

TypePro: Boosting LLM-Based Type Inference via Inter-Procedural Slicing

TEYU LIN, Xiamen University, China
MINGHAO FAN, Xiamen University, China
HUAXUN HUANG, Xiamen University, China
ZHIRONG SHEN, Xiamen University, China
RONGXIN WU, Xiamen University, China

Dynamic languages (such as Python and JavaScript) offer flexibility and simplified type handling for programming, but this can also lead to an increase in type-related errors and additional overhead for compile-time type inference. As a result, type inference for dynamic languages has become a popular research area. Existing approaches typically achieve type inference through static analysis, machine learning, or large language models (LLMs). However, current work only focuses on the direct dependencies of variables related to type inference as the context, resulting in incomplete contextual information and thus affecting the accuracy of type inference. To address this issue, this paper proposes a method called TypePro, which leverages LLMs for type inference in dynamic languages. TypePro supplements contextual information by conducting inter-procedural code slicing. Then, TypePro proposes a set of candidate complex types based on the structural information of data types implied in the slices, thereby addressing the lack of domain knowledge of LLMs. We conducted experiments on the ManyTypes4Py and ManyTypes4TypeScript datasets, achieving Top-1 exact match (EM) rates of 88.9% and 86.6%, respectively. Notably, TypePro improves the Top-1 Exact Match by 7.1 and 10.3 percentage points over the second-best approach, showing the effectiveness and robustness of TypePro.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: Type Inference, Large Language Model, Code Slicing

ACM Reference Format:

Teyu Lin, Minghao Fan, Huaxun Huang, Zhirong Shen, and Rongxin Wu. 2026. TypePro: Boosting LLM-Based Type Inference via Inter-Procedural Slicing. In *Proceedings of The ACM International Conference on the Foundations of Software Engineering (FSE '26)*. ACM, New York, NY, USA, 22 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Dynamic languages like Python and JavaScript do not require explicit type declarations for variables, meaning a variable's type is determined only at runtime. This enables faster development and greater flexibility in how variables are used, but it also introduces issues such as type errors that only surface at runtime and additional runtime overhead for type-related inference [5]. To address issues caused by dynamic typing, Python introduced a standard for type annotations in PEP 484 [30].

Authors' Contact Information: Teyu Lin, Xiamen University, Xiamen, Fujian, China, linteyu@stu.xmu.edu.cn; Minghao Fan, Xiamen University, Xiamen, Fujian, China, fanminghao@stu.xmu.edu.cn; Huaxun Huang, Xiamen University, Xiamen, Fujian, China, huanghuaxun@xmu.edu.cn; Zhirong Shen, Xiamen University, Xiamen, Fujian, China, wurongxin@xmu.edu.cn; Rongxin Wu, Xiamen University, Xiamen, Fujian, China, wurongxin@xmu.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FSE '26, Montreal, Canada

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

```

50 1 def MakeBatchBuilder(dataset_root,log_dir,analysis,epoch_type,model: <mask>, batch_size,use_cdfg, ... ):
51 2     if limit_max_data_flow_steps:
52 3         data_flow_step_max = model.message_passing_step_count
53     ...
54 4     return DataflowGgnnBatchBuilder(..., vocabulary=model.vocabulary, max_node_size=...)
55 5 class Ggnn(classifier_base.ClassifierBase):
56 6     def Makebatch()
57     ...
58 7     message_passing_step_count = ...
59 8     vocabulary = ...
60 9 def TrainDataFlowGGNN(...):
61 10     obj = Ggnn(vocabulary=vocab,test_only=False...)
62     ...
63 11     val_batches = AsyncBatchBuilder( MakeBatchBuilder(..., analysis=analysis, model=obj, ... ) )

```

Fig. 1. The motivating example where model serves as the target variable for data type inference.

The JavaScript superset TypeScript [6] provides static type checking at compile time and is then compiled (transpiled) to plain JavaScript for execution at runtime [14]. These solutions likewise aim to mitigate problems introduced by dynamic typing. However, writing type annotations manually is both time-consuming and error-prone [37], which has motivated research into type inference for dynamic languages.

Existing research has explored dynamic type inference from several aspects. First, industry-used automated type inference tools (such as mypy [3]) mainly rely on pre-defined typing rules. However, such pre-defined rules are often incomplete, which has limited the inference capability of these tools. Second, classification-based methods have cast type inference as a multi-class classification task and have use machine learning models to automatically predict the type of a code fragment—for example, DeepTyper [14] for JavaScript, DLInfer for Python [36], and Type4Py [21]. These methods require substantial amounts of labeled data for training and are heavily dependent on the quality and comprehensiveness of the training set. Consequently, their prediction accuracy often declines when encountering types not included in the training data, such as user-defined types. In recent years, with the advancement of large language models (LLMs), a class of generative LLM-based approaches has emerged—e.g., TypeGen [25] and Tiger [31]. These methods have improved type inference by combining prompt engineering with program-analysis-provided context (such as program slices and suggestions of likely third-party libraries or user-defined types) and other feedback mechanisms.

Although existing approaches have made some progress, a major limitation is that the program context they rely on is often incomplete. Type inference typically requires diverse contextual information: variables may be passed to and modified by different functions, functions may be invoked from other files, and variables may depend on global state or external inputs. Acquiring such context requires inter-procedural data-flow analysis. For example, to infer the type of the parameter model in line 1 of Figure 1, one must perform inter-procedural backward data-flow analysis on the function TrainDataFlowGGNN (lines 9–11). From line 10 it can be determined that the variable is of the user-defined type Ggnn (the class definition of Ggnn appears at line 5). Because they lack inter-procedural data-flow analysis, existing methods (e.g., [21, 25, 31]) cannot accurately infer the type of model.

Based on the above observation, we propose TypePro, an LLM-based type inference method for dynamic languages. Specifically, with a knowledge base of third-party libraries and the project's own data types, TypePro performs inter-procedural code slicing to extract contextual information of the target variable. To address the issue of LLMs lacking knowledge of third-party libraries and custom types, TypePro selects a candidate list of data types from this knowledge base whose

99 structural information is similar to that implied in the slice. These two pieces of information are
100 then combined into a prompt, guiding the LLM to generate data types.

101 We evaluated TypePro in the widely used ManyTypes4Py [20] and ManyTypes4TypeScript [17]
102 datasets. The evaluation results show that TypePro achieved Top-1 accuracies of 87.8% (Many-
103 Types4Py) and 86.6% (ManyTypes4TypeScript). Notably, TypePro improves the Top-1 Exact Match
104 by 7.1 and 10.3 percentage points over the second-best approaches in ManyTypes4Py and Many-
105 Types4TypeScript, showing the effectiveness and robustness of TypePro.

106 In summary, this paper makes the following contributions:

- 107 • To the best of our knowledge, we propose the first inter-procedural static code-slicing method
108 for type inference of dynamic programming languages.
- 109 • We design a new candidate data type selection strategy, which uses the structural information of
110 the data type inferred from the slice to perform type inference.
- 111 • We implemented an automated tool called TypePro and performed extensive experiments against
112 baselines. The results show that TypePro is effective in both Python and TypeScript, and it
113 outperforms existing work in Top-1 accuracy.
114

115 2 Related Work

116 In this section, we categorize existing type inference approaches into three main types: rule-based
117 methods, classification-based methods, and generation-based methods. Each type is discussed in
118 detail below.
119

120 2.1 Rule-Based Methods

121 These methods relied on a predefined set of type rules and employed static analysis to explore
122 variables in the code. When a target variable met a specific pre-defined rule, its type was inferred
123 accordingly [13]. Examples of such methods included Pyright and Pylance [19] released by Mi-
124 crosoft, pyType released by Google [10], and the official Python type checker mypy [3]. In the
125 JavaScript/TypeScript ecosystem, relevant approaches included the official TypeScript compiler tsc
126 and Babel. Additionally, several cross-language methods existed [8, 23, 35].
127

128 These rule-based approaches were known for their high precision, enabling accurate type
129 inferences for the scenarios they encompassed. Nonetheless, a primary challenge associated with
130 these methods was the considerable initial effort needed to establish a comprehensive set of rules,
131 given the immense workload involved. Moreover, a significant limitation of these methods was
132 their insufficient coverage when addressing external function calls and the dynamic features of
133 programming languages. Without developing new rules tailored specifically to these situations,
134 achieving satisfactory results remained challenging.
135

136 2.2 Classification-Based Methods

137 These methods frame type inference as a machine learning classification task, typically employing
138 supervised learning with features extracted from code and target types represented as vectors. In
139 the TypeScript ecosystem, DeepTyper [14] frames type inference as a machine translation problem
140 using a bidirectional RNN to encode code context. LambdaNet [34] adopts a graph neural network
141 on type dependency graphs to emulate static analysis.

142 In Python, DLInfer [36] and Type4Py [21] combine static code slicing with neural models.
143 Type4Py constructs a large vocabulary and maps it to type clusters in a high-dimensional space via
144 a hierarchical neural network.

145 These machine learning methods benefit from extensive prior research and achieve competitive
146 performance on benchmarks. However, they are constrained by a fixed label vocabulary, which
147

limits generalization to unseen types [25]. Performance also heavily depends on high-quality training data, and insufficient or noisy data can degrade effectiveness. Additionally, they often lack interpretability, making it difficult to understand the rationale behind predictions.

2.3 Generation-Based Methods

Generative models are designed to generate missing information, making them suitable for tasks like code generation and infilling. Typically, these approaches place a mask at the location requiring a type annotation and task the model with filling in the type. Common models for such tasks include CodeT5 [33], UnixCoder [12], and InCoder [9], which are pre-trained on large-scale code corpora to predict subsequent code sequences given context. During inference, decoding strategies are used to generate candidate code.

Compared to classification-based methods, generative models often achieve better performance across a wider range of scenarios. However, their effectiveness remains limited. Some works, such as TIGER [31], enhance generative models by incorporating similarity computation for type recommendation, improving their ability to handle user-defined types. Nevertheless, these methods—whether directly using base models like CodeT5 or InCoder, or enhanced frameworks like TIGER—rely heavily on incomplete information, typically restricted to the immediate local context. This often omits critical information and limits the ability to infer user-defined types. Even TIGER, despite its similarity-based recommendation, lacks essential structural information about related types and inter-procedural context.

Large language models (LLMs) have emerged as a prominent research topic in recent years. Trained on large corpora of text and code and comprising billions of parameters, these models demonstrate human-like capabilities in processing and generating text and can be viewed as a class of generative models. Notably, LLMs have been applied to type-inference tasks. Similar to the direct use of generative models, methods based on LLMs may employ a direct question-answering approach, such as TypeGen [25]. Unlike approaches that require training a dedicated generative model from scratch, LLM-based methods leverage general-purpose pre-trained models, significantly reducing development time. Additionally, these methods often incorporate techniques like chain-of-thought prompting to enhance reasoning and improve performance. For example, TypeGen [25] combines LLMs with static analysis by providing code slices as input to the LLMs and translating static analysis results into a chain-of-thought format, i.e., expressing static analysis outcomes in natural language. This approach has demonstrated strong performance on type inference benchmarks. Nevertheless, as previously mentioned, it still faces the same limitations in code slicing and handling user-defined types.

3 Motivation

In this section, we evaluate state-of-the-art approaches to identify their limitations in type inference. We apply HiTyper [24], TypeGen [25], and TIGER [31] to the example shown in Figure 1. Specifically, HiTyper is a classification-based (CLS) method that constructs a Type Dependency Graph (TDG) to represent the composition and dependencies of variables, and then performs type inference using a similarity-based model. TypeGen is a generation-based (GEN) approach that slices code using TDG and integrates the natural language summary of static analysis with type inference examples to construct a chain of thought prompt and generate types. TIGER is considered as a generation-and-similarity-based (GEN+SIM) approach, employing a pre-trained generative model along with a similarity computation model to rank possible data types. It is worth noting that there are other CLS and GEN approaches available, while we selected one representative state-of-the-art method from each category for illustration.

Table 1. The results of TypePro and baselines for the example in Figure 1

Approach	Baseline	Inferred Data Type
GEN	TypeGen	1. typing.Type 2. model.vocabulary 3. model
CLS	Hityper	1. int 2. str
GEN+SIM	Tiger	1. M 2. DataflowGgnnBatchBuilder 3. Ggnn
	TypePro	Ggnn

Slicing Result	Approach
<pre> 1 def MakeBatchBuilder(dataset_root,log_dir,analysis,epoch_type,model: <mask>, batch_size,use_cdfg, ...): 2 data_flow_step_max = model.message_passing_step_count 3 return DataflowGgnnBatchBuilder(... vocabulary=model.vocabulary, max_node_size=batch_size..) 4 class Ggnn(classifier_base.ClassifierBase): 5 def Makebatch() 6 message_passing_step_count = ... 7 vocabulary = ... 8 val_batches = AsyncBatchBuilder(MakeBatchBuilder(..., epoch_type=epoch_pb2.VAL, analysis=analysis,model=obj, ...)) 9 ... </pre>	<u>TypePro</u>
<pre> 1 def MakeBatchBuilder(dataset_root,log_dir,analysis,epoch_type, model, batch_size,use_cdfg, ...): 2 if limit_max_data_flow_steps: 3 data_flow_step_max = model.message_passing_step_count 4 return DataflowGgnnBatchBuilder(... vocabulary=model.vocabulary, max_node_size=batch_size..) </pre>	<u>TypeGen</u>
<pre> 1 def MakeBatchBuilder(dataset_root,log_dir,analysis,epoch_type,model: <extra_id_0>, batch_size,use_cdfg, ...): 2 if limit_max_data_flow_steps: 3 data_flow_step_max = model.message_passing_step_count 4 else: 5 data_flow_step_max = None 6 logfile = (log_dir / 7 "graph_loader" / 8 f"{epoch_pb2.EpochType.Name(epoch_type).lower()}.txt") 9 ... #Complete function content. </pre>	<u>CodeT5&Tiger</u>

Fig. 2. The slices of Figure 1 for TypePro, TypeGen and Tiger.

Table 1 presents the output of TypePro and baseline methods in the example illustrated in Figure 1. We now discuss their results according to the categories of different methods as follows.

- CLS Approach.** It can be observed that the current state-of-the-art classification-based type inference technique, HiTyper, fails to accurately infer the specific type of the parameter `model`. The root cause lies in the fact that the deep learning model of HiTyper relies heavily on large amounts of annotated data for training, making its performance highly dependent on the quality and coverage of the training set. Therefore, the success rate of HiTyper can drop when HiTyper encounters types not present in the training data (e.g., user-defined types). Specifically, for the example in Figure 1, HiTyper incorrectly output primitive types such as `int` and `str`, therefore failing to recognize that the variable should be in the user-defined type called `Ggnn`.
- GEN Approach.** It can be observed that the current state-of-the-art generation-based approach, TypeGen, also fails to infer the type of the parameter `model`. The main reason is that the slices generated by TypeGen only include the data flow of parameters within the function, lacking

any data flow information on the actual arguments of the parameters. For instance, when the TypeGen method performs code slicing on the code shown in Figure 1, it only considers the code related to the parameter `model` inside the function `MakeBatchBuilder`, as well as the statements calling the `MakeBatchBuilder` function (as shown in Figure 2). These sliced segments can only suggest that the parameter `model` might be of the same type as the actual argument `obj`. However, since the slice does not include the type information of `obj`, it is difficult to further infer the concrete type of the parameter `model`.

- **GEN+SIM Approach.** Although TIGER covers user-defined types and third-party library types and is capable of inferring the correct data type for `model`, it only ranks the correct data type in the third position. This is primarily because the type inference process of TIGER relies solely on the local context surrounding the target location (as shown in Figure 2) and ranks candidate data types through generative likelihood computation and the extracted contextual information. However, depending exclusively on the immediate context before and after the target variable makes it difficult to fully reconstruct the structure of non-primitive types, especially when the definitions of these types might be distributed across different locations within the project.

In the scenario depicted in Figure 1, to let LLM accurately determine the type of the parameter `model` in the function `MakeBatchBuilder`, an automated tool should: (1) analyze the call statements related to `MakeBatchBuilder` (i.e., lines 9–11), (2) perform code slicing on the actual argument `obj` passed to `MakeBatchBuilder`, and (3) reconstruct the structural features of potential user-defined data types (e.g., `model.message_passing_step_count`, as shown in line 3 of Figure 1). Specifically, the `def` statement of `obj` occurs at line 10, and its usage is located at line 11. Including these statements in the code slice helps infer the type of the actual argument `obj`. It is worth noting that the definition of `obj` is accomplished through a call to `Ggnn`. Therefore, it can be inferred that the type of `obj` is either consistent with the return type of `Ggnn` or corresponds to the `Ggnn` class itself.

To further infer the type of `obj`, one can perform backward slicing starting from `Ggnn`, incorporating its function/class definition (e.g., the signature at line 5) into the code slice. Finally, by analyzing the transitions from the target function parameter `model` to the argument `obj`, and further to the return type of the function `Ggnn`, together with the structural information (e.g., `model.message_passing_step_count`), TypePro can guide LLM to determine that the type of `model` is `Ggnn`.

To summarize, the key contributions of TypePro comparing to existing approaches are as follows:

- TypePro incorporates inter-procedural analysis of function calls for type inference, in contrast to TypeGen’s code slicing that is limited to intra-procedural analysis. For example, the slicing process of TypePro includes `model` → `obj` flow that depicts the call of `MakeBatchBuilder` in line 11 of Figure 1. Compared to Tiger, which extracts code snippets around the target variable based on heuristics, the code slicing process of TypePro can provide context information that is data-flow related to the target variable, and this contextual information has been experimentally shown to improve the type inference precision (see Section 4.3 for details).
- Through this slicing process, TypePro restores the structural information of user-defined data types (e.g., the `Ggnn` type containing `message_passing_step_count` and vocabulary fields in lines 2-3 of Figure 1) and suggests candidate types matching this structure to the LLM, thereby improving effectiveness (see Sections 4.2 and 4.4).

4 Approach

4.1 Overview

Figure 3 illustrates the overall workflow of TypePro. First, the analysis process of TypePro relies on a knowledge base constructed from a large number of project files, incorporating domain knowledge

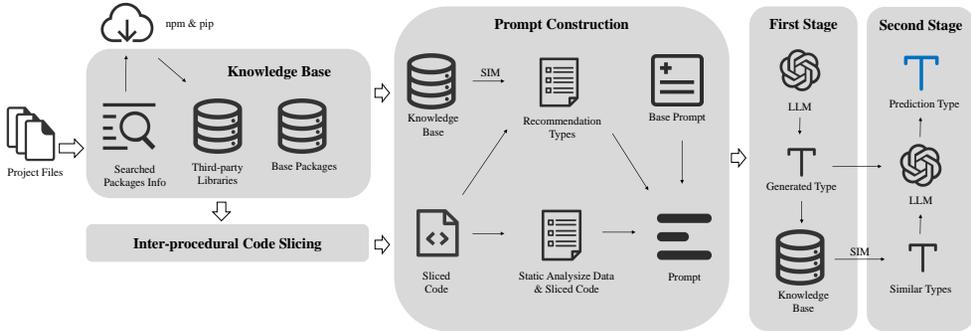


Fig. 3. The Overview of TypePro.

of user-defined data types and third-party library types included in the project under analysis. When processing a target file, for each variable requiring type inference, TypePro performs inter-procedural code slicing to extract code snippets relevant to the variable at runtime as context. Based on the above two inputs, TypePro constructs a prompt containing (1) the code snippets obtained from slicing and (2) a list of candidate data types retrieved from the knowledge base. Finally, it leverages a LLMs to generate the inferred data type.

4.2 Knowledge Base Construction

To enable TypePro to support user-defined types and custom types from third-party libraries, TypePro collects all class definitions in the project files where the current file is located as user-defined types. Then, it examines the import statements to determine which source files are imported in the current file and adds the class definitions of these files to the list of user-defined types. For types from third-party libraries, we followed the existing practices of Peng et al. [25] to download the top 5,000 most popular packages in the Python ecosystem (from `libraries.io`) and the top 5,000 packages in the TypeScript ecosystem (from `npm`). The knowledge base stores the following information closely related to the structure of user-defined data types: (1) class names, (2) the packages to which the classes belong, (3) public API signatures of these classes, and (4) the list of fields of these classes.

4.3 Code Slicing

TypePro performs inter-procedural code slicing to identify and analyze code segments essential to the target (e.g., variables, expressions, functions). Existing related work, such as TypeGen [25] and DLInfer [36], also employs code slicing for type inference. However, as noted in Section 3, these methods exhibit deficiencies in handling inter-procedural dependencies, resulting in limited success rates when using sliced information as context for type inference. In contrast, TypePro's inter-procedural slicing not only supplements context beneficial for type inference—such as signatures of used functions and call information—but also further incorporates other variables that exhibit dependency relationships with the target variable (either being depended on or depending on it). This approach provides richer and more effective contextual information for type inference.

Algorithm 1 illustrates TypePro's slicing process, which is adopted from the process proposed by Horwitz et al. [15]. Given a target program P as input, TypePro first constructs the System

Algorithm 1: Code Slicing Process

```

344 Algorithm 1: Code Slicing Process
345 Input:  $P$ : The target program;
346  $v$ : The variable whose type is to be inferred.
347 Output:  $slices$ : A set of slices containing relevant statements for type inference.
348 // 4.3.1 System Dependency Graph Construction
349 1 Initialize an empty list  $PDG \leftarrow []$ ;
350 2 foreach function  $f$  in  $P$  do
351   3 |  $pdg_f = \text{buildPDG}(f)$ ;
352   4 | Add  $pdg_f$  to  $PDG$ ;
353 5  $G = \text{buildSDG}(PDG)$ ;
354 // 4.3.2 Code Slicing Generation
355 6 Identify the set of target variables  $TV$  based on  $v$ ;
356 7 Initialize an empty list  $slices \leftarrow []$ ;
357 8 foreach  $tv \in TV$  do
358   9 | Initialize the worklist with the pair  $(tgtStmt, \{tgtStmt\})$ ;
359   10 | while worklist is not empty do
360     11 | Remove  $(s', slice)$  from worklist;
361     12 | Mark  $s'$  as visited;
362     13 | if no unvisited statement  $s$  with an edge  $(s, s')$  in  $G$  then
363       14 | | Add  $slice$  to  $slices$ ;
364       15 | | continue;
365     16 | foreach unvisited statement  $s$  with an edge  $(s, s')$  in  $G$  do
366       17 | | if  $s$  is a def or use of  $tv$  then
367         18 | | |  $slice \leftarrow slice \cup \{s\}$ ;
368       19 | | else
369         20 | | | Add the pair  $(s, slice)$  into worklist;
370 21 return  $slices$ ;

```

Dependence Graph (denoted as G) to support inter-procedural control-flow and data-flow analysis. Then, given the target variable v , TypePro identifies all statements in G related to v to provide the context for the LLM to infer the corresponding data type.

4.3.1 System Dependency Graph Construction. TypePro first constructs G for the target program P (lines 1–4). Following the practices of existing slicing approaches [36], for each function defined in P , TypePro scans the code to extract data flow and control flow information to build the program dependency graph (PDG). Then, to construct the inter-procedural information of G , TypePro searches in PDG for call statements in the form of $ret1, ret2, \dots = \text{func}(p1, p2, \dots)$, resolving the function func according to the following three rules:

- **Function Name Matching.** TypePro first tries to match the name of the callee func in the call statement, and process only the functions with the same name further.
- **Parameter Matching.** TypePro further match the actual parameters at the call site with the formal parameters in the function signature. First, priority is given to explicit parameter names (e.g., keyword arguments) for alignment with the formal parameters of the candidate functions. If no explicit parameter names are provided, the original types of the actual parameters ($p1$,

393 $p2, \dots)$ (e.g., number, float, etc.) will be inferred and matched with the constraints of the
 394 candidate function’s formal parameters.

395 • **Return Value Matching.** In Python, a function may return more than one value. Therefore,
 396 TypePro scans the return statements within the body of function `func` to infer the actual number
 397 of return values, and checks whether this number matches the number of receiving variables
 398 (a.k.a, `ret1, ret2, ...`). For example, `return (x, y)` indicates that the function returns two
 399 values. If there are multiple return statements in the function body, TypePro takes the maximum
 400 number of return values among all the return statements as the number of return values for the
 401 function.

402 **4.3.2 Code Slicing Generation.** After constructing G , TypePro takes as input the variable v for the
 403 project. In line 5, TypePro starts by expanding the set of target variables TV associated with v
 404 to enable precise type inference. Such the type of expansion is intended to support inter-procedural
 405 analysis and is governed by the following rules:

- 406 • **Fields:** For variables that are fields of a class, TypePro adds all the use statements of these fields
 407 to the TV .
- 408 • **Function Parameters:** Since function parameters lack explicit definitions, TypePro not only
 409 adds the function signature where the parameter appears to the TV , but also enumerates the
 410 uses of these functions within the project, adding the uses of these functions to *slice* as well.
- 411 • **Return Values:** If the v accepts the return value of a function (e.g., `ret1, ret2,...`), then all return
 412 statements of that function are added to the *slice*.
- 413 • **Other Variables:** For the other variables, TypePro will directly add v to TV .

414 Subsequently, for each identified tv in TV , TypePro proceeds by performing backward and
 415 forward tracing on the identified *slice* to collect all def and use statements (lines 7–19). Specifically,
 416 TypePro locates the statement *tgtStmt* where tv locates in G (line 8), *tgtStmt* also works as the
 417 initial slice for the given tv . Then, TypePro employs a worklist to progressively perform forward
 418 and backward propagation in G to add the statements containing all related defs and uses of tv
 419 into *slice*. This process continues until all def-use relations of the target variable and its related
 420 variables are fully traced, and the relevant statements are incorporated into the slice. The algorithm
 421 finally outputs the slices, which contain all the *slices* for v (line 20).

422 4.4 Prompt Construction

423 Finally, TypePro generates prompts and leverages a LLMs for type inference. Figure 4 demonstrates
 424 the example prompt generated by TypePro to infer the data type of the variable `model` from Figure 1.
 425 Within this prompt, the location of the variable whose type needs to be inferred is marked as
 426 `<mask>`, and the prompt incorporates the following two key pieces of information: (1) The previously
 427 generated inter-procedural slice for the target variable; (2) Candidate data types recommended
 428 from user-defined or third-party libraries.

429 As discussed in Section 3, existing methods [24, 25, 31] struggle to effectively infer the data type
 430 of the variable `model` because the contextual information they rely on is insufficient to reconstruct
 431 the structural features of the data type (such as fields, public functions, etc.). TypePro addresses this
 432 limitation by recommending candidate data types whose structural information highly aligns with
 433 that inferred from the code slice, thereby enhancing the accuracy of type inference. Specifically, for
 434 a given candidate type $type$, TypePro calculates a matching score, which quantifies how closely the
 435 structure of type matches that of *tgtVar*. This score is computed as follows:
 436

$$437 \text{Score}(Var, type) = \frac{|(F_{Var} \cup M_{Var}) \cap (F_{type} \cup M_{type})|}{|F_{Var} \cup M_{Var}|} \quad (1)$$

442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490	<p>Prompt Template for TypePro</p> <p>System: You are an expert in dynamic language type inference. Please perform accurate type inference based on the content I provide.</p> <p>User: Next, you will be provided with a piece of Python code slicing. You will infer the variable type or function return type in Python and fill in the type annotation in <code><mask></code>. example for your output: <code>mask: str</code></p> <p>The candidate types analyzed from the import information are: {Candidate types}</p> <p>The code you need to make a prediction is: {Slicing Code}</p> <hr/> <p>Example</p> <p>System: You are an expert in dynamic language type inference. Please perform accurate type inference based on the content I provide.</p> <p>User: Next, you will be provided with a piece of Python code slicing. You will infer the variable type or function return type in Python and fill in the type annotation in <code><mask></code>. example for your output: <code>mask: str</code></p> <p>The candidate types analyzed from the import information are:</p> <pre>class Ggnn(classifier_base.ClassifierBase): def MakeBatch(self, epoch_type, graphs, ctx) -> batches.Data: def GraphReader(self, epoch_type, graph_db, filters, limit, ctx) -> graph_database_reader.BufferedGraphReader: ...</pre> <p>The code you need to make a prediction is:</p> <pre>import pathlib import time ... def MakeBatchBuilder(dataset_root,log_dir,analysis,epoch_type,model: <mask>, batch_size,use_cdfg, ...): ... data_flow_step_max = model.message_passing_step_count ... return DataflowGgnnBatchBuilder(... vocabulary=model.vocabulary, max_node_size=batch_size..) class Ggnn(classifier_base.ClassifierBase): def Makebatch() ... obj=Ggnn(vocabulary=vocab,test_only=False,node_y_dimensionality=2,graph_y_dimensionality=0,graph_x_dimensionality=0.) ... val_batches = AsyncBatchBuilder(MakeBatchBuilder(..., epoch_type=epoch_pb2.VAL,analysis=analysis, model=obj, ...) ... </pre>
---	--

Fig. 4. The prompt generated for the example in Figure 1

Specifically, F_{Var} denotes the set of fields associated with $tgtVar$ that appear in *Slice*; M_{Var} denotes the set of public functions associated with $tgtVar$ that appear in the *Slice*, F_{type} denotes the set of fields defined in candidate type, and M_{type} denotes the set of public functions defined in candidate type. TypePro selects up to $max_{candidate}$ (set to 5 in the evaluation) candidate data types from both the project itself and the knowledge base and then filter out data types whose similarity scores fall below the set minimum thresholds, we set the minimum threshold $threshold_{Score}$ for the *Score* as 0.5. The detailed structural information of these candidate types is then explicitly listed in the prompt.

The aforementioned process constructs the initial prompt. However, LLMs may generate outputs in different formats. For example, (1) directly outputting the structure information of a data type, such as `{name: string, age: number}`, which is actually a user-defined type (e.g., `class Student`) that has already been included in the knowledge base, and (2) outputting the name of a data type but with typos (e.g., `Neme`, which is a typo for `Name`). To reduce hallucination issues caused by the randomness of LLMs, TypePro executes a two-stage type generation process. In the first stage, TypePro provides the prompt generated through the above process to the LLMs and obtains its initial output. TypePro then repeats the above actions with reference to the output of the initial prompt. Specifically, for the first type of output, TypePro refers to Equation 1 to attempt to map it to a user-defined data type in the knowledge base. For the second type of output, TypePro considers

Table 2. Dataset for Type Inference. Specifically, **Arg** refers to function parameter types, **Ret** refers to function return type, **Var** refers to local and global variables, **Gen** refers to generic types, **Ele** refers to elementary types (e.g., str and int), and **Usr** denotes types defined in the project or third-party libraries.

Language	Dataset	Total	Categories					
			Arg	Ret	Var	Ele	Gen	Usr
Python	Training Set	226,767	44,276	20,395	162,096	119,742	62,268	44,757
		100%	19.52%	8.99%	71.48%	52.80%	27.46%	19.74%
	Test Set	101,392	20,885	9,393	71,114	52,869	28,227	20,296
		100%	20.60%	9.26%	70.14%	52.14%	27.84%	20.02%
	Sampled Test Set	11,029	2,269	958	7,802	5,684	3,312	2,033
		100%	20.57%	8.69%	70.74%	51.54%	30.03%	18.43%
TypeScript	Sampled Training Set	260,767	106,571	42,780	111,416	246,784	-	13,983
		100%	40.87%	16.41%	42.73%	94.64%	-	5.36%
	Sampled Test Set	30,805	13,019	4,055	13,731	29,731	-	1,074
		100%	42.26%	13.16%	44.57%	96.51%	-	3.49%

the similarity of type names by using the BM25 [26] algorithm, and selects the most similar type from the knowledge base as the output.

4.5 Discussion

Note that TypePro’s effectiveness may be affected by limitations in SDG construction. As discussed in Section 4.3.2, inferring inter-procedural information relies on matching function names, parameters, and return values. For certain call statements, TypePro may identify multiple candidate functions and conservatively provide all of them to the LLM. While this ensures completeness, it may introduce imprecision that affects type inference. To evaluate the impact of the aforementioned limitations, we sampled 2,500 variables from the ManyTypes4Py and ManyTypes4TypeScript datasets. In ManyTypes4Py, 2,273 variables involved slicing with a single matching function, achieving 89.93% precision; the remaining 227 with multiple matches achieved 89.42%. In ManyTypes4TypeScript, 2,321 variables had a single match with 85.61% precision, and 179 had multiple matches with 85.47% precision. These results indicate that the above process has limited impact on TypePro’s performance.

Another threat is that the proposed slicing process does not support code generation mechanisms in Python and TypeScript, such as calls to the eval function, which are rarely used. In the ManyTypes4Py dataset, there are 210 eval calls across 122 files, while in the ManyTypes4TypeScript dataset, there are only 5 eval calls in 2 files. As stated in existing work [11, 27, 28], such dynamic features limit the effectiveness of static analysis approaches. We plan to explore incorporating runtime information to better handle these features in future work.

5 Evaluation

We evaluated TypePro by answering the following research questions (RQs):

- **RQ1:** How does TypePro perform on Python datasets compared to other baselines?
- **RQ2:** How robust is TypePro in inferring types for different categories of variables, including elementary types, generic types, and user-defined types?
- **RQ3:** Can the results of TypePro be generalized to different types of LLMs?

- **RQ4:** What is the contribution of each component of TypePro (i.e., **code slicing** and **type recommendation**) to its overall effectiveness?
- **RQ5:** How effective is TypePro on inferring data types in TypeScript programs?

5.1 Evaluation Datasets

We utilized the ManyTypes4Py [20] and ManyTypes4TypeScript [17] datasets for our experiments. Specifically, ManyTypes4Py is a widely adopted dataset containing 869K annotations of 5,382 Python projects for type inference. Following the practices of recent studies [25], we partitioned the dataset into training and testing sets using a 70-30 split and further selected data from 100 projects within the test set for evaluation. On the other hand, the ManyTypes4TypeScript dataset is designed for type inference in TypeScript. ManyTypes4Typescript contains 9M+ data annotations in 13,953 projects, which is ten times the size of ManyTypes4Py. We extracted 10% of the pre-partitioned training set from it as the training data and further selected data from 100 projects within the test set for evaluation. The statistics of selected training and test dataset are shown in Table 2. Note that the ManyTypes4TypeScript dataset does not include data annotations for generic types (e.g., generic types such as `Array<T>` and `Map<K, V>` are annotated simply as `array` or `map`).

5.2 Baselines

We selected the following research methods in type inference as our baselines.

- **Type4Py [21]:** A supervised learning-based classification method that constructs large-scale Python type clusters and categorizes programs into different clusters for type prediction. For Type4Py, we directly used the Docker image released by the authors to perform inference on the test set.
- **HiType [24]:** A method that builds a Type Dependency Graph (TDG) to describe variable composition and dependencies, combined with a similarity-based model for type inference. For HiType, we utilized Type4Py as its inference model.
- **TypeGen [25]:** An approach that performs code slicing via TDG, constructs a chain-of-thought and examples using natural language descriptions of static analysis knowledge, and then leverages a LLMs to generate types. For TypeGen, we followed the default parameters set by the authors, preprocessed the dataset according to their methodology, and then conducted inference on the test set.
- **TIGER [31]:** A two-stage type inference method for Python that employs a trained generative model along with a similarity computation model to rank and recommend plausible types. For TIGER, we adhered to the authors' approach, training the model on our training set and subsequently performing predictions.

Note that we omitted rule-based approaches as they are unlikely to outperform machine learning-based methods in terms of performance and generalization on these datasets. Furthermore, we compared TypePro with three cloze-style type inference models: CodeT5 [33], CodeT5+ [32], and UnixCoder [12]. We did not include DLInfer [36] in our baselines because its type inference process relies on PySlicer, a static slicing tool for Python that has not been open-sourced. For CodeT5 [33], CodeT5+ [32], and UnixCoder [12], we downloaded the initial weights from the Hugging Face Hub [16] and fine-tuned them on our training set.

5.3 Metrics

We employ the following metrics from prior work [21, 24, 25, 31] to evaluate the performance of TypePro and the baseline methods:

- **Exact Match (EM):** This metric calculates the proportion of type predictions made by a method that exactly match the developer-provided type annotations.
- **Base Match (BM):** This metric calculates the proportion of type predictions that partially match the developer-provided type annotations.

Formally, we parse Python or TypeScript tag types T_{label} and the predicted types from a method T_{pre} into type sets Set_{label} and Set_{pre} .

$$\begin{aligned} T_{label} &\rightarrow Set_{label} = Set(T_1, T_2, \dots, T_n) \\ T_{pre} &\rightarrow Set_{pre} = Set(T_1, T_2, \dots, T_n) \end{aligned}$$

For example, the Python type `Union[str, int]` is parsed as $Set(str, int)$, and the TypeScript type `{a?:number, b:string}` is parsed as $Set(\{a:number, b:string\}, \{b:string\})$. When the two type sets are identical, we consider Exact Match to be satisfied. If the two sets have any overlap, we consider Base Match to be satisfied.

$$\begin{aligned} Set_{label} = Set_{pre} &\Rightarrow EM \\ Set_{label} \cap Set_{pre} \neq \phi &\Rightarrow BM \end{aligned}$$

Notably, for generic types in Python and TypeScript, we consider predictions to satisfy BM as long as they belong to the same generic category. For example, in Python, `List[str]` and `List[int]` are considered a base match; similarly, in TypeScript, `Array<string>` and `Array<number>` are considered a base match.

For the any type: If the annotated type in the dataset is any, any predicted type is considered to satisfy both EM and BM. If the predicted type is any, but the annotated type in the dataset is any specific type (other than any), it is considered to satisfy neither EM nor BM.

Furthermore, we utilized MRR (Mean Reciprocal Rank) as an evaluation metric [18, 21]. This is a commonly used indicator to assess the performance of models such as search engines or recommendation systems, particularly suited for query-based problems like information retrieval and recommendation tasks. The score for each data point is calculated by taking the reciprocal of the rank of the first correct answer. The formula is as follows:

$$MRR@n = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{1}{rank_i} (rank_i \in [1, n]) \quad (2)$$

where $|D|$ denotes the total number of data, and $rank_i$ represents the rank position of the first correct answer for the i -th query, $MRR@n$ represents the retrieval of data ranked from 1 to n , and $rank_i$ falls within the range $[1, n]$.

TypePro utilized the API provided by OpenAI and selected GPT-4.1 as the generative model, which is one of the state-of-the-art LLMs. We set the temperature to 0.2 and top_p to 0.3, generating 20 samples for each target variable. The final prediction was determined by ranking the generated results based on their frequency of occurrence, This is the same approach as other studies [25]. All experiments were conducted on a Linux machine (Ubuntu 24.04) equipped with a 128-core Intel Xeon Gold 6338 CPU @ 3.200GHz and two NVIDIA RTX A6000 GPUs.

5.4 Results of RQ1

To answer RQ1, we evaluate the effectiveness of TypePro in type inference of Python by comparing the results with baselines on ManyTypes4Py [20]. We compared two classification-based methods, HiTyper [24] and Type4Py [21], as well as generative models such as CodeT5 [33], CodeT5+ [32] and UnixCoder [12]. TypeGen is categorized as the GEN-based approach, as LLMs are also considered as a form of generative model.

Table 3. The ManyTypes4Py results of TypePro and baselines in terms of **Var**, **Ret**, and **Arg**. The data point with a dark blue background indicates the best result. The data point with a light blue background indicates the second-best result.

Metric	Approach	Catalog	Top-1				Top-3				Top-5				MRR@5				
			Var	Ret	Arg	All													
EM	HiTyper	CLS	82.2	60.7	67.3	76.2	85.5	64.5	77.3	81.4	85.9	65.1	78.3	81.9	87.0	70.4	68.5	81.7	
			62.2	41.8	68.1	61.9	70.8	51.1	81.1	71.5	72.7	53.4	81.7	73.1	66.5	46.7	74.3	66.6	
	TypeGen	GEN	72.6	62.1	67.4	70.7	81.2	71.9	79.0	80.0	82.8	73.9	81.4	81.7	76.9	67.1	73.2	75.3	
			84.7	62.9	64.6	78.7	89.4	69.9	74.2	84.6	91.0	72.7	76.3	86.4	87.2	66.8	69.4	81.6	
			83.2	65.8	60.9	77.1	90.5	75.1	75.9	86.2	92.3	77.0	79.2	88.3	86.9	70.4	68.5	81.7	
			84.6	65.4	63.4	78.6	88.8	71.8	73.3	84.1	90.5	73.9	75.6	86.0	86.9	68.9	68.4	81.5	
	TIGER	GEN+SIM	85.1	70.6	76.0	81.9	90.7	81.2	89.3	89.6	91.5	83.3	91.4	90.8	87.8	75.8	82.7	85.7	
			89.6	88.9	87.2	88.9	93.1	91.6	91.3	92.4	93.2	91.9	91.7	92.7	91.1	90.2	89.1	90.6	
	BM	HiTyper	CLS	90.3	80.6	74.7	85.2	93.8	84.9	84.6	90.5	94.4	85.2	85.8	91.3	91.9	80.6	75.1	87.5
				69.3	45.2	70.0	67.6	79.5	55.8	83.2	78.4	81.4	58.2	84.4	80.2	74.4	50.6	76.4	73.0
TypeGen		GEN	85.5	70.7	75.0	82.1	90.0	80.1	85.1	88.1	90.6	81.7	86.4	88.9	87.7	75.5	79.9	85.1	
			90.9	70.6	72.8	85.4	94.4	78.3	81.8	90.4	95.6	80.6	84.2	91.9	92.7	74.6	77.4	87.9	
			89.1	76.8	68.1	83.7	94.5	84.6	81.8	91.0	96.2	86.5	85.1	93.1	91.9	80.6	75.1	87.5	
			90.9	75.5	70.7	85.4	94.2	82.6	79.5	90.2	95.5	84.9	82.2	91.9	92.7	79.3	75.3	87.9	
TIGER		GEN+SIM	90.8	78.9	82.6	88.1	93.7	88.7	93.0	93.1	94.3	90.0	94.7	94.0	92.3	83.6	87.9	90.6	
			93.5	91.2	88.1	92.1	95.6	94.1	91.7	94.6	95.7	94.2	92.1	94.8	94.5	92.5	89.8	93.3	

Table 3 presents the results. As we can see, TypePro achieved an 88.9%, 92.7%, and 92.7% accuracy in Top-1, Top-3, and Top-5 Exact Match, representing an improvement of 7.0, 2.8, and 1.9 percentage points over the second-best method, TIGER. In detail, TypePro achieved the best results across all aspects of exact match in Var, Ret, and Arg. An interesting observation is that, for Top-1 Exact Match, TypePro showed remarkable improvements in inferring Ret and Arg, outperforming TIGER by 18.3 and 11.2 percentage points, respectively. As stated by existing work[21, 22], developers are more likely to use the first suggestion provided by a tool, which highlight the effectiveness of TypePro in achieving the highest success rate without requiring additional manual effort.

For Base Match, TypePro also achieves a Top-1 accuracy of 92.1%, which is a 4 percentage point improvement over the second-best method. Specifically, in terms of Top-3 and Top-5 results, TypePro ranks either first or second across different groups. This demonstrates the robustness of TypePro regarding effectiveness under different evaluation settings.

5.4.1 Comparison with Classification-Based Approaches. As shown in Table 3, TypePro clearly outperforms CLS approaches, including HiTyper and Type4Py, across all type categories in terms of Exact Match and Base Match metrics. Notably, TypePro achieves improvements of 12.8 and 27.1 percentage points over HiTyper and Type4Py, respectively, in Top-1 Exact Match.

The main reason for these results is that elementary types are typically included in the vocabularies of CLS approaches' pre-trained models and have a large number of training samples. However, such a large amount of training data does not include information generated by inter-procedural analysis. Therefore, when applying HiTyper and Type4Py to Ret and Arg categories, their performance drops significantly. For example, the accuracy of Type4Py on Ret category is only 41.8%. In contrast, TypePro achieves success rates of 88.9% and 87.2% on these categories, respectively, highlighting its advantage over CLS approaches.

5.4.2 Comparison with Generation-Based Approaches. As shown in Table 3, the generation-based (GEN) approaches CodeT5 and CodeT5+ achieved Top-1 accuracies of 78.7% and 78.6%, respectively,

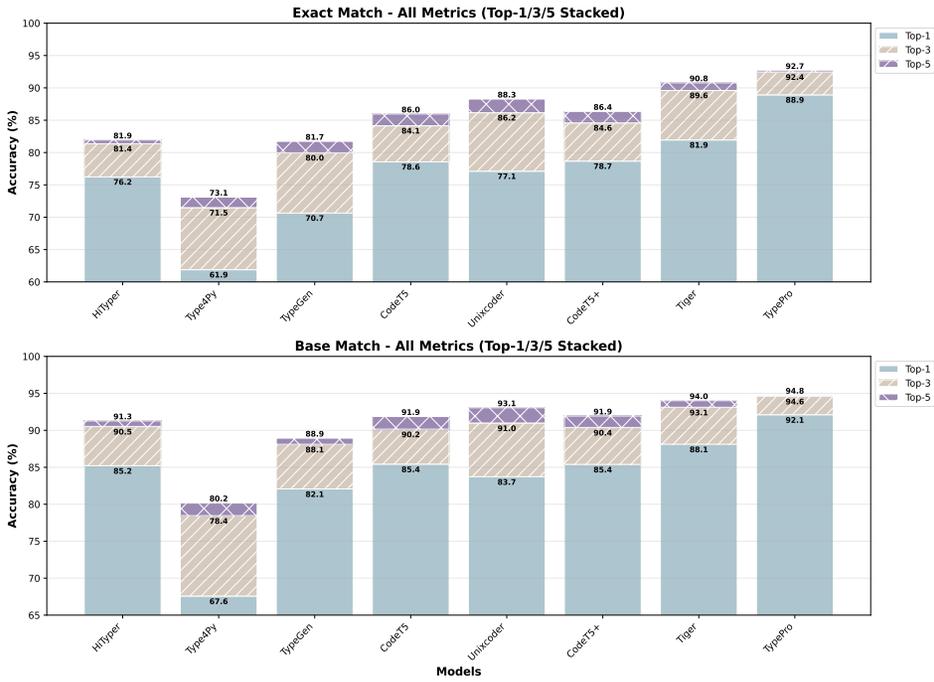


Fig. 5. Correct answer distribution between TypePro and baselines

while UnixCoder and TypeGen reached 77.1% and 70.7%, respectively. For the same reason mentioned above, these models also performed well in the *Var* category; for example, CodeT5 achieved an accuracy rate of 84.7% in this category, largely because variables constitute a significant portion of the training data (71.48%). However, the accuracy of these three methods in other categories drops to below 70%. In contrast, TypePro’s inter-procedural code slicing process incorporates structural information of types, giving it an advantage in inferring the *Ret* and *Arg* categories. It is worth noting that although TypeGen pre-extracts some third-party library types and user-defined types for inclusion in its prompts to the LLMs, it only provides class names without structural information. This lack of contextual detail leads to reduced success rates.

5.4.3 Comparison with TIGER. As shown in Table 3, TIGER achieved a Top-1 exact match rate of 81.9% on our test set, while TypePro outperformed TIGER by 7.9 percentage points in overall accuracy. Specifically, although the performance on *Var* types was similar, TIGER focuses on variable names and contextual textual similarity, whereas TypePro leverages inter-procedural analysis to obtain contextual information of the target variable and provides it to the LLM. This enables TypePro to achieve improvements of 17.3 and 11.2 percentage points over TIGER for *Ret* and *Arg* types, respectively.

For the Base Match task, although we can see that TypePro achieves overall better Top-1, Top-3, and Top-5 performance than TIGER on the test set, its performance in the *Arg* group for infer Top-3 and Top-5 only reaches 91.7% and 92.1%, which is 1.3 and 2.6 percentage points lower than TIGER, respectively. According to our observations, since TypePro is an LLM-based type inference method, when we configure the LLM to output more diverse results, it often produces hallucinations, leading to a decline in performance for Top-3 and Top-5 predictions. Nevertheless, TypePro still demonstrates a significant advantage over TIGER in terms of Top-1 performance.

Table 4. Comparison results with baselines on the three data types **Ele**, **Gen**, and **Usr**.

Approach	Catalog	Top-1			Top-3			Top-5			MRR@5		
		Ele	Gen	Usr									
HiType	CLS	86.6	63.6	66.9	92.0	67.5	73.3	92.7	68.0	73.6	89.8	75.4	69.4
Type4Py		79.6	42.0	45.9	88.1	54.5	53.9	89.3	57.6	54.3	83.8	48.2	49.7
TypeGen	GEN	84.2	44.8	74.8	88.9	62.4	83.4	89.8	66.3	84.4	86.5	53.7	79.0
CodeT5		88.3	68.9	67.6	92.3	76.7	75.8	93.7	79.0	77.7	90.5	72.3	71.8
Unixcoder		86.3	69.8	63.5	93.2	80.8	75.3	94.8	83.5	77.8	89.8	75.4	69.4
CodeT5+		88.9	68.6	65.8	92.9	75.2	74.2	94.1	77.7	76.8	91.0	72.2	70.3
TIGER	GEN+SIM	88.9	71.6	79.3	91.7	84.4	92.3	91.8	86.7	94.7	90.3	77.9	85.7
TypePro		93.2	82.6	87.6	95.0	88.4	92.3	95.0	88.7	92.9	93.9	85.4	89.8

5.4.4 *MRR*. As we can see in Table 3, In terms of MRR@5, TypePro achieves 90.6 and 93.2, outperforming all the baselines we compared. It can be observed that TypePro significantly outperforms baselines in MRR@5 for Ret and Arg category, with 14.4 and 6.4 percentage point of improvement on Ret and Arg, respectively. For Var category, the difference between TypePro and the second-best approach is minimal, as these categories are relatively straightforward and well-represented in the training set. Moreover, Figure 5 shows the distribution of correct answers between TypePro and baselines. Moreover, Figure 5 shows the distribution of correct answers between TypePro and the baselines. As we can see, on the test dataset, TypePro ranks the correct answers in the Top-1, Top-3, and Top-5 for 88.9%, 92.4%, and 92.7% of the data, respectively, all of which surpass the baselines. This demonstrates that TypePro consistently prioritizes the correct data types more effectively, further validating its advantages in ranking accuracy across different baselines.

5.4.5 *Efficiency*. On the ManyTypes4Py dataset, TypePro achieves an average runtime of 2.46 seconds per sample (range: 1.38–8.38 seconds). The slicing operation itself averages only 0.50 seconds (range: 0.33–0.72 seconds), indicating that the more time-consuming part is not code slicing but rather LLM API calls which inevitably introduce network latency. Additionally, statistical analysis of prompt lengths shows an average of 1.96K tokens (range: 0.83K–10.45K). These results demonstrate that both the runtime and token consumption of TypePro fall within an acceptable range, reflecting stable and efficient performance.

5.4.6 *Sensitivity Analysis*. We conducted a sensitivity analysis on the selection of threshold *Score* using the ManyTypes4Py dataset, with values set at 0.1, 0.3, 0.5, 0.7, and 0.9. The corresponding Top-1 exact match accuracies were 86.4%, 87.2%, 88.9%, 86.9%, and 86.9%, respectively. The best performance was achieved when *Score* was set to 0.5. Specifically, a low *Score* value leads to too many irrelevant type information, while a high *Score* makes the strategy too conservative and may exclude valid types that contain optional fields, both of which adversely affect performance.

Answer of RQ1: TypePro achieves higher accuracies and MRR@5 compared to all the baselines. Notably, TypePro improves the Top-1 Exact Match by 7.1 percentage points over the second-best approach, TIGER. This demonstrates the effectiveness of TypePro in various scenarios, especially in the scenarios where fully automation is required (e.g., compilation optimization).

5.5 Results of RQ2

To answer RQ2, we compared the performance of TypePro against various baseline methods across different data types. The data was categorized into three classes: elementary types (Ele), generic types (Gen), and user-defined types (Usr). We recorded the exact match results for TypePro and the baselines at Top-1, Top-3, and Top-5.

The results are shown in Table 4. Overall, TypePro outperforms the baselines with Top-1 accuracies of 93.2%, 82.6%, and 87.6% for Ele, Gen, and Usr, respectively. Compared to the baselines, TypePro showed improvements in Top-1 accuracy, particularly in the Gen and Usr categories, with improvements of 11.0 and 8.3 percentage points, respectively.

However, in the Top-3 and Top-5 scenarios, TIGER slightly outperforms TypePro in user-defined types. The main reason is that, during the candidate type ranking process, TypePro only recommends data types with a fitness score $Score > 0.5$, which may result in fewer than three or five data types being recommended. In such cases, TypePro relies solely on semantic inference from code slices for data type prediction, which can easily lead to hallucinations. Future research directions could guide large language models in type inference to not only consider the structural information of types, but also design appropriate guidance mechanisms to help models understand the semantic relations between the inferred data type and the context, thereby further enhancing the effectiveness of LLM-based approaches.

As for the Ele category, we observe that the three code generation models (CodeT5, UnixCoder, CodeT5+) achieve over 80% accuracy. This is because Ele types dominate the training dataset (52.8% in ManyTypes4Py). When fine-tuning these models, the inherent bias in the data distribution led to their performance on Ele types. Additionally, compared to generation-based methods, classification-based methods clearly underperformed on user-defined types. For example, Type4Py achieved only 45.8% Top-1 accuracy on Usr types, indicating that vocabulary-dependent classification methods struggle to handle types not present in the training vocabulary.

The MRR@5 results demonstrate that TypePro still achieved the best overall performance across the Ele, Gen and Usr. Although TIGER performed slightly better on Top-5 user-defined types, TypePro's MRR@5 remained higher than TIGER's. This is because TIGER's Top-1 accuracy was relatively low and despite its increase from Top-1 to Top-3, Top-1 accuracy carries the highest weight in MRR calculation. These results validate the effectiveness of TypePro across all variable categories.

Answer of RQ2: TypePro achieved high accuracy and MRR@5 across elementary, generic, and user-defined, and significantly outperformed all baselines in Top-1 accuracy, showcases strong and more consistent inference accuracy across all variable categories.

5.6 Results of RQ3

To answer RQ3, we evaluated the effectiveness of TypePro on state-of-the-art open-source and closed-source large language models. Specifically, we selected two closed-source LLMs (GPT-4.1 [1], claude-4 [2]) and three open-source models (qwen3-coder-480b [4], llama-4-128x17b [29], openPangu-Embedded-7B-V1.1 [7]) to evaluate TypePro. Since TypePro does not require fine-tuning of LLMs, we access all LLMs via API calls. We select the largest open-source models in terms of parameter size in the evaluation.

Table 5 shows the results. We can see that although there are slight differences in the Top-1 EM achieved by TypePro across different large language models, the differences are not significant. The lowest (qwen3-coder-plus) reached 83.8%. Additionally, we observe that GPT-4.1 performed the best among the five models we selected, achieving a Top-1 EM of 88.9%. On the other hand, we

found that openPangu-Embedded-7B-V1.1 achieved the lowest accuracy of 59.6%, mainly because the model has fewer parameters. This indicates that TypePro’s approach can generalize across different LLMs.

Table 5. Top-1 Exact Match (%) accuracy of TypePro in different LLMs.

Approach	All	Var	Ret	Arg	Ele	Gen	Usr
qwen3-coder-plus [4]	83.8	83.3	87.2	83.7	89.8	72.5	84.8
llama-4-128x17b [29]	85.6	84.9	89.2	86.0	88.6	80.1	85.7
openPangu-Embedded-7B-V1.1 [7]	59.6	61.1	63.9	52.9	50.8	65.5	74.1
claude-sonnet-4 [2]	88.7	91.2	88.1	92.3	82.3	89.2	88.8
GPT-4.1 [1]	88.9	89.6	88.9	87.2	93.2	82.6	87.6

Answer of RQ3: TypePro can achieve stable precision in different types of LLMs, showing the generality of TypePro.

5.7 Results of RQ4

To answer RQ4, we conducted an ablation study by comparing TypePro with the following baselines:

- **w/o Candidate Type Ranking:** We configured TypePro by removing the candidate type recommendation from the prompt.
- **w/o Inter-Procedural Analysis:** We configured TypePro by leveraging the code slicing process of TypeGen, which does not contain sufficient inter-procedural information.
- **w/o Slicing:** We configured TypePro by leveraging the contextual information considered by TIGER. Specifically, this context is sampled by taking 300 characters both before and after the position of the target variable.
- **w/o All:** We configured TypePro by leveraging the contextual information considered by TIGER, while removing the candidate type recommendation from the prompt.

Table 6. Top-1 Exact Match (%) accuracy of TypePro when different components of the prompt design in TypePro are removed.

Approach	All	Var	Ret	Arg	Ele	Gen	Usr
w/o All	62.0	67.8	54.8	51.0	72.5	56.7	42.9
w/o Slicing	63.9	69.2	58.6	53.5	72.3	59.2	49.5
w/o Candidate Type Ranking	83.3	82.4	87.7	84.1	87.9	77.5	79.3
w/o Inter-Procedural Analysis	74.7	77.7	79.7	66.0	80.4	67.0	71.4
TypePro	87.4	87.9	88.6	85.5	91.5	81.7	85.3

Following the methodology of Peng et al.[25], we compare the performance of TypePro with baseline approaches in terms of Top-1 Exact Match accuracy, as presented in Table 6. It can be observed that the accuracy of w/o Inter-Procedural Analysis decreases to 74.7% on the Top-1 Exact Match. Notably, for the Arg and Gen categories, the accuracy drops by 19.5 and 14.7 percentage points, respectively. Furthermore, when adopting the contextual information extraction process of TIGER (a.k.a., w/o Slicing), the overall accuracy declines by 23.5 percentage points, underscoring the

Table 7. Comparison with baselines on the ManyTypes4TypeScript dataset

Metrics	Approach	Top-1				Top-3				Top-5				MRR@5			
		Var	Ret	Arg	All												
EM	CodeT5	75.3	85.1	69.3	74.1	85.8	89.5	78.7	83.3	87.6	91.0	80.6	85.1	80.5	87.4	74.0	78.6
	CodeT5+	79.5	84.4	70.5	76.3	87.2	89.9	78.9	84.0	88.6	91.5	80.7	85.6	83.4	87.3	74.8	80.3
	UnixCoder	77.8	85.5	68.4	74.9	86.5	90.3	78.8	83.7	88.6	91.8	81.2	85.9	82.2	88.0	73.7	79.4
	TypePro	85.6	89.1	86.7	86.6	88.5	92.1	91.1	90.1	88.5	92.3	91.2	90.2	87.0	90.6	88.9	88.3
BM	CodeT5	75.8	85.5	69.6	74.4	86.0	89.7	79.0	83.5	88.0	91.3	81.0	85.5	80.9	87.8	74.3	79.0
	CodeT5+	79.6	84.7	70.7	76.5	87.6	90.5	79.5	84.6	89.2	92.3	81.4	86.3	83.7	87.7	75.1	80.6
	UnixCoder	77.9	85.8	68.6	75.0	86.8	90.4	79.2	84.0	88.9	92.0	81.7	86.3	82.4	88.2	74.0	79.6
	TypePro	85.7	89.3	86.7	86.6	88.5	92.2	91.1	90.2	88.6	92.4	92.4	90.3	87.1	90.8	88.9	88.4

contribution of inter-procedural analysis to provide sufficient contextual information and therefore improve the accuracy of TypePro. In addition, removing the candidate type ranking process (a.k.a., w/o Candidate Type Ranking) results in a 3.9 percentage point reduction in overall accuracy, with a particularly impact on the Usr category, where the accuracy decreases by 6.0 percentage points. These findings demonstrate that candidate type ranking effectively supplements the LLMs with domain knowledge related to user-defined and third-party library data types, thereby enhancing the overall accuracy of TypePro.

Answer of RQ4: The inter-procedural code slicing in TypePro’s prompt design enhances overall type inference performance by 13.0 percentage points, whereas the candidate type ranking contributes an additional 3.9 percentage point improvement. The above results show the contribution of different components integrated in TypePro.

5.8 Results of RQ5

To answer RQ5, we run TypePro and the baseline models on the ManyTypes4TypeScript dataset to evaluate the effectiveness of type inference in TypeScript. Specifically, we selected CodeT5, CodeT5+, and UnixCoder as the baselines. CLS approaches (i.e., HiTyper and Type4Py) were excluded mainly because their performance is inferior to the GEN and GEN+SIM approaches, as shown in RQ1. In addition, we did not compare with TypeGen and TIGER, as these approaches are designed specific to Python.

Table 7 presents the results. As we can see, TypePro demonstrates comprehensive outperformance over all baseline models on both the Exact Match and Base Match metrics, with only a marginal shortfall observed in the Top-5 Var category. Notably, TypePro surpasses the best baseline (CodeT5+) by 10.3% in Top-1 Exact Match, and achieves an accuracy of 85.6% for the Var category, 89.1% for the Ret category, and 86.7% for the Arg category in Top-1 Exact Match. On the other hand, the improvement in the Ret category for ManyTypes4TypeScript is smaller compared to ManyTypes4Py. This is mainly because more than one-third of Ret in the TypeScript dataset are labeled as void, and function bodies are typically short, allowing CLS-based models to capture the complete context within their input window. In contrast, for the Arg category, TypePro achieves an improvement of approximately 20% over the baselines. These results demonstrate that TypePro exhibits robustness across different programming languages.

Answer of RQ5: TypePro outperforms all baselines on both Exact Match and Base Match metrics in ManyTypes4TypeScript, showing the effectiveness of robustness of TypePro in different programming languages.

6 Limitations

6.1 Limitations of the LLM-Based Method

In our experiments, we used a frequency-based voting mechanism on multiple sampling outputs to generate Top-N results from LLMs, instead of directly ranking candidates by generative likelihood as in CodeT5. The quality of this approach essentially depends on two factors: the diversity of the LLM API responses and the configuration of API parameters (e.g., temperature). Specifically, if the temperature is set too low, the model tends to generate identical or very similar outputs across multiple samplings, increasing the frequency of certain results and reducing the diversity of the Top-N outputs. Conversely, if we set a higher temperature for the LLM, although this increases output diversity, as mentioned earlier, in some cases the candidate type ranking process cannot consistently provide a sufficient number of candidate types due to threshold limitations. In such cases, relying solely on the LLM for semantic type inference leads to limited improvement in Top-3 and Top-5 accuracy of TypeGen compared to baseline methods. In the future, the candidate type ranking process can enhance the sampling around the target variable and provide richer syntactic and semantic guidance for LLMs.

6.2 Limitations in the Experimental Setup

Due to the unavailability of DLInfer’s source code [36], we were unable to include it in our evaluation. Although both DLInfer and our method use static code slicing, we chose to compare our method with TypeGen, which also combines code slicing with LLMs. Nevertheless, this limitation does not affect the validity of our comparative analysis with other baseline methods.

7 Conclusion

This paper introduces TypePro, a novel type inference method for dynamic languages based on code slicing and LLMs. TypePro incorporates inter-procedural information related to variables into static code slicing and traces variables with dependency relationships. It then recommends candidate types by calculating text similarity between the initially generated results from the LLMs and variable declaration-related information. Extensive experiments on the ManyTypes4Py and ManyTypes4TypeScript datasets demonstrate that TypePro achieves improvements across various type categories outperforming existing Python baselines in different programming languages. Notably, TypePro achieves the highest Top-1 accuracy comparing to the baselines, showing the usefulness in the scenarios that require full automation (e.g., compilation speedup).

8 Data Availability

We released the implementation of TypePro, together with the raw data of evaluation results (RQ1–5), on the project website <https://github.com/umxadmin/TypePro>.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62402405), Youth Program of the Xiamen Natural Science Foundation (Grant No. 3502Z202471016), and the Fundamental Research Funds for the Central Universities (Grant No. 20720240087). Huaxun Huang is the corresponding author.

References

- [1] 2025. *ChatGPT*. <https://chat.chatbot.app/gpt4>
- [2] 2025. *Claude Sonnet 4.6*. <https://www.anthropic.com/claude/sonnet>
- [3] 2025. *Mypy: Static Typing for Python*. <https://github.com/python/mypy/>
- [4] 2025. *Qwen Chat*. <https://chat.qwen.ai/>

- 981 [5] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of*
982 *the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) (PLDI 2020).
983 Association for Computing Machinery, New York, NY, USA, 91–105. doi:10.1145/3385412.3385997
- 984 [6] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on*
985 *Object-Oriented Programming*. Springer, 257–281.
- 986 [7] Hanting Chen, Yasheng Wang, Kai Han, Dong Li, Lin Li, Zhenni Bi, Jinpeng Li, Haoyu Wang, Fei Mi, Mingjian Zhu,
987 et al. 2025. Pangu embedded: An efficient dual-system llm reasoner with metacognition. *arXiv preprint arXiv:2505.22375*
988 (2025).
- 989 [8] Michael Emmi and Constantin Enea. 2016. Symbolic abstract data type inference. In *Proceedings of the 43rd Annual*
990 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) (POPL '16).
991 Association for Computing Machinery, New York, NY, USA, 513–525. doi:10.1145/2837614.2837645
- 992 [9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettle-
993 moyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE]
994 <https://arxiv.org/abs/2204.05999>
- 995 [10] Google. 2023. *pytype - A Static Type Analyzer for Python Code*. <https://github.com/google/pytype>
- 996 [11] Salvatore Guarneri and V Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and
997 Reliability Policies for JavaScript Code.. In *USENIX Security Symposium*, Vol. 10. 78–85.
- 998 [12] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal
999 Pre-training for Code Representation. arXiv:2203.03850 [cs.CL] <https://arxiv.org/abs/2203.03850>
- 1000 [13] Yimeng Guo, Zhifei Chen, Lin Chen, Wenjie Xu, Yanhui Li, Yuming Zhou, and Baowen Xu. 2024. Generating Python
1001 Type Annotations from Type Inference: How Far Are We? *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 123 (June
1002 2024), 38 pages. doi:10.1145/3652153
- 1003 [14] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In
1004 *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the*
1005 *foundations of software engineering*. 152–162.
- 1006 [15] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM*
1007 *Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.
- 1008 [16] Hugging Face Inc. 2025. *Hugging Face Hub: A platform for sharing machine learning models, datasets and demos*.
1009 <https://huggingface.co/>
- 1010 [17] Kevin Jesse and Premkumar T. Devanbu. 2022. ManyTypes4TypeScript: a comprehensive TypeScript dataset for
1011 sequence-based type inference. In *Proceedings of the 19th International Conference on Mining Software Repositories*
1012 (Pittsburgh, Pennsylvania) (MSR '22). Association for Computing Machinery, New York, NY, USA, 294–298. doi:10.
1013 1145/3524842.3528507
- 1014 [18] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*.
1015 Cambridge University Press, Cambridge, UK.
- 1016 [19] Microsoft. 2023. *Pyright - Static Type Checker for Python*. <https://github.com/microsoft/pyright>
- 1017 [20] Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. Manytypes4py: A benchmark python dataset for
1018 machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories*
1019 (MSR). IEEE, 585–589.
- 1020 [21] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity
1021 learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*.
1022 2241–2252.
- 1023 [22] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In
1024 *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.
- 1025 [23] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data flow refinement type inference. *Proc. ACM Program.*
1026 *Lang.* 5, POPL, Article 19 (Jan. 2021), 31 pages. doi:10.1145/3434300
- 1027 [24] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets
1028 deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on*
1029 *Software Engineering*. 2019–2030.
- 1030 [25] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative type inference for
1031 python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 988–999.
- 1032 [26] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations*
1033 *and Trends® in Information Retrieval* 3, 4 (2009), 333–389. doi:10.1561/1500000019
- 1034 [27] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical
1035 call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE,
1036 1646–1657.

- 1030 [28] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. 2018. Understanding and automatically preventing
1031 injection attacks on Node. js. In *Network and Distributed System Security Symposium (NDSS)*.
- 1032 [29] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste
1033 Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models.
1034 arXiv 2023. *arXiv preprint arXiv:2302.13971* 10 (2023).
- 1035 [30] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. [Online]. Available: <https://peps.python.org/pep-0484/>. Accessed: 2025-08-27.
- 1036 [31] Chong Wang, Jian Zhang, Yiling Lou, Mingwei Liu, Weisong Sun, Yang Liu, and Xin Peng. 2024. Tiger: A generating-
1037 then-ranking framework for practical python type inference. *arXiv preprint arXiv:2407.02095* (2024).
- 1038 [32] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open
1039 code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).
- 1040 [33] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-
1041 decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- 1042 [34] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph
1043 Neural Networks. arXiv:2005.02161 [cs.PL] <https://arxiv.org/abs/2005.02161>
- 1044 [35] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with
1045 natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of*
1046 *software engineering*. 607–618.
- 1047 [36] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. Dlinfer: Deep learning with static slicing for python
1048 type inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2009–2021.
- 1049 [37] Ming-Ho Yee and Arjun Guha. 2023. Do Machine Learning Models Produce TypeScript Types That Type Check?
1050 Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.ECOOP.2023.37
- 1051
- 1052
- 1053
- 1054
- 1055
- 1056
- 1057
- 1058
- 1059
- 1060
- 1061
- 1062
- 1063
- 1064
- 1065
- 1066
- 1067
- 1068
- 1069
- 1070
- 1071
- 1072
- 1073
- 1074
- 1075
- 1076
- 1077
- 1078