Generative Type Inference for Python

Yun Peng[†], Chaozheng Wang[†], Wenxuan Wang[†], Cuiyun Gao[‡*], Michael R. Lyu[†]
[†] Computer Science and Engineering Department, The Chinese University of Hong Kong, Hong Kong, China
[‡] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
{ypeng, wxwang, lyu}@cse.cuhk.edu.hk, adf111178@gmail.com, gaocuiyun@hit.edu.cn

***Abstract*—Python is a popular dynamic programming language, evidenced by its ranking as the second most commonly used language on GitHub. However, its dynamic type system can lead to potential type errors, leading researchers to explore automatic type inference approaches for Python programs. Existing type inference approaches can be generally grouped into three categories, i.e., rule-based, supervised, and cloze-style approaches. The rule-based type inference approaches can ensure the accuracy of predicted variable types, but they suffer from low coverage problems caused by dynamic features and external calls. Supervised type inference approaches, while feature-agnostic and able to mitigate the low coverage problem, require large, high-quality annotated datasets and are limited to pre-defined types. As zero-shot approaches, the cloze-style approaches reformulate the type inference problem into a fill-in-the-blank problem by leveraging the general knowledge in powerful pre-trained code models. However, their performance is limited since they ignore the domain knowledge from static typing rules which reflect the inference logic. What is more, their predictions are not interpretable, hindering developers' understanding and verification of the results.**

This paper introduces TypeGen, a few-shot generative type inference approach that incorporates static domain knowledge from static analysis. TypeGen creates chain-of-thought (COT) prompts by translating the type inference steps of static analysis into prompts based on the type dependency graphs (TDGs), enabling language models to learn from how static analysis infers types. By combining COT prompts with code slices and type hints, TypeGen constructs example prompts from human annotations. TypeGen only requires very few annotated examples to teach language models to generate similar COT prompts via in-context learning. Moreover, TypeGen enhances the interpretability of results through the use of the *input-explanation-output* strategy, which generates both explanations and type predictions in COT prompts. Experiments show that TypeGen outperforms the best baseline Type4Py by 10.0% for argument type prediction and 22.5% in return value type prediction in terms of top-1 Exact Match by using only five examples. Furthermore, TypeGen achieves substantial improvements of 27% to 84% compared to the zero-shot performance of large language models with parameter sizes ranging from 1.3B to 175B in terms of top-1 Exact Match.

***Index Terms*—type inference, chain-of-thought, generative model**

I. Introduction

With the boom of artificial intelligence and data science, Python is becoming increasingly popular in recent years. As a dynamically typed programming language, Python is famous for its convenience and usability. The dynamic type system makes it possible to reuse the same code snippets for different functionalities, which significantly improves development efficiency. However, this convenience comes with a cost. The dynamic type system poses a threat to the reliability of Python software by introducing more type errors. Oh *et al.* [31] find that 30% of questions raised by developers at GitHub and Stack Overflow are related to type errors. To reduce potential type errors, Python Software Foundation introduces type annotations in a series of Python Enhancement Proposals (PEPs) [19], [20], [43], [53]. Manually annotating types for each variable in Python programs is overwhelming, so many automatic type inference approaches [1], [15], [27], [35], [37] are proposed to infer types statically to release the burden of developers. Automatic type inference approaches work along with static type checkers [29], [38]–[40] to detect potential type errors for Python programs [31], [36].

The earliest proposed automatic type inference approaches are rule-based, as described in previous work [2], [5], [7], [10], [17], [33]. These approaches rely on pre-defined typing rules and static analysis to accurately infer types. However, they are limited by the low coverage problem since the types of many variables in Python programs cannot be resolved statically.

Inspired by the remarkable achievements of deep learning in the natural language processing (NLP) field, supervised type inference approaches [1], [18], [27], [37], [47] take the code context of the target variable as input and leverage deep learning models to classify the context into one type. They can naturally avoid the low coverage problem of rule-based approaches, as deep learning models are feature-agnostic and make predictions based on probability rather than rules. Taking advantage of identifiers in code, supervised type inference approaches are quite effective after training on a large dataset of annotated code. Despite the effectiveness, supervised type inference approaches based on classification methods classify inputs into pre-defined type categories and perform badly on rare types. Peng *et al.* [34] propose a hybrid type inference approach HiTyper to mitigate these problems by using deep learning models to recommend type predictions for static analysis. However, deep learning models in supervised approaches require large high-quality datasets of type annotations, which needs substantial human efforts.

Cloze-style type inference approaches [8], [9], [13], [14], [45] transform the type inference problem into a fill-in-the-blank problem by adding masks on the locations of type annotations in code. These approaches are well-aligned with the pre-training objectives of pre-trained code models and do

[*] Cuiyun Gao is the corresponding author.

not require large datasets, making them suitable for zero-shot settings. However, they face the following challenges:

1) Lack of static domain knowledge. Cloze-style approaches are characterized by the insertion of masks in source code, allowing pre-trained code models to predict the missing type information. While they have the advantage of not requiring large datasets, unlike supervised approaches, they rely solely on the general knowledge that pre-trained code models acquire during the pre-training phase. Consequently, their performance may be suboptimal, as they lack an understanding of how types are constructed based on typing rules.

2) Lack of interpretability. Current learning-based type inference approaches, including supervised and cloze-style approaches, adopt the *input-output* methodology, taking code as input and outputting single types. However, they provide no idea about how deep learning models reach the *output* types from the *input* code. This lack of transparency makes it challenging for developers to comprehend and validate the predicted types, particularly when there are insufficient static constraints.

Our work. We propose TYPEGEN, the first few-shot generative type inference approach for Python programs. TYPEGEN has four phases, including code slicing, type hint collection, chain-of-thought (COT) prompt construction, and type generation. In the code slicing phase, TYPEGEN generates type dependency graphs (TDGs) and builds code slices based on TDG as the contexts of target variables, i.e., the variables whose types need to be inferred. In the type hints collection phase, TYPEGEN collects all available user-defined types and third-party types as type hints via import analysis to provide additional knowledge that does not exist in code slices. In the COT prompt construction phase, TYPEGEN translates the inference steps of static analysis for target variables into a COT prompts [46]. The code slices, type hints and COT prompts generated in the first three phases are combined as the *example prompts*, which provides rich static domain knowledge. In the last type generation phase, TYPEGEN adopts the *in-context learning* (ICL) methodology and constructs the input prompt by concatenating several example prompts and the *target variable prompt*, which includes code slice as well as type hints of the target variable. A language model is then invoked to complete the input prompt with the COT prompt of the target variable. With both explanations and predicted types in the generated COT prompts, TYPEGEN can improve the interpretability of results.

We evaluate TYPEGEN on the widely-used ManyTypes4Py dataset [26]. Our experiment results show that TYPEGEN outperforms the most advanced baseline Type4Py [27] by 10.0% for argument types and 22.5% for return value types in terms of top-1 Exact Match. Furthermore, we observe that TYPEGEN can achieve improvements of 27% $\sim$ 84% over the zero-shot performance of language models with parameter sizes ranging from 1.3B to 175B in terms of top-1 Exact Match, which are $2\times \sim 3\times$ of the improvements achieved by the standard ICL method without static domain knowledge.

Contributions. We summarize our contributions as follows.

- To the best of our knowledge, we propose the first few-shot generative type inference approach named TYPEGEN for Python.
- We propose a novel prompt design to incorporate different static domain knowledge into language models, which includes code slices, type hints, and COT prompts.
- Extensive experiments demonstrate the effectiveness of TYPEGEN compared with supervised and cloze-style type inference approaches, as well as the capability of TYPEGEN on language models with different parameter sizes.

II. BACKGROUND AND RELATED WORK

We classify existing type inference approaches into three categories: rule-based, supervised and cloze-style approaches, and present an overview of three kinds of type inference approaches in Fig. 1.

A. Rule-based Type Inference

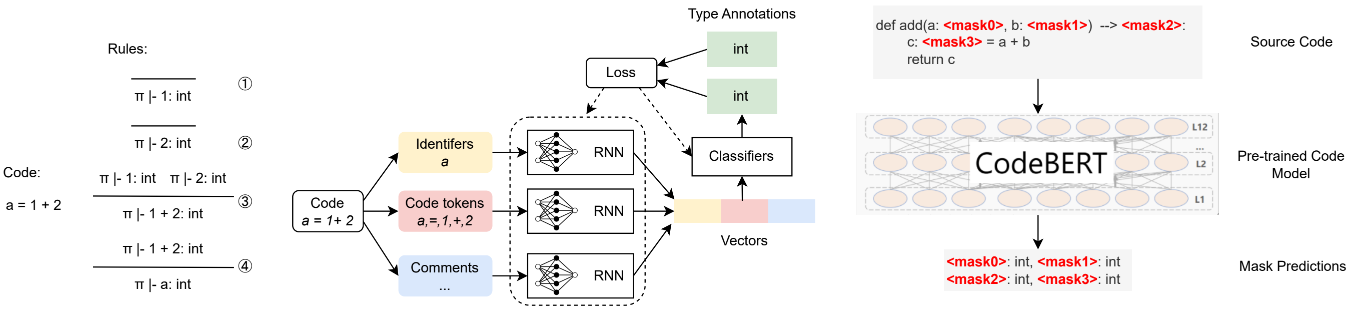
Rule-based approaches for type inference rely on predefined rules to determine the types of variables. Fig. 1(1) shows an example where four rules are associated with the type inference of variable *a*. Each rule has premises (above the line) and conclusions (below the line). A rule can be triggered only if all premises are known, and then the result type is given based on the conclusion.

To address the need for static type hints in dynamically typed programming languages, various approaches have been proposed for type inference and checking, such as Pyright [39] and Pylance from Microsoft, Pyre from Meta [38], Pytype from Google [40], and Python's official type checker mypy [29]. In addition to industry tools, some academic approaches have been proposed for type inference in different programming languages, such as Python and JavaScript [2], [5], [7], [10], [17], [33]. While these approaches are quite accurate, they are limited by the low coverage problem caused by dynamic features and external calls [34].

B. Supervised Type Inference

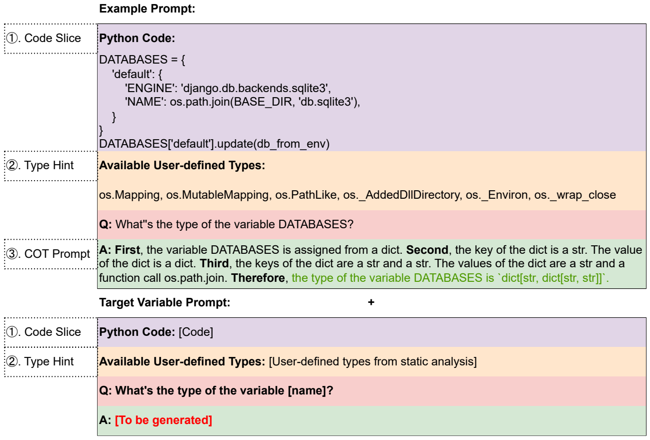
Supervised type inference approaches utilizing deep learning models have made significant progress in predicting types for dynamic languages. Fig. 1(2) illustrates the typical process of these approaches: features are extracted from code and encoded into vectors using deep learning models such as recurrent neural networks (RNNs) [42]. A classifier is then used to classify the vectors into pre-defined types. The loss is calculated based on the type prediction of the classifier and the human type annotation, and the parameters of the deep learning models and classifier are updated via back-propagation.

Allamanis *et al.* [1] adopt an open vocabulary model which encodes code as graphs to predict types. Pradel *et al.* [37] uses multiple RNN models to encode features such as identifiers and code tokens. Mir *et al.* [28] improve the top-1 accuracy via a deep similarity clustering algorithm. Wei *et al.* [48] propose to use graph neural networks to predict



(1) Rule-based Type Inference (2) Supervised Type Inference (3) Cloze-Style Type Inference

Fig. 1: Three kinds of type inference approaches.


①. Code Slice
②. Type Hint
③. COT Prompt
①. Code Slice
②. Type Hint
+

Example Prompt:

Python Code:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
DATABASES['default'].update(db_from_env)
```

Available User-defined Types:

os.Mapping, os.MutableMapping, os.PathLike, os._AddedDllDirectory, os._Environ, os._wrap_close

Q: What''s the type of the variable DATABASES?

A: First, the variable DATABASES is assigned from a dict. **Second**, the key of the dict is a str. The value of the dict is a dict. **Third**, the keys of the dict are a str and a str. The values of the dict are a str and a function call os.path.join. **Therefore**, the type of the variable DATABASES is `dict[str, dict[str, str]]`.

Target Variable Prompt:

Python Code: [Code]

Available User-defined Types: [User-defined types from static analysis]

Q: What's the type of the variable [name]?

A: [To be generated]

Fig. 2: Input prompt with example from the code in Fig. 4.

types. Jesse *et al.* [18] propose TypeBERT by reformulating type prediction as a NER problem. Peng *et al.* [35] propose HiTyper, which uses deep learning models to recommend types for static inference. While these approaches achieve satisfying performance, they require high-quality datasets for training, which can be difficult to obtain in the wild. Furthermore, supervised type inference approaches only provide predictions without any explanation about how they infer the types, making it challenging for developers to understand and verify the results.

C. Cloze-Style Type Inference

To enhance the reliability of Python software and prevent potential type errors, the Python Software Foundation has introduced a series of Python Enhancement Proposals (PEPs) [19], [20], [43], [53] that enable developers to add static type annotations to their code. As these annotations become part of the code, they can be leveraged by pre-trained code models that are trained on a vast amount of open-source Python programs. Cloze-style type inference approaches, as illustrated in Fig.1(3), add masks on the locations of type annotations in the code and invoke pre-trained code models to fill in the masks with predicted types.

All pre-trained code models with Masked Language Modeling (MLM) training objectives such as CodeBERT [8], GraphCodeBERT [14] and CodeT5 [45] can be naturally used to predict type annotations. Most recently, UniXcoder [13] is a unified cross-modal pre-trained model to support both code-related understanding and generation tasks. InCoder [9] is a large code generative model that can refill arbitrary regions of code. Leveraging pre-trained code models, cloze-style type inference approaches can be readily implemented. However, they still exhibit limited performance as they solely rely on the general knowledge of pre-trained code models. These approaches can hardly handle complicated types without domain knowledge from static typing rules, and their predictions lack interpretability without explanations.

III. Generative Type Inference

A. Overview

As a generative approach, TypeGen first generates domain knowledge-aware prompts and then inputs them into language models for type prediction. To achieve this, TypeGen adopts the widely-used *in-context learning* methodology [6]. This methodology provides a few example questions and answers as demonstrations for the language model and then asks the answer for a new question. Leveraging this methodology, TypeGen constructs the input prompt by adding some domain knowledge-aware *example prompts* (example questions and answers) before the *target variable prompt* (new question). Fig. 2 illustrates an input prompt with an example from the code in Fig. 4. The domain-aware *example prompts* include three parts: code slice, type hint and COT prompt, as shown in Fig. 2. They are designed to incorporate different static domain knowledge for language models. Specifically, the **code slice** isolates the statements contributing to the construction of the type for the target variable, with the remaining unrelated statements removed. The **type hint** includes external knowledge that is specific to different code slices, including user-defined types and third-party types. The **COT prompt** indicates the inference steps of static analysis, aiming at teaching language models how to infer types. The *target variable prompt* contains only the code slice and the type hint of the target variable.

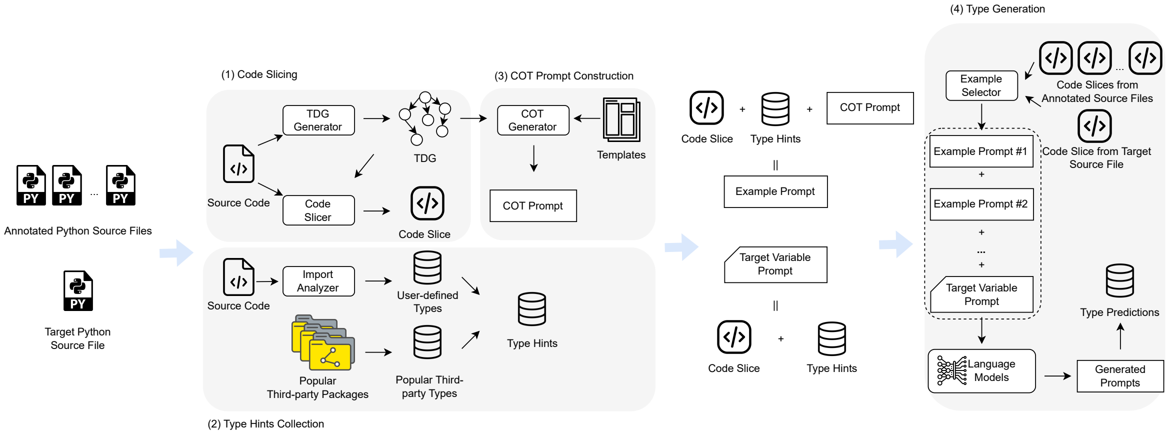


Fig. 3: The overview of TypeGen.

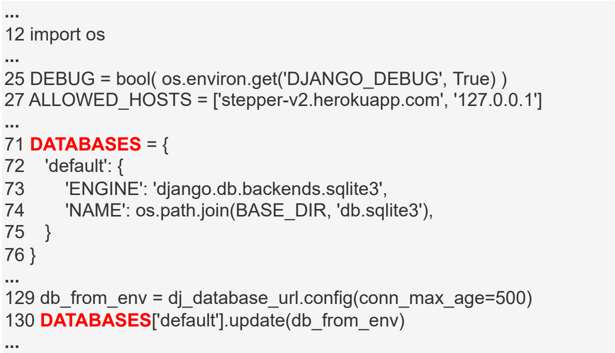
We provide an overview of TypeGen's workflow in Fig. 3. To start, TypeGen takes a set of annotated Python source files to select examples and a target Python source file where the target variable is located. For each target source file, TypeGen generates an input prompt that incorporates domain-aware example prompts and the target variable prompt. A language model is then employed to produce the COT prompt, which includes both the predicted type and corresponding explanations. To generate domain-aware example prompts, TypeGen conducts three phases: (1) *code slicing*, (2) *type hint collection*, and (3) *COT prompt construction* to generate the code slice, type hint, and COT prompt, respectively. Finally, in the (4) *Type Generation* phase, TypeGen leverages in-context learning to infer the types of target variables.

B. Code Slicing

The code slicing phase aims at identifying the code statements related to the target variable based on the type dependency graph (TDG) which indicates the type dependencies among variables [35].

1) TDG Generation: In order to extract the type dependencies of the target variable, TypeGen generates type dependency graphs (TDGs) using HiTyper [35]. A TDG is a directed graph (N, E), where N is the node set and E is the edge set. Each node $n \in N$ in TDG represents a variable (symbol node), an operation (operation node), or a type (type node), while each edge $e \in E$ indicates that the type of the output node depends on the type of the input node. If the target variable is an argument, a return value or a local variable in the function, TypeGen generates the TDG for the specific function. Otherwise, if the target variable is a global variable, TypeGen generates the TDG for all statements in the source file except class definitions and function definitions. To better illustrate the code slicing phase, we give a code example in Fig. 4 and its sliced TDG in Fig. 5, where the target variable is "DATABASES".

To refine the initial TDG, TypeGen prunes the nodes that do not have any type dependency with the target variable. For instance, in the code presented in Fig.4, TypeGen removes the nodes generated from statements at lines 25, 27, 129, etc. TypeGen then merges identical symbol nodes that are directly connected, since they represent different occurrences of the same variable. To generate the sliced TDG, TypeGen locates the sub-graph where the target variable is defined in the refined TDG. For the example in Fig.4, TypeGen identifies the target variable node "DATABASES" at line 71 and extracts the reachable sub-graph of the refined TDG as the sliced TDG, as illustrated in Fig. 5.



```
...
12 import os
...
25 DEBUG = bool( os.environ.get('DJANGO_DEBUG', True) )
27 ALLOWED_HOSTS = ['stepper-v2.herokuapp.com', '127.0.0.1']
...
71 DATABASES = {
72   'default': {
73       'ENGINE': 'django.db.backends.sqlite3',
74       'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
75   }
76 }
...
129 db_from_env = dj_database_url.config(conn_max_age=500)
130 DATABASES['default'].update(db_from_env)
...
```

Fig. 4: The source code for the example prompt in Fig. 2, where DATABASES is the target variable.

2) Code Slice Generation: Using the sliced TDG, TypeGen generates a code slice from the original input source code, containing only the statements that have type dependencies with the target variable. In this way, TypeGen reduces the entire function into a smaller code slice that includes only the information relevant to the type inference of the target variable. TypeGen employs different strategies for generating code slices for local variables, return values, and arguments.

Local Variables and Return Values. To generate code slices for local variables and return values, TypeGen leverages the clear definitions that indicate how their types are constructed in the code. TypeGen initiates the process by starting from the definition node of the target variable and traversing **backward** on the TDG, i.e., in the opposite direction to the

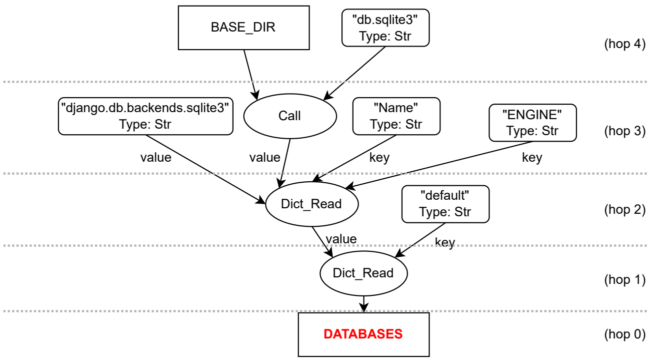


Fig. 5: The sliced type dependency graph (TDG) of code in Fig. 4.

TDG edges, to include all the nodes that contribute to the definition of the target variable. The distance of each node from the target variable node is determined by the number of edges (hops) between them. A maximum threshold is also set to prevent infinite loops, and any nodes with distances exceeding the threshold are removed. Once the traversal is completed, TypeGen combines the corresponding statements of the remaining nodes in the TDG to form the code slice.

Arguments. To facilitate the generation of code slices for function arguments, TypeGen adopts a different approach, since arguments do not have explicit definitions. TypeGen collects the usages of function arguments, as variable usages often provide hints about their types. For instance, operations such as *open* are usually associated with *File* types. Thus, TypeGen begins with all nodes of the target argument and traverses **forward** on the TDG, i.e., in the same direction as the TDG edges, to include nodes that use the target argument. TypeGen generates code slices for nodes with distances within a specified maximum threshold, similar to local variables and return values.

C. Type Hints Collection

Unlike built-in types that are available in every source file, user-defined and third-party types are specific to each source file and are defined through class definitions and import statements. However, general knowledge bases in language models do not cover this specific domain knowledge [35]. To address this issue, TypeGen collects all the user-defined and third-party types that are imported to the current source file as type hints.

To identify user-defined types, TypeGen performs an import analysis on the current source directory. First, it collects all class definitions in the current source file as user-defined types. Then, it examines the import statements to determine which source files are imported in the current file and adds their class definitions to the list of user-defined types. For third-party types, following previous study [51], TypeGen downloads the top 10,000 popular Python packages ranked by libraries.io [23] and employs the same import analysis technique to identify third-party types. All third-party types collected by TypeGen are stored in a database, which can be queried by TypeGen to identify the available third-party types based on the import statements in the current source file.

TABLE I: Chain-of-Thought Prompt Template. [NAME] indicates the name of symbol nodes, [OP] indicates the name of operation nodes and [TYPE] indicates the name of type nodes. [GTTYPE] indicates the annotated type for the target variable. DD-RV and DD-A indicate the dependency description for local variables and return values, and arguments, respectively.

Part	Type	Template
DD-RV	Operation→Symbol	The variable/return value of [NAME] is assigned from [OP] operation.
	Symbol→Symbol	The variable/return value of [NAME] is assigned from variable [NAME].
	Type→Symbol	The variable/return value of [NAME] is assigned from [TYPE].
	Operation→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are [OP] operation.
	Symbol→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are variable [NAME].
	Type→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are [TYPE].
DD-A	Usage	The argument [NAME] is used in [OP]/[NAME].
	Naming	Based on the naming convention, it is reasonable to assume that the type of the argument [NAME] is [GTTYPE].
Con	Conclusion	Therefore, the type of the variable/return value of/argument [NAME] is [GTTYPE].

When generating type hints, TypeGen analyzes all the import statements in the current source file. If a user-defined package is imported, TypeGen directly conducts import analysis to gather all available user-defined types. If a third-party package is imported, TypeGen queries the database and obtains all available third-party types. All available types are concatenated to build the type hint, with an example shown in Fig. 2 (highlighted in orange color). To prevent excessively long type hints, TypeGen imposes a maximum threshold (set to 50 in this paper) for the number of collected types. Considering the scarcity of user-defined types, TypeGen prioritizes the importance of types based on the following order: "user-defined types in current source file > user-defined types in other source files > third-party types".

D. Chain-of-Thought Prompt Construction

TypeGen translates the type inference steps of static analysis into chain-of-thought (COT) prompts [46] to involve static domain knowledge of how a type is constructed, where a COT is a series of intermediate reasoning steps [46]. To generate COT prompts, TypeGen utilizes the sliced TDG produced by the *code slicing* phase.

Given the sliced TDG, TypeGen first organizes the nodes into different hops according to their distance from the target variable node. The hop of the node of the target variable is set to 0, and the hops of other nodes are determined by their distance, as shown in Fig. 5. The COT prompt constructed by TypeGen includes *dependency description* and *conclu-*

sion. The *dependency description* explains how the type is inferred, whereas the *conclusion* gives the final type prediction. TYPEGEN translates each hop in the TDG into a sentence of *dependency description* and generates the *conclusion* based on the annotated types from developers. Following the recent study [52], we summarize the powerful templates of COT prompts from the zero-shot outputs of language models and present them in Table I. For the *conclusion*, TYPEGEN fills in the variable name and annotated type into the template. Regarding the *dependency description*, TYPEGEN employs different prompt templates for local variables, return values, and arguments as TYPEGEN utilizes different information for them in the sliced TDGs in Sec. III-B2.

Local Variables and Return Values. Local variables and return values have clear definitions in the code, so it is possible to construct a comprehensive description of how the type of target variable should be inferred. To construct the *dependency description* for them, TYPEGEN starts with the edges connected to the target variable node and traverses backward on the TDG to translate each edge into a sentence of *dependency description*. Since there are three major types of nodes in the TDG, we design six templates for six kinds of edges in TDG to generate *dependency descriptions*, as shown in the first part of Table I. Note that the type node cannot be the output node as its type does not depend on other nodes. Take the TDG in Fig. 5 as an example. There is an edge from operation node *Dict_Read* to symbol node *DATABASES*. For this edge, TYPEGEN adopts the *Operation→Symbol* template and generates a sentence "*The variable DATABASES is assigned from a dict*". There is also an ordinal number at the beginning of each sentence to indicate one inference step. Sentences generated from all edges are concatenated together according to the backward traversal order to form a complete *dependency description*.

Arguments. As arguments usually do not have clear definitions in the code, it is difficult for static analysis to infer their types. Rather than providing solid definition information, TYPEGEN provides usage and naming information as hints for type prediction. We present the usage template and naming template in the second part of Table I. For usage information, TYPEGEN collects all nodes in the sliced TDG and constructs the sentence "*The argument ... is used in ...*". For the naming information, TYPEGEN adds the sentence "*Based on the naming convention, it is reasonable to assume that the type of the argument is ...*" to remind language models to consider the argument name. These two sentences form a complete *dependency description* for arguments.

The generated *dependency description* and *conclusion* are finally combined together to form a complete COT prompt. Fig. 2 shows the COT prompt generated by TYPEGEN for the code example in Fig. 4, highlighted in green color.

E. Type Generation

For each input source file and target variable, TYPEGEN generates its corresponding code slice and selects a set of code slices from the training set as examples based on BM25 similarity [41]. BM25 similarity calculates the token similarity between two code slices and has been widely used in recent studies [12], [21]. Following previous work [24], the example prompts are ordered based on the BM25 similarity of code slices: $\{EP_1, ..., EP_n\} (\forall i \in [1, n)\ BM25(EP_i, TP) \leq BM25(EP_{i+1}, TP))$ where EP is an example prompt and TP is the target variable prompt. The example prompts are then combined with the target variable prompt to form the complete input prompt, as shown in Fig. 2.

TABLE II: The statistics of the ManyTypes4Py dataset. 'Arg" indicates function arguments, "Ret" indicates function return values, "Var" indicates global and local variables, "Ele" indicates elementary types, "Gen" indicates generic types, and "Usr" indicates user-defined types and third-party types.

Dataset	Total	Arg	Ret	Var	Ele	Gen	Usr
Training Set	242,954 100%	48,461 20.0%	22,034 9.1%	172,459 70.9%	128,006 52.7%	67,185 27.6%	47,763 19.7%
Test Set	85,205 100%	16,700 19.6%	7,754 9.1%	60,751 71.3%	44,605 52.5%	23,310 27.4%	17,290 20.1%
Sampled Test Set	10,000 100%	1,995 20.0%	914 9.1%	7,091 70.9%	5,199 52.0%	2,748 27.5%	2,053 20.5%

Given the examples in the input prompt, language models can learn to generate a similar COT prompt for the target variable. To facilitate the automatic evaluation of the generated COT prompts, TYPEGEN surrounds type predictions in example COT prompts with quotes, such as d̀ict[str, dict[str, str]]̀in Fig. 2. This allows language models to learn to emphasize type predictions in generated COT prompts by adding quotes. Ultimately, TYPEGEN extracts the content within the quotes as the types predicted by the language models.

IV. EXPERIMENT SETUP

A. Dataset

We follow previous work [28], [35] and evaluate our approach on the ManyTypes4Py dataset [26] by splitting the dataset into a training set and a test set with an 80:20 ratio. We use the training set to train the baseline models and select examples for TYPEGEN and evaluate the performance of TYPEGEN and baselines in the test set. In order to accommodate the computational resource limitations, we further sample 10,000 instances from the test set during the evaluation of large language models. Table II presents the statistics of the experimental datasets.

B. Baselines

We choose the following four type inference approaches as our baselines:

- **TypeBERT** [18] is a supervised type inference approach. It reformulates the type inference problem into a Named Entity Recognition (NER) problem and regards types as labels.
- **TypeWriter** [37] is a supervised type inference approach. It extracts different code features, such as identifiers and code tokens, and utilizes four RNNs to encode the extracted features and make type predictions.

TABLE III: The statistics of language models used in the evaluation.

Model	#Parameters	Training Dataset	Type
CodeT5	220M/770M	CodeSearchNet & BigQuery	Generative & Infilling
UniXcoder	126M	CodeSearchNet	Infilling
GPT-Neo	1.3B/2.7B	The Pile	Generative
InCoder	1.3B/6.7B	GitHub & Stack Overflow	Generative & Infilling
CodeGen	6B	The Pile, BigQuery & GitHub	Generative
GPT-J	6.7B	The Pile	Generative
GPT-3.5	175B	-	Generative
ChatGPT	175B	-	Generative

- **Type4Py** [28] is a supervised type inference approach. It builds different type clusters and classifies a new Python program into one of the type clusters to determine the type.
- **HiTyper** [35] is a hybrid type inference approach. It builds a type dependency graph (TDG) for each target variable and utilizes both static analysis and neural prediction to fill the blanks in the TDG and finally outputs the validated types.

We choose three popular pre-trained code models, including CodeT5 [45], UniXcoder [13], and InCoder [9] to represent the performance of cloze-style type inference approaches. Besides, we choose GPT-Neo [3], GPT-J [44], CodeGen [30], GPT-3.5 [4] and ChatGPT [32] to evaluate the performance of TypeGen on language models with different parameter sizes. We present the statistics of all the language models in Table III.

C. Metrics

We use two commonly used metrics in previous work [1], [28], [35] to evaluate the performance of TypeGen and other baselines:

- **Exact Match** is defined by the ratio of type predictions made by an approach that **exactly** match type annotations from developers.
- **Match to Parametric** is defined by the ratio of type predictions made by an approach that **share the common outmost type** with type annotations from developers.

For example, *List[int]* and *List[str]* are considered Match to Parametric but not Exact Match since they are different types while sharing the same outmost type *List*.

D. Implementation

For the four type inference baselines TypeBERT [18], TypeWriter [37], Type4Py [28] and HiTyper [35], we directly use the replication packages released by the authors and other researchers. For all the language models except GPT-3.5 and ChatGPT, we download them from HuggingFace Hub [16] and deploy them locally. Following the previous work [28], [35], we adopt the generated sentences with top-5 probabilities as predictions. For GPT-3.5 and ChatGPT, we use the public APIs provided by OpenAI under engine "*text-davinci-003*" and "*gpt-3.5-turbo-0301*", respectively. We acquire 50 samples with a temperature of 1.0 for each target variable and rank the top-5 predictions according to the occurrence frequency, following the work [49], [50]. We choose the maximum distance threshold of TDG nodes at 3 for the code slicing phase of TypeGen, as previous studies [1], [28] only consider types with nested levels smaller than 3. All experiments are conducted on a Linux machine (Ubuntu 18.04) with one 112-core Intel Xeon Gold 6348 CPU@ 2.60GHz, two NVIDIA A100-80GB GPUs, and 1TB RAM.

V. Evaluation

A. Research Questions

In the evaluation, we focus on the following four research questions:

- **RQ1**: How effective is TypeGen in type inference compared with existing approaches?
- **RQ2**: How capable is TypeGen in language models with different parameter sizes?
- **RQ3**: What are the impacts of different parts in the prompt design of TypeGen?
- **RQ4**: What are the impacts of different examples in TypeGen?

To study RQ1, we conduct experiments on both TypeGen and baseline approaches with the entire test set (85,205 instances), aiming to comprehensively verify the effectiveness of TypeGen against state-of-the-art type inference techniques. However, due to limited computational resources, we only use the sampled test set (10,000 instances) for RQ2-4. For RQ2, we evaluate six language models under TypeGen to examine the tool's effectiveness across language models with different parameter sizes. We also include two additional settings: **Zero-Shot** and **Standard ICL**. In the **Zero-Shot** setting, we do not provide any example and only use the source code of the target variable as the input prompt for language models in the type prediction. The Zero-Shot setting tests the basic performance of language models on type prediction. In the **Standard ICL** setting, we provide three fixed example prompts before the target variable prompt and use only source code in the input prompts, which is the same with recent study [22]. The Standard ICL setting indicates the basic performance of language models with the *in-context learning* methodology. To fairly compare TypeGen with the Standard ICL setting, we also set the number of examples in TypeGen to 3 in RQ2. For RQ3, we remove different parts of the prompt design in TypeGen to study the impacts of each part. For RQ4, we vary the number of examples and the example selection method to investigate the impacts of different examples. We choose ChatGPT as the base model of TypeGen and use five examples in TypeGen in all the RQs except RQ2.

B. RQ1: Effectiveness of TypeGen

1) Comparison with Supervised Approaches: We compare TypeGen with three supervised type inference approaches, namely TypeBERT, TypeWriter, and Type4Py. The results

TABLE IV: The performance of TypeGen along with the baselines under four types of variables in terms of Top-1,3,5 Exact Match (%) and Match to Parametric (%). “Arg”, “Ret”, “Var”, and “All” indicate function arguments, function return values, global and local variables, and all of above, respectively. Under each metric the best performance is marked as gray .

Metric	Category	Approach	Top-1				Top-3				Top-5			
			Arg	Ret	Var	All	Arg	Ret	Var	All	Arg	Ret	Var	All
Exact Match (%)	Supervised	TypeBERT	28.0	38.5	51.1	45.4	34.8	52.6	55.8	51.4	36.5	57.1	58.6	54.1
		TypeWriter	53.3	52.8	-	-	61.1	60.7	-	-	65.8	65.3	-	-
		Type4Py	66.5	56.1	82.0	76.6	72.0	59.2	83.8	79.3	73.8	60.7	84.3	80.1
	Cloze Style	InCoder-1.3B	20.9	20.5	15.1	16.7	21.3	20.8	15.5	17.1	21.3	21.0	15.6	17.2
		InCoder-6.7B	24.1	42.0	18.7	21.9	24.6	42.7	19.1	22.3	24.7	43.1	19.2	22.4
		UniXcoder	55.0	49.2	35.9	40.9	66.9	64.6	42.1	49.0	70.6	69.8	45.2	52.4
		CodeT5-base	51.1	57.6	21.7	30.7	59.3	64.4	28.0	37.4	62.0	66.9	30.7	40.1
		CodeT5-large	56.2	60.2	44.7	48.4	61.6	64.5	50.4	53.9	63.9	66.3	53.4	56.6
	Generative	TypeGen	73.1	68.7	82.2	79.2	81.0	77.1	87.9	85.6	82.7	79.1	89.1	87.0
Match to Parametric (%)	Supervised	TypeBERT	29.8	41.4	54.0	48.1	36.0	55.9	58.0	53.5	37.7	60.8	61.2	56.5
		TypeWriter	54.4	54.1	-	-	63.4	63.5	-	-	68.8	69.3	-	-
		Type4Py	68.0	59.0	86.2	80.2	74.1	64.1	88.3	83.3	75.9	66.3	88.8	84.3
	Cloze Style	InCoder-1.3B	22.9	22.8	18.7	19.9	23.3	23.1	19.1	20.3	23.4	23.3	19.2	20.4
		InCoder-6.7B	28.8	51.6	25.0	28.1	29.3	52.1	25.3	28.5	29.4	52.5	25.3	28.6
		UniXcoder	61.9	61.8	44.3	49.3	72.3	76.0	51.2	57.6	75.0	80.1	53.8	60.4
		CodeT5-base	54.8	66.7	27.7	36.6	62.9	74.2	34.4	43.6	65.6	76.4	37.1	46.3
		CodeT5-large	61.4	69.4	55.7	58.0	66.8	74.3	61.2	63.5	68.9	76.2	63.7	65.9
	Generative	TypeGen	78.7	75.6	91.2	87.3	84.9	83.0	93.7	91.0	86.1	84.5	94.1	91.7

TABLE V: The performance of HiTyper with different base models under four types of variables. Variable categories are the same with Table IV.

Metric	Base Model	Arg	Ret	Var	All
Exact Match (%)	-	8.0	43.5	65.7	52.4
	Type4Py	73.5	73.4	90.6	85.7
	TypeGen	84.9	77.9	90.5	88.3
Match to Parametric (%)	-	8.4	52.7	70.2	56.5
	Type4Py	76.1	83.4	95.3	90.1
	TypeGen	87.4	87.3	95.3	93.1

are presented in Table IV, where we report the top-1, top-3, and top-5 Exact Match and Match to Parametric for four categories of variables. It is worth noting that TypeWriter is designed solely for argument and return value type predictions. Analyzing the top-1 prediction results in Table IV, we observe that TypeGen outperforms the best supervised approach, Type4Py, by 10.0% for argument type prediction and 22.5% for return value type prediction in terms of Exact Match. This improvement of TypeGen over Type4Py is even more significant for top-5 predictions, where TypeGen outperforms Type4Py by 12.1% for argument type prediction and 30.3% for return type prediction in terms of Exact Match. Moreover, when considering Match to Parametric, TypeGen achieves a consistent improvement of 8.7% $\sim$ 9.3% on overall variables than Type4Py. These results demonstrate that, even with few annotated examples, the generative type inference approach TypeGen is more effective than supervised approaches such as Type4Py. We also observe that TypeGen does not perform much better than Type4Py on predicting local variables. This can be attributed to the lower difficulty of type inference for local variables compared to arguments and return values, so static domain knowledge incorporated by TypeGen provides limited improvements. This can also be verified by Table IV, where all the supervised approaches obtained much higher performance on local variables than arguments and return values.

2) *Comparison with Cloze-Style Approaches:* We compare TypeGen with cloze-style approaches, namely InCoder, UniXcoder and CodeT5, and present the results in Table IV. Our observations indicate that, in general, cloze-style approaches perform worse than supervised approaches due to their lack of domain knowledge from data and static analysis. By introducing five annotated examples and incorporating static knowledge, TypeGen outperforms the best cloze-style approach CodeT5-large by 63.6% on overall top-1 Exact Match and 53.7% on overall top-5 Exact Match. This suggests that incorporating domain knowledge from static analysis with a few examples can largely improve the performance of type inference.

3) *Comparison in Hybrid Approach HiTyper:* As HiTyper is a hybrid approach, we study its performance with the best supervised approach Type4Py and TypeGen, and present the experiment results in Table V. For Type4Py and TypeGen, we use their top-5 predictions as type recommendations since HiTyper can reject wrong types. The results show that HiTyper

TABLE VI: The performance of different language models under three settings for all variables in terms of Top-1,3,5 Exact Match (%). △ indicates the improvement of Standard ICL and TypeGen over the Zero-Shot setting.

Base Model	**Approach**	**Top-1** (△)	**Top-3** (△)	**Top-5** (△)
GPT-Neo (1.3B)	Zero-Shot	31.5	40.6	42.8
	Standard ICL	44.0 (40%)	50.0 (23%)	50.8 (19%)
	TypeGen	57.0 (81%)	61.5 (51%)	62.8 (47%)
GPT-Neo (2.7B)	Zero-Shot	43.2	50.0	51.9
	Standard ICL	46.6 (8%)	52.3 (5%)	52.8 (2%)
	TypeGen	55.5 (28%)	61.9 (24%)	63.0 (21%)
GPT-J (6.7B)	Zero-Shot	42.4	43.7	43.9
	Standard ICL	50.8 (20%)	54.9 (26%)	55.3 (26%)
	TypeGen	62.7 (48%)	67.3 (54%)	68.4 (56%)
CodeGen (6B)	Zero-Shot	34.7	44.0	45.5
	Standard ICL	54.1 (56%)	60.5 (38%)	61.9 (36%)
	TypeGen	63.7 (84%)	69.1 (57%)	70.8 (56%)
GPT-3.5 (175B)	Zero-Shot	62.0	65.4	66.3
	Standard ICL	69.7 (12%)	74.2 (13%)	75.8 (14%)
	TypeGen	78.9 (27%)	85.0 (30%)	86.2 (30%)
ChatGPT (175B)	Zero-Shot	61.3	66.1	67.5
	Standard ICL	68.0 (11%)	71.8 (9%)	73.1 (8%)
	TypeGen	78.8 (29%)	85.3 (29%)	86.7 (28%)

performs poorly when there is no base model, particularly in argument type inference, where it achieves an Exact Match of only 8%. This verifies the low coverage problem of static analysis. When associating with base models, HiTyper with TypeGen still outperforms HiTyper with Type4Py by 15.5% for argument type inference and 6.1% for return value inference. This indicates that the performance gap of TypeGen over Type4Py cannot be bridged by simply combining them with static analysis.

Answer to RQ1: TypeGen outperforms the best baseline Type4Py by 8.6% on all variables, with particularly notable improvements of 12.1% and 30.3% for argument and return value types, respectively, in terms of top-5 Exact Match.

C. *RQ2: Capability of* TypeGen *in Different Language Models*

We compare the performance of TypeGen on six language models with parameter sizes ranging from 1.3B to 175B with the Zero-Shot setting and the Standard ICL setting and present the overall top-1,3,5 Exact Match in Table III. In the Zero-Shot setting, our results indicate that language models with larger model sizes generally perform better, with ChatGPT achieving a 2× top-1 Exact Match than GPT-Neo-1.3B. When providing language models with three fixed examples in the Standard ICL setting, we observe an 8% $\sim$ 56% improvement in top-1 Exact Match, demonstrating the effectiveness of the *in-context learning* methodology. For TypeGen, we find consistent improvements of 27% $\sim$ 84% on different language models, with the improvements being more significant for smaller language models like GPT-Neo-1.3B than larger language models like ChatGPT. With less general knowledge stored in the models, smaller language models benefit more from the domain knowledge associated by TypeGen. Furthermore, the improvements achieved by TypeGen over the Zero-Shot setting are 2× $\sim$ 3× of that achieved by the Standard ICL setting, in terms of top-1 Exact Match. For the top-5 type prediction, TypeGen even achieves a 10× of improvement obtained by the Standard ICL setting on GPT-Neo-2.7B. These findings demonstrate the usefulness of incorporating static domain knowledge in the prompt design of TypeGen, which cannot be outweighed by simply providing some examples.

Answer to RQ2: TypeGen is capable of consistently improving the zero-shot performance of type inference for language models with different parameter sizes and achieves 2× $\sim$ 3× of improvements made by the Standard ICL setting.

D. *RQ3: Impacts of Different Parts of Prompt Design*

To investigate the impact of different parts of the prompt design of TypeGen, we conduct an ablation study and present the results in Table VII. The results show that removing code slicing techniques and inputting the whole function of target variables in the prompts leads to a significant performance drop of 24% on overall type inference. This decrease is mainly caused by local variables and arguments, as there is typically only a small set of statements in the function that have type dependencies with them, while inputting the entire function can introduce useless information and bias the language model. When type hints are removed, the performance of TypeGen on user-defined types decreases the most (11%), indicating the importance of providing available user-defined types as additional knowledge for language models. Additionally, when COT prompts are removed, the performance of TypeGen on generic types drops the most (10%), as generic types usually involve complicated type dependencies that should be well handled by static analysis. Providing the inference steps of static analysis in COT prompts can greatly help improve the performance of language models on generic types.

Answer to RQ3: In the prompt design, code slicing improves the overall performance of type inference by 24%, type hints improve the performance of user-defined type inference by 11%, and COT prompts improve the performance of generic type inference by 10%.

E. *RQ4: Impacts of Different Examples*

To evaluate the effects of the number of examples and example selection methods in the prompt design of TypeGen, we vary the number of examples from one to nine and compare two example selection methods: fixed examples and BM25 similarity-based examples. We present the top-5 Exact Match results of TypeGen in Fig. 6.

TABLE VII: The performance of TypeGen when removing different parts of the prompt design in TypeGen in terms of Top-5 Exact Match (%). “Ele”, “Gen”, and “Usr” indicate elementary types, generic types and user-defined types as well as third-party types, respectively. Other variable categories are the same with Table IV.

Ablation	Arg	Ret	Var	Ele	Gen	Usr	All
w/o Code Slice	74.8	77.0	68.8	75.1	75.5	73.9	70.8
w/o Type Hint	76.1	75.9	89.3	94.1	77.2	75.9	85.5
w/o COT Prompt	82.3	78.6	86.4	92.9	70.8	84.3	84.9
TypeGen	83.5	79.4	89.7	94.3	77.8	84.6	87.5

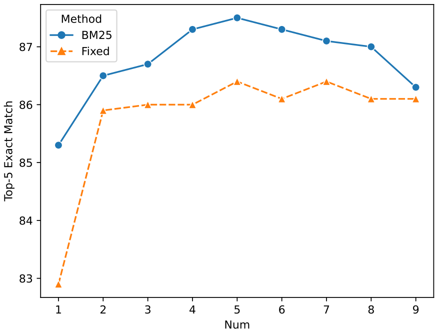
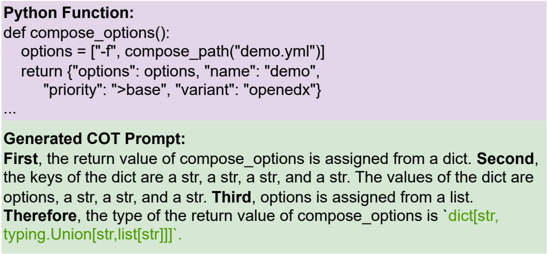


Fig. 6: The top-5 Exact Match of TypeGen with different numbers of examples and different example selection methods

Based on the results presented in Fig. 6, we observe that the performance of TypeGen is largely affected by the number of examples provided in the input prompts. Specifically, the performance drops notably when there is only one example, highlighting the importance of providing sufficient examples for effective *in-context learning*. Additionally, the performance of TypeGen using BM25 similarity-based examples increases up to five examples, after which it starts decreasing. This suggests that both inadequate and excessively long input contexts can harm the performance of TypeGen. When changing the example selection method, we find that using BM25 similarity-based examples performs better than using fixed examples, particularly when only one example is provided. However, when we provide five examples, the performance drop with fixed examples is relatively small (less than 1.3%). One possible explanation is that large language models learn how to perform type inference from the example prompts rather than solely relying on the direct correlations between type predictions in different example prompts [25].

Answer to RQ4: TypeGen achieves the best performance with five examples. TypeGen shows only a small performance drop (<1.3%) even when provided fixed examples, releasing the burden of developers to design examples.



Python Function:

```
def compose_options():
    options = ["-f", compose_path("demo.yml")]
    return {"options": options, "name": "demo",
        "priority": ">base", "variant": "openedx"}
...
```

Generated COT Prompt:

First, the return value of compose_options is assigned from a dict. **Second**, the keys of the dict are a str, a str, a str, and a str. The values of the dict are options, a str, a str, and a str. **Third**, options is assigned from a list. **Therefore**, the type of the return value of compose_options is `dict[str, typing.Union[str,list[str]]]`.

Fig. 7: A function whose return value type can only be inferred by TypeGen. The type annotation for the return value is Dict[str, Union[str, List[str]]].

VI. Discussion

A. *Interpretability of* TypeGen

To better illustrate the interpretability of COT prompts generated by TypeGen, we give an example of a function whose return value can only be inferred by TypeGen in Fig. 7. Due to the page limitation, we only present the code slice and the generated COT prompt. To infer the return type of function *compose_options*, TypeGen follows similar inference steps as static analysis. First, it infers that the return value is assigned from a dictionary in the generated COT prompt. Then it identifies the types of keys and values by specifying that there are four keys in the dictionary with types of *str*, and there are three values with types of *str* as well as one variable named *options*. The second step in the generated COT precisely matches the code given in the input prompt, indicating that large language models like ChatGPT have the capacity to simulate the inference steps of static analysis. In the third step of the generated COT prompt, TypeGen recognizes the unknown variable *options* and locates its assignment from a list. Since we set the maximum number of hops to 3, TypeGen generates the conclusion directly after the third step. From this example, we can find that by providing an explanation in the COT prompt, human developers can easily understand the predictions and determine whether the predictions are correct based on the explanations.

B. *Limitations of* TypeGen

Despite the effectiveness of TypeGen, we also identify the following limitations:

1) *Limited context.* Although TypeGen adopts static code slicing techniques based on TDGs, we have observed a limited number of instances in the test set (∼ 1000) with code slices exceeding the maximum context length of language models. This primarily occurs in extremely long functions with complex type dependencies. We recognize that extracting code slices without sacrificing key dependency information is still a challenge.

2) *Limited knowledge for function arguments.* As function arguments lack precise definitions, TypeGen provides naming and usage information to enable language models to predict their types. However, this information is incomplete and can potentially introduce biases in the model's predictions [11]. In-

corporating data flow information via inter-procedural analysis may be a possible solution to enhance argument information.

VII. Conclusion

This paper presents TypeGen, a few-shot generative type inference method for Python programs. Our approach incorporates static domain knowledge into language models via a novel prompt design in the *in-context* learning paradigm. Experimental results show that TypeGen outperforms both state-of-the-art supervised type inference methods and cloze-style type inference methods.

VIII. Acknowledgement

The authors would like to thank the efforts made by anonymous reviewers. The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). The work was also supported by National Natural Science Foundation of China under project (No. 62002084), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), and Key Program of Fundamental Research from Shenzhen Science and Technology Innovation Commission (Project No. JCYJ20200109113403826). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the above sponsoring entities.

IX. Data Availability

You can find the code and data related to this paper at https://github.com/JohnnyPeng18/TypeGen.

References

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, page 428–452, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Sid Black, Stella Biderman, Leo Gao, Phil Wang, and Connor Leahy. Gpt-neo, 2022. https://github.com/EleutherAI/gpt-neo.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[5] Sheng Chen and Martin Erwig. Principal type inference for gadts. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 416–428. ACM, 2016.

[6] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. A survey on in-context learning, 2023.

[7] Michael Emmi and Constantin Enea. Symbolic abstract data type inference. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 513–525. ACM, 2016.

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

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[10] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1859–1866, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1933–1945, 2023.

[12] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R. Lyu. Constructing effective in-context demonstration for code intelligence tasks: An empirical study. *CoRR*, abs/2304.07575, 2023.

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics, 2022.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[15] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery.

[16] HuggingFace. Huggingface hub, 2023. https://huggingface.co/.

[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, page 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. Learning type annotation: Is big data enough? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1483–1486, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Jukka Lehtosalo. PEP 589 – TypedDict: Type hints for dictionaries with a fixed set of keys, March 2019. https://www.python.org/dev/peps/pep-0589/.

[20] Ivan Levkivskyi, Jukka Lehtosalo, and Łukasz Langa. PEP 544 – protocols: Structural subtyping (static duck typing), March 2017. https://www.python.org/dev/peps/pep-0544/.

[21] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 155–166, 2021.

[22] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. Towards enhancing in-context learning for code generation. *CoRR*, abs/2303.17780, 2023.

[23] libraries.io. libraries.io, 2023. https://libraries.io/.

[24] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In *Workshop on Knowledge Extraction and Integration for Deep Learning Architectures; Deep Learning Inside Out*, 2021.

[25] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 11048–11064. Association for Computational Linguistics, 2022.

[26] Amir M. Mir, Evaldas Latoskinas, and Georgios Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 585–589. IEEE, 2021.

[27] Amir M Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py, 2022. https://github.com/saltudelft/type4py.

[28] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2241–2252. ACM, 2022.

[29] Mypy. https://github.com/python/mypy/.

[30] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[31] Wonseok Oh and Hakjoo Oh. Pyter: effective program repair for python type errors. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 922–934. ACM, 2022.

[32] OpenAI. Chatgpt, 2022. https://openai.com/blog/chatgpt.

[33] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. Data flow refinement type inference. *Proc. ACM Program. Lang.*, 5(POPL):1–31, 2021.

[34] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Hityper, 2022. https://github.com/JohnnyPeng18/HiTyper.

[35] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R. Lyu. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors. *CoRR*, abs/2306.01394, 2023.

[37] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 209–220, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Pyre check. https://pyre-check.org/.

[39] Pyright. https://github.com/microsoft/pyright.

[40] Pytype. https://github.com/google/pytype.

[41] Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, 2009.

[42] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[43] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484 – Type Hints, 2014. https://www.python.org/dev/peps/pep-0484/.

[44] Ben Wang and Aran Komatsuzaki. Gpt-j, 2022. https://github.com/kingoflolz/mesh-transformer-jax.

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022.

[47] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. *CoRR*, abs/2005.02161, 2020.

[48] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[49] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models. *CoRR*, abs/2210.14179, 2022.

[50] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *CoRR*, abs/2301.13246, 2023.

[51] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. Knowledge-based environment dependency inference for python programs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1245–1256. ACM, 2022.

[52] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[53] Łukasz Langa. PEP 589 – type hinting generics in standard collections, March 2019. https://www.python.org/dev/peps/pep-0585/.