) uses static analysis to generate domain-aware chain-of-thought (COT) prompts that teach general-purpose LLMs to generate types via in-context learning.

As illustrated in Figure 2 and Figure 4, each task in Benchmark-F is composed of the implementation of a function and explicitly provides the type signatures of all other functions it invokes. This process, which we call 'dependency solving,' ensures a closed logical chain within each task, making our approach fundamentally different from previous neural type inference tasks (Pradel et al., 2020; Jesse et al., 2021; Mir et al., 2022; Peng et al., 2023). Prior approaches primarily treat type inference as a token prediction problem, focusing on inferring variable or return types based on function implementations alone, without incorporating the logical dependencies between functions. This negates the aforementioned idea of *propositions as types* and *proofs as programs*. In contrast, our method treats type inference as a reasoning process, where type dependencies are provided explicitly to solve the task. Relying solely on probability-based token prediction without considering known propositions is logically unsound.

5 Conclusion

In this paper, we present Benchmark-F, a novel benchmark designed to evaluate the ability of language models to understand program logic and reasoning. Benchmark-F focuses on type inference in System F, utilizing Haskell syntax and functions to derive symbolic deductive reasoning. This setup is well-suited to capture the natural elements of code understanding, offering a robust framework for logical reasoning. By expanding the scope of tasks related to reasoning and code understanding, Benchmark-F aims to inspire further research on logic-based approaches for evaluating state-of-the-art language models.

Limitation

We have not evaluated the latest OpenAI models, o1-preview and o1-mini. Access to these models' APIs requires a minimum spending threshold, which we have not yet met.

References

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *Preprint*, arXiv:2404.14219.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE 22. ACM.

AI@Meta. 2024a. Llama 3 model card.

AI@Meta. 2024b. Llama 3.1 model card.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2091–2100, New York, New York, USA. PMLR.

Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 119, New York, NY, USA. Association for Computing Machinery.

Uri Alon, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL).

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2012. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2).

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer Science & Business Media.

Max Brunsfeld. 2024. Tree-sitter.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. *SIGPLAN Not.*, 40(9):241253.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Haskell Core Libraries Committee. 2024a. ghc-internal: Basic libraries.

Haskell Core Libraries Committee. 2024b. The glasgow haskell compiler.

Jonathan Crussell, Clint Gibler, and Hao Chen. 2015. Andarwin: Scalable detection of Android application clones based on semantics. *IEEE Transactions on Mobile Computing*, 14(10).

Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. MAdFraud: Investigating ad fraud in Android applications. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*, Bretton Woods, NH, USA.

Haskell B Curry. 1934. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590.

Haskell B Curry. 1951. *Outlines of a formalist philosophy of mathematics*, volume 3. Elsevier.

DeepSeek. 2024. Deepseek official website.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *Preprint*, arXiv:1810.04805.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Lloyd D Fosdick and Leon J Osterweil. 1976. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330.

Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gottlob Frege. 1976. *Nachgelassene Schriften und Wissenschaftlicher Briefwechsel. Zweiter Band: Wissenschaftlicher Briefwechsel*, volume 2. Felix Meiner Verlag.

Jean-Yves Girard. 1986. The system f of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. Unitsyn: A large-scale dataset capable of enhancing the prowess of large language models for program testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 10611072, New York, NY, USA. Association for Computing Machinery.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

David Hilbert. 2019. Mathematical problems. In *Mathematics*, pages 273–278. Chapman and Hall/CRC.

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60.

William A Howard et al. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490.

Jiabo Huang, Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2024. Code representation pre-training with complements from program executions. In *Empirical Methods in Natural Language Processing: Industry Track (EMNLP)*.

Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 1211255, New York, NY, USA. Association for Computing Machinery.

Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. 1997. The coq proof assistant a tutorial. *Rapport Technique*, 178.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 14831486, New York, NY, USA. Association for Computing Machinery.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *Preprint*, arXiv:2401.04088.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(1):1–82.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA. IEEE Computer Society.

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

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107115.

AI@Meta Llama Team. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Simon Marlow et al. 2010. Haskell 2010 language report.

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375.

Amir M. Mir, Evaldas Latokinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE 22. ACM.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. 2024. Granite code models: A family of open foundation models for code intelligence. *Preprint*, arXiv:2405.04324.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. 2022. Reframing instructional prompts to GPTk's language. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 589–612, Dublin, Ireland. Association for Computational Linguistics.

Mistral-AI. 2024. Codestral: Hello, world! empowering developers and democratising coding with mistral ai.

Neil Mitchell. 2024. Haskell api search engine.

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

Ollama. 2024. [link].

Alexander Paseau and Fabian Pregel. 2023. Deductivism in the Philosophy of Mathematics. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*, Fall 2023 edition. Metaphysics Research Lab, Stanford University.

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative type inference for python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 988–999. IEEE.

Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 209220, New York, NY, USA. Association for Computing Machinery.

Dag Prawitz. 2006. *Natural deduction: A proof-theoretical study*. Courier Dover Publications.

John C Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, pages 408–425. Springer.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Bertrand Russell. 2020. *Principles of mathematics*. Routledge.

CodeGemma Team. 2024a. Codegemma: Open code models based on gemma. *Preprint*, arXiv:2406.11409.

Gemma Team. 2024b. Gemma 2: Improving open language models at a practical size. *Preprint*, arXiv:2408.00118.

Gemma Team. 2024c. Gemma: Open models based on gemini research and technology. *Preprint*, arXiv:2403.08295.

Qwen Team. 2024d. Qwen2.5: A party of foundation models!

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Philip Wadler. 2015. Propositions as types. *Commun. ACM*, 58(12):7584.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 6076, New York, NY, USA. Association for Computing Machinery.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *Preprint*, arXiv:2407.10671.

Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2023. Understanding programs by exploiting (fuzzing) test cases. In *Findings of the Association for Computational Linguistics (ACL)*, Toronto, Canada.

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 427440, New York, NY, USA. Association for Computing Machinery.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

A Appendix

A.1 Motivating Example

Predicting a function's type signature is an efficient and objective method for demonstrating the understanding of the function's logic. As shown in Figure 4, correctly inferring the types of `xs` and `ys` requires deductive reasoning through the logical flow of all functions in the expression. For both humans and rule-based type checkers, the first step in solving Figure 4 involves analyzing the higher-order function `map`. Since the first input to `map` is the function `ord :: Char -> Int`, we can deduce that the parametric type variable `a` is instantiated to `Char` and `b` is instantiated to `Int`. This process

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
"Hello world" :: [Char]

xs = map ord "Hello world"
- complete the following type signature for `xs`
xs :: [Int]

map :: (a -> b) -> [a] -> [b]
chr :: Int -> Char
[1 .. 10] :: [Int]

ys = map chr [1 .. 10]
- complete the following type signature for `ys`
ys :: [Char]
```

Figure 4: Examples of type inference task similar to Benchmark-*F* tasks.

is often referred to as *type instantiation* or *specialization*. The next step is to verify whether the third input to `map` matches these deductions. As expected, this parameter is a string literal, which has the type `[Char]`, leading us to conclude that `xs` should have the type `[Int]`. The type inference for `ys` follows a similar reasoning process.

Both examples demonstrate the close relationship between type inference and understanding a function's logic. Testing a language model's ability to infer types not only reflects its understanding of the program but also its proficiency in logical reasoning. As the Curry-Howard Isomorphism suggests, the expression bound to `xs` serves as proof that `xs` has the type `[Int]`. This connection between function implementation and logical reasoning highlights the importance and effectiveness of Benchmark-*F* in providing a sound and fine-grained evaluation of language models.

A.2 Additional Figures

```
words :: String -> [String]
words s =  case dropWhile isSpace s of
        "" -> []
        s' -> w : words s''
          where (w, s'') = break isSpace s'
```

Listing (3) Monomorphism.

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

Listing (4) Parametric polymorphism.

Figure 5: Example function definitions taken from Haskell Standard Prelude (Jones, 2003).

```
Act as a static analysis tool for type inference.
NOTE: all tasks are monomorphic or parametric
     polymorphic, Do not output any type class.
```

System prompt.

```
1. Use the lowercase alphabet [a..z] for type
     variables instead of numbers.
2. ONLY output the type signature. Do Not Provide
     any additional commentaries or explanations.
```

Instruction prompt.

Figure 6: Prompt used in Benchmark-F.

A.3 Type Rewrite Algorithm

A.4 Additional Evaluation Results

Algorithm 1 Removing natural language identifiers

function TYPEREWRITE($task$)
  $tokens \leftarrow$ TOKENIZE($task$)
  $AST \leftarrow$ PARSE($tokens$)
  $poly_types, mono_types, funcs, vars \leftarrow$ GET_IDENTIFIERS(AST)

  $poly_map \leftarrow$ ZIP($poly_types$, [a to z])
  $mono_map \leftarrow$ ZIP($mono_types$, [A to Z])
  $func_map \leftarrow$ ZIP($funcs$, [f_1, f_2, …, f_n])
  $var_map \leftarrow$ ZIP($vars$, [v_1, v_2, …, v_n])
  $map \leftarrow poly_map + mono_map + func_map + var_map$

  $tokens' \leftarrow [\,]$
  for all $t \in tokens$ **do**
    $t' \leftarrow$ ""
    if $t \in map$ **then**
      $t' \leftarrow map[t]$
    else
      $t' \leftarrow t$
    $tokens' \leftarrow$ APPEND($tokens'$, t')
  $task' \leftarrow$ ASSEMBLE($tokens'$)
  return $task'$

LLM Type	Model	Version/Size	#Correct	Accuracy (%)
API	gpt-3.5-turbo	0125	42	50.00
	gpt-4o-mini	2024-07-18	43	51.19
	gpt-4-turbo	2024-04-09	48	57.14
	gpt-4o	2024-05-13	50	59.52
	claude-3-haiku	2024-03-07	29	34.52
	claude-3-sonnet	2024-02-29	30	35.71
	claude-3-opus	2024-02-29	40	47.62
	claude-3.5-sonnet	2024-06-20	**58**	**69.05**
OSS	Phi-3 (Abdin et al., 2024)	3.8B	1	1.19
		14B	20	26.19
	Mistral (Jiang et al., 2023)	7B	15	17.86
	Mixtral (Jiang et al., 2024)	8×7B	17	20.24
		8×22B	42	50.00
	Gemma (Team, 2024c)	2B	2	2.38
		7B	22	26.19
	Gemma2 (Team, 2024b)	9B	29	34.52
		27B	38	45.24
	Qwen2 (Yang et al., 2024)	1.5B	5	5.95
		7B	21	25.00
		72B	34	40.48
	Qwen2.5 (Yang et al., 2024)	1.5B	18	21.43
		7B	35	41.67
		72B	43	51.19
	DeepSeek-V2 (DeepSeek-AI, 2024)	16B	16	19.05
		236B	38	45.24
	DeepSeek-V2.5 (DeepSeek-AI, 2024)	236B	46	54.76
	Llama3 (AI@Meta, 2024a)	8B	22	26.19
		70B	31	36.90
	Llama3.1 (AI@Meta, 2024b)	8B	28	33.33
		70B	40	47.62
		405B	**51**	**60.71**
Code	CodeGemma (Team, 2024a)	2B	4	4.76
		7B	19	22.61
	Qwen2.5-Coder (Team, 2024d)	1.5B	27	32.14
		7B	39	46.43
	Codestral (Mistral-AI, 2024)	22B	36	42.86
	CodeLlama (Roziere et al., 2023)	7B	26	30.95
		13B	16	19.05
		34B	36	42.86
	Granite-Code (Mishra et al., 2024)	2B	6	7.14
		8B	15	17.86
		20B	23	27.38
		34B	40	47.62
	DeepSeek-Coder-V2 (Zhu et al., 2024)	16B	34	40.48
		236B	**42**	**50.00**

Table 4: Full evaluation results of popular LLMs on Benchmark-F.