



Article ID 1007-1202(2024)04-0349-08 DOI <https://doi.org/10.1051/wujns/2024294349>

Cite this article: CHENG Yu, HUANG Guanming, WU Yishun, *et al.* Ask Me Any Type: Type Inference Plugin for Partial Code on the Web and in the Integrated Development Environment[J]. *Wuhan Univ J of Nat Sci*, 2024, 29(4): 349-356.

Ask Me Any Type: Type Inference Plugin for Partial Code on the Web and in the Integrated Development Environment

□ CHENG Yu^{1*}, HUANG Guanming^{2*}, WU Yishun¹, ZHAO Zijie¹, HE Zhenhao¹, LU Jiaying^{1†}

1. College of Computer and Information Engineering, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;

2. The High School Attached to Jiangxi Normal University, Nanchang 330013, Jiangxi, China

Abstract: Inferring the fully qualified names (FQNs) of undeclared receiving objects and non-fully-qualified type names (non-FQNs) in partial code is critical for effectively searching, understanding, and reusing partial code. Existing type inference tools, such as COSTER and SNR, rely on a symbolic knowledge base and adopt a dictionary-lookup strategy to map simple names of undeclared receiving objects and non-FQNs to FQNs. However, building a symbolic knowledge base requires parsing compilable code files, which limits the collection of APIs and code contexts, resulting in out-of-vocabulary (OOV) failures. To overcome the limitations of a symbolic knowledge base for FQN inference, we implemented Ask Me Any Type (AMAT), a type of inference plugin embedded in web browsers and integrated development environment (IDE). Unlike the dictionary-lookup strategy, AMAT uses a cloze-style fill-in-the-blank strategy for type inference. By treating code as text, AMAT leverages a fine-tuned large language model (LLM) as a neural knowledge base, thereby preventing the need for code compilation. Experimental results show that AMAT outperforms state-of-the-art tools such as COSTER and SNR. In practice, developers can directly reuse partial code by inferring the FQNs of unresolved type names in real time.

Key words: type inference; large language model; prompt learning; web and integrated development environment (IDE) plugin

CLC number: TP311

0 Introduction

Partial code is prevalent in application programming interface (API) documentation and online forums (e.g., Stack Overflow) and is frequently reused in programming tasks^[1,2]. However, partial code often contains non-fully qualified names (non-FQNs) and unde-

clared receiving objects, the compilation error "symbol cannot be resolved" usually occurs when developers copy such partial code into an integrated development environment (IDE). Furthermore, partial code can only be used as text if undeclared receiving objects and non-FQN types are not resolved, severely limiting the effectiveness of program analysis tools in utilizing partial-

Received date: 2023-03-20 © Wuhan University 2024

Foundation item: Supported by the Key Scientific and Technological Research Projects of the Jiangxi Provincial Department of Education (GJJ2200303) and the National Social Science Foundation Major Bidding Project (20&ZD068)

Biography: CHENG Yu, male, undergraduate, research direction: intelligent software engineering. E-mail: yc@jxnu.edu.cn; 2905926811@qq.com

† Corresponding author. E-mail: 003484@jxnu.edu.cn

* These authors contributed equally to this work

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (<https://creativecommons.org/licenses/by/4.0>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

code. For example, due to the lack of accurate type information, a code search tool may recommend irrelevant code^[2], or a code vulnerability analysis tool may fail to detect malicious API behaviors^[3].

To infer the fully qualified names (FQNs) of the undeclared receiving objects and non-FQNs in partial code, existing tools (e. g., SMT^[4], COSTER^[5], SNR^[6]) adopt a dictionary-lookup strategy, which maps simple names and code context to FQNs based on a symbolic knowledge base. Creating a symbolic knowledge base requires a large number of compilable code files. This compilation overhead limits the amount of code context and APIs collected. There will always be code contexts and APIs that the symbolic knowledge base has not previously stored, leading to an out-of-vocabulary (OOV) failure^[7].

Aiming to address the above problems, we propose Ask Me Any Type (AMAT), an inference tool based on our recently proposed neural type inference technique^[7]. Unlike existing tools that parse compiled code files to build the symbolic knowledge base, AMAT treats code as text. It utilizes an FQN fine-tuned code masked language model (MLM) (i. e., CodeBERT^[7] in the current implementation) as a neural knowledge base, eliminating the compilation overhead. Based on MLM, AMAT employs a closed fill-in-the-blank strategy that uses the surrounding code context to fill in missing type names rather than the dictionary lookup strategy used by existing tools^[4-6].

To enhance the usability of AMAT, we integrated it as a plugin into web browsers and integrated development environments (IDEs). This integration directly offers type inference services to users, enhancing code analysis and enriching the coding experience. Using a web plugin, developers can benefit from real-time type inference assistance when reading partial code in API documentation or online forums. The tool can infer the FQNs of undeclared receiving objects and non-FQNs, making it easier for developers to understand the APIs in code snippets. Similarly, the IDE plugin helps developers complete missing import statements when they copy-paste code snippets from online resources into their IDE projects. This feature streamlines the reuse of code snippets by automating import statement completion, making the tool more accessible in web browsers and IDEs.

We evaluated the accuracy and operational efficiency of AMAT's recognition of FQNs. Fine-tuning our tools can significantly improve the accuracy of MLM. In

addition, AMAT shows higher accuracy when using datasets from Stack Overflow code snippets compared with COSTER and SNR. User studies have shown that coding tasks can be completed more quickly and accurately when utilizing our tools for quick fixes instead of integrated development environments.

The contributions of this paper are as follows:

- AMAT overcomes the limitations of a symbolic knowledge base^[4-6] by using an FQN-prompt-tuned code MLM as a neural knowledge base for type inference.
- AMAT is the first plugin to support type inference in web browsers and IDEs, significantly extending the practicality and usage scenarios of FQN inference.
- AMAT solves a fundamental problem in software engineering: FQN inference in partial code, which would enhance subsequent type-aware partial code analysis.

1 Enhancing Code Type Inference with FQN-Based MLM

The architecture of AMAT, as shown in Fig. 1, consists of five components that serve two functions: inference of types and completion of import statements. The former returns a ranked list of top-*k* FQNs for each FQN inference point in partial code. The latter completes the missing import statements. For full technical details of the neural type inference method, readers are referred to Ref.[7].

The following describes the five components of AMAT and their roles in achieving these functions.

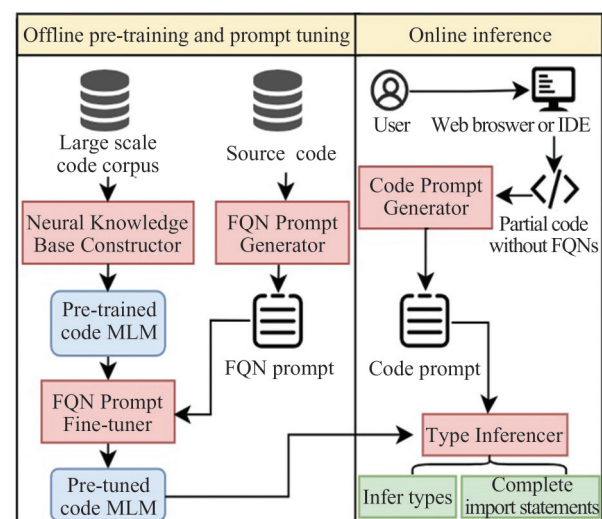


Fig. 1 Architecture of AMAT

1) Neural Knowledge Base Constructor: Based on the concept of code naturalness^[8], code can be treated as text to train the code-based masked language model (MLM). The Neural Knowledge Base Constructor aims to pre-train a code MLM on a large code corpus. We use the pre-trained CodeBERT model^[9], trained on a large code corpus, to predict masked words in the input text. We set type inference as a fill-in-the-blank language task, which aligns well with the training objectives of the masked language model. To address the problem of rare words in program element names, we use the sub-word tokenization technique WordPiece to overcome vocabulary limitations and ensure better representation of rare symbols. In this paper, we propose a new method for type inference using the source code and the pre-trained CodeBERT model. We experimentally show that it is a more effective and efficient method than existing methods.

2) FQN Prompt Generator: Driven by the needs of the type of inference task, we designed an FQN Prompt Generator to automatically generate contextual FQN prompts for tuning the code MLM. FQN Prompt Generation involves FQN annotation, context collection, and FQN masking. FQN annotation is the process of annotating code with FQNs for all types and APIs used in the code, which is crucial for avoiding naming conflicts and ensuring that the code can be compiled and executed correctly. Context collection involves using a small local context, which is key for the model's ability to make accurate type inferences, especially in short and partial code, where limited context can make accurate inferences more challenging. FQN masking involves masking FQN tokens within a small local context of code while leaving other tokens unchanged to create unique prompts for the task. To match the FQN characteristics, the FQN Prompt Generator uses a full-span mask strategy for FQNs during prompt generation rather than a random mask strategy^[9]. The FQN prompts and pre-trained code MLM are then trained in the FQN Prompt Fine-tuner to become a prompt-tuned code MLM capable of understanding code syntax and semantics. More details about the FQN prompt can be found in our work^[7].

3) FQN Prompt Fine-tuner: The FQN Prompt Fine-tuner is designed to integrate code-based MLM into type inference tasks. This method employs FQN prompts to refine the performance of the code-based MLM, facilitating an autonomous enhancement of the model's capabilities.

The homogeneous nature of the pre-training and prompt-tuning processes ensures that the resulting language model can accurately infer FQNs of undeclared receiving objects and non-FQNs in partial code. By leveraging these FQN prompts, the FQN Prompt Fine-tuner trains a pre-existing code-based MLM to generate a prompt-tuned model adept at understanding code syntax and semantics.

4) Code Prompt Generator: The Code Prompt Generator is designed to identify type inference points, which include all undeclared receiving objects and non-FQNs within partial code. For each identified type of inference point, the generator extracts a code block comprising the line containing the type of inference point and the two lines immediately preceding and following it, resulting in up to five lines of context. Subsequently, the Code Prompt Generator processes each code block by tokenizing it using the WordPiece tokenizer^[10]. Mask tokens with a fixed-length mask span are added before the non-FQN type names in the tokenization process to ensure consistent processing across different instances.

5) Type Inferencer: A Type Inferencer based on a prompt-tuned code language model has been developed to enable precise type inference. This component uses a cloze-style fill-in-the-blank strategy to infer the FQNs of type inference points in partial code. Our Code Prompt Generator creates a code prompt based on the provided partial code. Then, our Type Inferencer fills in the masked tokens within the code prompt to obtain the corresponding FQNs. Notably, our Type Inferencer utilizes a variable-length mask prediction method^[11] to predict FQNs of various lengths, accommodating the variability in FQN length. Using this approach, we can achieve accurate and flexible type inference capabilities, enhancing user coding experience.

2 Practical Applications of AMAT in Code Reuse and Understanding

AMAT is a supplementary tool designed for program analysis systems, which is precious when precise information on the FQN type is needed. Integrated as a plugin in web browsers and IDEs, it enables developers to directly retrieve reusable code snippets from popular programming resources like Stack Overflow. AMAT analyzes the FQNs in these snippets, providing essential information on APIs and other programming elements to help developers understand their functions and applica-

tions.

Additionally, AMAT can automatically fill in missing import statements, reducing the risk of programming errors and the workload of manual coding. These features make the code reuse process more intuitive and efficient, enhancing the programming experience for developers, and allowing them to focus more on creative coding tasks. For example, suppose a developer needs to work with JSON Object data in their project and requires the use of a library to parse the data. While browsing online sources, the developer may find a code snippet that demonstrates how to parse JSON Object data us-

ing the library, as shown in Fig.2(a). However, they may not know the correct FQN of the library's JSON Object parsing method. By integrating AMAT as a plugin within their web browser or IDE, as depicted in Fig.2(b), the developer can easily obtain the correct FQN and correctly utilize the library's JSON Object parsing method. Therefore, this tool streamlines the process of comprehending and reusing code snippets, saving developers' time and effort and ultimately enhancing their productivity and efficiency. AMAT is an indispensable tool because it supports developers in their programming tasks and improves their overall experience.



Fig. 2 An example of using JSON Object in search on the web and IDE

2.1 Web Plugin

Developers often rely on online resources, such as Stack Overflow, to search for code examples on API usage (<https://stackoverflow.com/questions/16665124/>). For example, Figure 3(a) shows a code example from the Stack Overflow post. However, this code snippet contains the non-FQNs (e.g., "File"). Our AMAT's web plugin can help developers show the FQN of the non-FQN and link it to the corresponding API specification documentation. Precisely, the developer can move the mouse to hover over the "File", and the top-*N* FQNs (default 2) of the "File" will be displayed in a pop-up hover page when given a partial code on Stack Overflow, as shown in Fig.3(a). This feature helps developers quickly identify the correct FQN, even when encountering ambiguous or unfamiliar code snippets. In addition, developers can click on any inferred FQN and be redirected to the API specification documentation for the FQN, as shown in Fig.3(b). This functionality makes it easy for developers to access the relevant API documentation, which can help them better understand the code snippet and its API or library. Overall, our AMAT web plugin is a valuable tool for developers, increasing their productiv-

ity and helping them overcome obstacles when working with code snippets from online resources.

2.2 IDE Plugin

When developers discover a useful code sample on a Stack Overflow post, they copy and paste it into their IDE for reuse. However, the absence of import statements prevents the code snippet from being directly compiled in the IDE. Since the type of information is missing, a compilation error "symbol cannot be resolved" is thrown, and the APIs are highlighted in red, as shown in Fig.3(a). At this point, the developer can select a zone of code lines and launch the IDE plugin of AMAT, then click the "Complete Import Statements" button, as shown in Fig.4(a). The IDE plugin automatically resolves all the non-FQNs into the corresponding FQNs (default top-*N*) and automatically populates all missing imports. However, the developer can update the generated imports by manually selecting from the ranked list, similar to the web browser plugin. Additionally, developers can choose a specific API and click the "Infer Type" button to obtain the corresponding FQN. Finally, a compiled code file is immediately generated, as shown in Fig. 4(b).



Fig. 3 An example of using the web plugin to make type inference

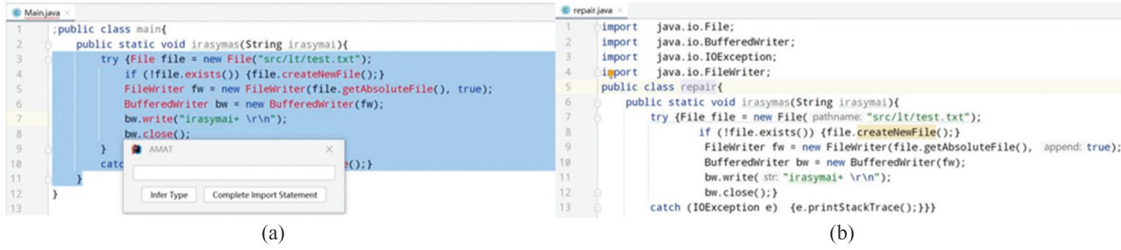


Fig. 4 An example of using the IDE plugin to generate missing import statement

3 Evaluation

AMAT comprises five components, with the FQN Prompt Fine-tuner and Type Inferencer being pivotal. The FQN Prompt Fine-tuner enhances AMAT’s effectiveness by building a neural knowledge base, and the Type Inferencer crucially affects performance by inferring FQNs. To directly assess these capabilities, we propose the following research questions:

RQ1: Is the prompt learning method effectively enabling AMAT to recognize FQNs?

RQ2: How does AMAT compare to state-of-the-art type inference tools such as COSTER and SNR?

We aim to evaluate AMAT’s effectiveness and utility in improving developers’ programming experience by addressing these questions.

3.1 Accuracy of FQNs Recognition

The prompt learning method is used to fine-tune a code-based MLM, establishing a neural knowledge base for type inference. This evaluation assesses its effectiveness in enhancing the model’s ability to accurately recognize FQNs. We seek to verify the model’s accuracy and precision in identifying FQNs, which is crucial for correctly integrating code snippets.

We downloaded the source code for the dataset and metrics for six libraries from their GitHub repositories, including Android, Google Web Toolkit (GWT), Hiber-

nate, Joda Time, and the Java Development Kit (JDK). These source code files are divided into 40% for prompt tuning and 60% for type inference. In this experiment, we randomly select only 10% of the data from the prompt-tuning dataset to fine-tune the MLM because we want to see if the prompt-learning method can achieve a higher training effect with a small amount of data. We use accuracy and BLEU score^[12] to measure the performance of our approach. The accuracy represents the percentage of correctly inferred FQNs, and the BLEU score is used for text generation evaluation based on the fact that the n -gram matches with reference text. We calculated the BLEU score to compare the predicted FQN with the ground-truth FQN, where higher scores indicate better similarity, ranging from 0.00 to 1.00. Additionally, since the shortest FQNs to be inferred contain only three tokens, we computed BLEU-2 (i.e., 2-gram).

When fine-tuning the pre-trained CodeBERT model, we used the following parameter settings: a learning rate of $5E-5$, a batch size of 16, 15 epochs, 100 warmup steps, a weight decay of 0.01, and a maximum sequence length of 512 tokens. These parameters were chosen based on best practices from Ref.[7].

Table 1 shows that MLMs without fine-tuning (zero-shot setting) have an average accuracy of 30.7% and a BLEU score of 47.8%. This indicates that the vanilla code MLM has a particular ability to capture FQN information. In contrast, AMAT (with prompt learning)

achieves an average score of 84.6% and a BLEU-2 score of 93.4% while simulating pre-trained code MLM as neural knowledge bases with only 10% fine-tuned datasets. This indicates that the prompt learning method is effective to stimulate the vanilla code MLM to learn FQN syntactic and usage patterns, even with a small portion of library code. In addition, the average BLEU-2 is also much higher (i.e., 93.4% vs. 47.8%). This shows that the predicted FQN is similar to the ground-truth FQN, even though they are not precisely the same. With additional tool support (e.g., search engine), one may find the correct FQNs from the nearly accurate predictions.

Table 1 Accuracy of FQNs recognition %

Model	Accuracy	BLEU-2
Zero-shot	30.7	47.8
10% prompt	84.6	93.4

3.2 Performance Comparison of Type Inference Tools

Incomplete data types are prevalent in online code snippets, making it essential for tools like AMAT to perform efficiently. This section compares AMAT's operational capabilities and practical value against leading type inference tools such as COSTER^[5] and SNR^[6]. We evaluate AMAT's speed, scalability, and accuracy in real-world programming scenarios to determine its impact on developer productivity and usability.

We use StatType-SO^[4] for comparison, which comes from Stack Overflow and has been used in previous work^[7]. This dataset initially contains 268 partial code snippets. To provide a more comprehensive evaluation, we expand the dataset by collecting additional data from Stack Overflow, resulting in a total of 978 partial code snippets. Each code snippet primarily focuses on API usage and is sourced from one of six libraries: Android, GWT, Hibernate, Joda Time, Xstream, or JDK. To evaluate the performance of AMAT, we utilize accuracy as the primary metric.

A comparison of the performance results is shown in Table 2. It is shown that the average accuracy of COSTER^[5] and SNR^[6] is 83.5% and 68.9%, respectively, while the average accuracy of AMAT is 90.3%, surpassing state-of-the-art tools. This indicates that AMAT can still strongly infer from real code snippets.

Table 2 Comparative accuracy of type inference tools %

Tool	Average accuracy
COSTER	68.9
SNR	83.5
AMAT	90.3

4 Comparative User Study: AMAT vs. IDE Quick Fixes for Code Import Tasks

This section is dedicated to a user study comparing the effectiveness of the AMAT IDE plugin with standard IDE quick fixes in managing code import tasks. The study evaluates the usability and efficiency of AMAT in a real-world programming environment.

We chose 30 code snippets from StatType-SO and pasted the code snippets into a Java IDE, causing the "symbol cannot be resolved" compilation error due to undeclared receiving objects or non-FQNs. However, IDE quick fix can only handle compilation errors caused by non-FQNs. In this study, we focused on completing import statements to resolve non-FQN types, allowing AMAT and IDE quick fixes to address the same compilation error. We recruit 10 master students with three years of Java programming experience to complete the missing import statements. We randomly assign these students into two groups (i.e., GAMAT and GIDE) (5 students in each group). The GAMAT participants use our AMAT tool to fix the missing import statements, who are not permitted to use the IDE quick fix, and GIDE participants use the IDE quick fix for the missing imports. Here, GIDE participants use the IDE to fix problems quickly instead of using COSTER or SNR tools because the tools cannot be successfully deployed. Participants in either GAMAT or GIDE are given 10 minutes of training on how to fix missing imports using the corresponding tools. The time to fix the import for each code snippet is two minutes. We statistic the completion time and the correctness of the fixed import per participant. Furthermore, we interviewed each participant to get their thoughts on how to fix the import statement.

Based on statistical results, it has been found that the average completion time for participants using GAMAT is faster than that of those using GIDE, with times of 1 662 s and 2 262 s, respectively. Additionally, GAMAT participants were found to be more accurate, achieving a success rate of 89.63% compared with the

GIDE success rate of 84.68%.

Through interviews conducted with the participants, it was discovered that five GAMAT users reported a seamless experience when fixing missing imports. They only needed to select the code snippet and then click the "Import Statement" option. In contrast, three GIDE participants reported encountering difficulties when the IDE recommended multiple fully qualified names. Thus, they had to hover and select the appropriate import. This additional step is believed to be the reason behind the slower completion time for GIDE users.

5 Discussion

Our study utilized CodeBERT, a pre-trained code MLM with 1.25 million parameters, to evaluate the prompt learning method for type inference. As discussed in Section 3.1, the results demonstrated that AMAT significantly improves accuracy and BLEU-2 scores compared to zero-shot settings, underscoring the effectiveness of prompt learning even with a small fine-tuning dataset. Considering the potential of larger models like GPT-3, which has 175 billion parameters, there is a strong indication that performance could be further enhanced. Nonetheless, larger context windows might introduce noise, necessitating a balance in context window sizes to effectively capture long-range contextual.

Type inference was performed independently at each point to avoid reliance on code analysis, as detailed in Section 3.2. Investigating correlations between simultaneous inference points could potentially improve FQN generation but may also increase computational overhead. The observed errors in FQN formats, corrected through post-processing, suggest that strategies like reinforcement learning could effectively reduce syntactic errors. This insight highlights an avenue for improving the robustness of AMAT.

6 Conclusion

Using prompt learning to construct a neural knowledge base represents a novel approach to type reasoning. Additionally, introducing the "full-span mask strategy" in prompt learning design significantly improves upon the previous random mask strategy. Notably, the plugin developed through this neural knowledge base type reasoning method demonstrates greater accuracy than even the latest type of reasoning tools, underscoring the prac-

ticality of the plugin. AMAT is the first plugin to support type inference both on the web and in IDEs. It overcomes the out-of-vocabulary problems of previous work^[4-6] for type inference because it relies on the neural knowledge base (i. e., the prompt-tuned code-based masked language model). The tool's source code and dataset are publicly available on GitHub (<https://github.com/SE-qinghuang/AMAT>), allowing other researchers to replicate and extend our work. To further illustrate, a video demonstrates how to use the tool both on the web and in the IDE (<https://youtube/whVW3yzaDoY>). In conclusion, this novel approach to type reasoning presents promising prospects for developing more accurate and efficient type reasoning tools.

References

- [1] Gupta P, Mehrotra N, Purandare R. JCoffee: Using compiler feedback to make partial code snippets compilable[C]//2020 *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. New York: IEEE, 2020: 810-813.
- [2] Thummalapenta S, Xie T. Parseweb: A programmer assistant for reusing open source code on the web[C]//*Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. New York: ACM, 2007: 204-213 .
- [3] Zhou Y Q, Liu S Q, Siow J K, *et al.* Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[C]//*Proceedings of the 33rd International Conference on Neural Information Processing Systems*. New York: Curran Associates Inc, 2019: 10197-10207.
- [4] Phan H, Nguyen H A, Tran N M, *et al.* Statistical learning of API fully qualified names in code snippets of online forums [C]//*Proceedings of the 40th International Conference on Software Engineering*. New York: ACM, 2018: 632-642 .
- [5] Khaled Saifullah C M, Asaduzzaman M, Roy C K. Learning from examples to find fully qualified names of API elements in code snippets[C]//2019 *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York: IEEE, 2019: 243-254.
- [6] Dong Y W, Gu T X, Tian Y Q, *et al.* SNR: Constraint-based type inference for incomplete Java code snippets[C]//*Proceedings of the 44th International Conference on Software Engineering*. New York: ACM, 2022: 1982-1993.
- [7] Huang Q, Yuan Z Q, Xing Z C, *et al.* Prompt-tuned code language model as a neural knowledge base for type inference

- in statically-typed partial code[C]//*Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. New York: ACM, 2022: 1-13.
- [8] Feng Z Y, Guo D Y, Tang D Y, *et al.* Codebert: A pre-trained model for programming and natural languages. [2020-11-03]. <https://arXiv preprint arXiv:2002.08155>.
- [9] Allamanis M, Barr E T, Devanbu P, *et al.* A survey of machine learning for big code and naturalness[J]. *ACM Computing Surveys*, 2019, **51**(4): 1-37.
- [10] Guo D, Ren S, Lu S, *et al.* Analyzing CodeBERT's performance on natural language code search[EB/OL]. [2022-11-03]. <https://api.semanticscholar.org/CorpusID:252587541>.
- [11] Wu Y H, Schuster M, Chen Z F, *et al.* Google's neural machine translation system: Bridging the gap between human and machine translation[EB/OL]. [2016-12-26]. <http://arxiv.org/abs/1609.08144>.
- [12] Papineni K, Roukos S, Ward T, *et al.* BLEU: A method for automatic evaluation of machine translation[C]//*Proceedings of the 40th Annual Meeting on Association for Computational Linguistics — ACL '02*. Morristown: Association for Computational Linguistics, 2001: 311-318.

问我任何类型:嵌入Web浏览器和集成开发环境(IDE)的代码片段类型推理插件

程煜^{1*}, 黄冠鸣^{2*}, 吴贻顺¹, 赵梓杰¹, 何祯豪¹, 卢家兴^{1†}

1. 江西师范大学 计算机信息工程学院, 江西 南昌 330022

2. 江西师范大学 附属中学, 江西 南昌 330013

摘要: 推理代码片段中未声明的接收对象和非完全限定类型名称(非FQNs)的完全限定名称(FQNs)对于有效搜索、理解和重用代码片段至关重要。现有的类型推断工具,如COSTER和SNR,依赖于符号知识库并采用字典查找策略,将未声明的接收对象和非FQNs的简单名称映射到FQNs。然而,构建符号知识库需要解析可编译的代码文件,它限制了API和代码上下文的收集,导致待搜索的FQN不在符号知识库范围。为克服符号知识库在FQN推理中的局限性,本文实现了一种嵌入Web浏览器和集成开发环境(IDE)的类型推理插件——Ask-Me-Any-Type (AMAT)。AMAT使用填空式策略而不是字典查找策略进行类型推理,通过将代码视为文本,把经过微调的大型语言模型(LLM)作为神经知识库,避免了代码编译的需要。实验结果表明,AMAT的性能优于COSTER和SNR等工具。在实践中,开发人员可以运用AMAT实时推理未解析类型名称的FQNs,直接重用代码片段。

关键词: 类型推理;大型语言模型;提示学习;网页和集成开发环境插件

□