HiTyper: A Hybrid Static Type Inference Framework with Neural Prediction

Yun Peng
The Chinese University of Hong Kong
Hong Kong, China
ypeng@cse.cuhk.edu.hk

Zongjie Li
Harbin Institute of Technology
Shenzhen, China
lizongjie@stu.hit.edu.cn

Cuiyun Gao∗
Harbin Institute of Technology
Shenzhen, China
gao cuiyun@hit.edu.cn

Bowei Gao
Harbin Institute of Technology
Shenzhen, China
1160300103@hit.edu.cn

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Michael Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

ABSTRACT
Type inference for dynamic programming languages is an important yet challenging task. By leveraging the natural language information of existing human annotations, deep neural networks outperform other traditional techniques and become the state-of-the-art (SOTA) in this task. However, they are facing some new challenges, such as fixed type set, type drift, type correctness, and composite type prediction.

To mitigate the challenges, in this paper, we propose a hybrid type inference framework named HiTyper, which integrates static type inference into deep learning (DL) models for more accurate type prediction. Specifically, HiTyper creates a new syntax graph for each program, called type graph, illustrating the type flow among all variables in the program. Based on the type graph, HiTyper statically infers the types of the variables with appropriate static constraints. HiTyper then adopts a SOTA DL model to predict the types of other variables that cannot be inferred statically, during which process a type correction algorithm is employed to validate and correct the types recommended by the DL model. Extensive experiments show that HiTyper outperforms the SOTA DL approach by 12.7% in terms of top-1 F1-score. Moreover, HiTyper filters out 50.6% of incorrect candidate types recommended by the SOTA DL model, indicating that HiTyper could improve the correctness of predicted types. Case studies also demonstrate the capability of HiTyper in alleviating the fixed type set issue, and in handling type drift and complicated types such as composite data types.

1 INTRODUCTION
Dynamically typed programming languages are becoming increasingly prevalent in recent years. According to GitHub Octoverse 2019 and 2020 [12], JavaScript and Python outperform Java and C/C++ and become the top two most popular programming languages. Their powerful dynamic features provide more flexible coding styles and enable fast prototyping at the expense of omitting type information. Without concretely defined variable types, these programming languages are faced with challenges on ensuring security and performance. To address such problems, researchers try to borrow designs of statically typed programming languages [13, 18, 35], such as reusing compiler backend of the statically typed languages [19], predicting types for most variables [1, 4, 9, 14, 16, 17, 29], etc. Python also supports type annotations in the Python Enhancement Proposals (PEP) [21, 22, 39, 43].

Type prediction is a popular task performed by most attempts. Traditional static type inference techniques [4, 9, 14, 17, 36] and type inference tools such as Pytype [34], Pysonar2 [33], and Pyre Infer [31] can predict sound results for the variables with enough static constraints, e.g., a = 1, but are unable to handle the variables with few static constraints, e.g. most function arguments. On the other hand, dynamic type inference techniques [3, 37] and type checkers simulate the workflow of functions and solve types according to input cases and typing rules. They achieve better precision than static inference techniques but suffer from limited code coverage and much time consumption. This hinders them from being deployed on large code bases. Therefore, both static analysis and dynamic analysis are limited on this task. Fortunately, with the recent development of natural language processing (NLP) methods, we can leverage more type hints such as identifiers and existing type annotations to help predict types. Many NLP-based methods, we can leverage more type hints such as identifiers and existing type annotations to help predict types. Many NLP-based methods [1, 16, 24, 29, 42] using machine learning and deep neural networks are proposed, and they show great improvement compared with static and dynamic techniques [20].

Challenges. While the NLP based approaches are effective, they still face some challenges:

1) Fixed type set: The set of all candidate types are generally fixed in most models during prediction, so the types not in the type set are hard to be accurately predicted. Built-in types in Python such as int, float can be easily included and predicted by these approaches but user-defined types, which are not pre-defined in Python’s standard libraries and usually correspond to classes defined by developers, vary across projects. Therefore, a fixed type set cannot include all possible types, especially user-defined types. Typilus [1] tries to ameliorate the problem by copying existing types from the training set, but the problem would still exist when applying it to new projects with types not appearing in the training set. Retraining the model to address this cold start issue requires developers to manually add some annotations in the new projects, limiting the generalizability of the type prediction model.

2) Type drift: Hellendoorn et al. [16] find that the type predicted by sequence models of one variable may change throughout the definition and use of the variable in source code, even if the true

∗Corresponding author

1https://docs.python.org/3/library/stdtypes.html
type actually does not change. To avoid this problem, most neural networks assume that the variable types are constant in one program, and focus on predicting the type of each variable just once. However, dynamic type systems allow the types of the same variables to change at runtime, thus the assumption could impact the accuracy of the predicted types.

3) **Type correctness**: Compared with static inference, probabilistic type predictions are inherently imprecise as they suggest one or more candidate types for each type slot but do not guarantee their correctness since only one type is correct in the end. How to choose the only one correct type from all candidate types is a problem. TypeWriter [29] addresses this problem by using type checkers to validate its predictions, but such a passive validation method consumes much time when validating variable-level predictions as the search space grows fast. What’s more, this method also fails if the correct type of one variable is not in the predicted list but can be inferred from the predicted list of other variables. We will better illustrate this with the motivating example shown in Listing 1 in Sec. 2.

4) **Composite data type**: Composite data types refer to any data types which can be constructed using primitive data types and other composite types, such as list, tuple, etc. An example of composite data types is `List[Tuple[int, float, str]]`. Different from user-defined types, composite types are constructed by several existing built-in types. However, the number of combinations of existing built-in types is almost infinite, making it impossible to build a fixed type set. Besides, composite types may not appear in the training set, so NLP-based methods can hardly make accurate predictions for them.

**Solution.** To address these problems, we propose HiTyper, a hybrid type inference framework, utilizing both static inference and neural network recommendations to better predict types. HiTyper infers types for all variables in source code (i.e., at variable level) corresponding with the prior work [1] instead of only function arguments and return values (i.e., at function level), because more and more tasks such as code representation [2, 40], API recommendations [5, 7, 15, 23, 41] and bug detection [11, 28, 30] can benefit from using the inferred variable types.

Given a function, HiTyper first creates the corresponding type graph. A type graph is similar to a data flow diagram but it represents the type flow instead of data flow among all the variables in the function. Based on the type graph, HiTyper statically infers the types of the variable slots with static constraints. For the slots that cannot be inferred by static analysis, HiTyper then adopts a deep learning (DL) model for further prediction. To ensure the correctness of the predicted types, we design type rejection rules and a user-defined type correction algorithm to validate and correct the candidate types recommended by the DL model. This process may iterate until all the type slots are successfully inferred. Finally, HiTyper outputs type assignments for all the variables.

HiTyper mitigates the aforementioned challenges through the following design:

1) For the fixed type set: HiTyper dynamically builds the type set for each source code file and adopts a type correction algorithm to supplement the user-defined types that do not appear in the training set of DNNs. 2) For the type drift: HiTyper creates a type graph for each function where occurrences of the same variables are connected if no writing operations are detected. In this way, the types of different occurrences could be accurately predicted.

3) For the type correctness: HiTyper validates the recommended candidate types by the DL model based on standard typing rules and designed type rejection rules. 4) For the composite data type: HiTyper divides the prediction of a whole composite type into the prediction of its element types since DL models generally present good performance in predicting simpler and common types. HiTyper then constructs the composite data type based on all the predicted element types.

**Results.** We evaluate HiTyper on the dataset released by Alamans et al. [1]. We show that HiTyper achieves an overall top-1 F1 score of 0.62 on all types, 0.47 on user-defined types, and 0.60 on return values, outperforming SOTA deep neural network Typilus by 12.7%, 38.3%, and 39.5% respectively. Besides, HiTyper successfully infers 11.3% of type slots in the test set which cannot be inferred by Typilus. HiTyper still outperforms Typilus on top-5 F1 score by 9.2%, which means that HiTyper could correctly infer some types not existing in the top-5 ranked list returned by Typilus. These types cannot be easily corrected using the type checkers mentioned in [29]. A comparison on the average number of candidate types added into each type slot also demonstrates the effectiveness of the designed type rejection rules in HiTyper.

**Contributions.** Our contributions can be concluded as follows:

- To the best of our knowledge, we propose the first framework which combines static inference with deep neural networks for more accurate type prediction.
- We design an innovative type graph to provide type templates with strict static constraints for improving type correctness.
- We mitigate the fixed type set issue by dynamically building the type set and implementing a type correction algorithm to supplement user-defined types in predictions of neural networks.
- Extensive experiments demonstrate the superior effectiveness of the proposed HiTyper than SOTA baseline models in the task.

## 2 MOTIVATION

In this section we elaborate on the motivation of our work. We first analyze the type distribution of a benchmark dataset to illustrate...
the challenges of the type prediction task, and then provide an example to demonstrate how HiTyper mitigates the challenges.

**Type Distribution.** We analyze the type distribution of the benchmark dataset released by Allamanis et al. [1]. It contains 12,071 Python source files from 592 popular Python projects on GitHub. We collect 153,731 annotations which are associated with 12,138 different types. Figure 1 shows the distribution of top 11 annotated types according to the occurrence frequency among all 12,138 types. In this figure, we can see that the number of the type annotations drops dramatically from the most popular type to the type Optional[str]. Some basic built-in types like str, int, bool, and float are quite commonly used and account for about 55% of annotations. However, the remaining 12,127 types are rarely used since each of them accounts for less than 0.5% of the annotations. This indicates a Long Tail phenomenon: quite a few basic built-in types are commonly used but a substantial number of other types, which account for ~50% of the total annotations, are rarely utilized. Most of these types are user-defined types and composite types, which only occur in a few source files and can vary across projects, making it hard to include them in a fixed type set and for DL models to learn the features. For example, Typilus’s precision drops by 50% when predicting types occurring less than 100 times than types occurring more than 10,000 times in the training set [1]. Therefore, it is challenging but yet critical to handle user-defined types and composite types in order to build a sound and comprehensive approach for Python type inference.

**Observation 1:** Although some basic built-in types are commonly used and account for more than half of human annotations, the amount of user-defined types and composite types are substantial. It is crucial to handle them well if we want to build a more accurate type inference tool.

**Motivating Example.** To address the limitations of current approaches, we propose HiTyper to combine: 1) static inference to utilize the constraints between variables and to eliminate the predictions which contradict with standard typing rules; 2) deep neural networks to leverage comments and variable names for predicting types. We take Listing 1 from the WebDNN project as a motivating example to further illustrate our framework. The types of the argument, local variable pt and return value of the parse() function inferred by the baseline approaches and HiTyper are listed in Table 1.

```python
# src/graph_transpiler/webdnn/graph/shape.py
def parse(text):
    normalized_text = _normalize_text(text)
    tmp = ast.literal_eval(normalized_text)
    shape = []
    placeholders = {} 
    for i, t in enumerate(tmp):
        if isinstance(t, str):
            pt = Placeholder(label=t)
            placeholders[t] = pt 
        elif isinstance(t, int):
            pt = t
        shape.append(pt)

return shape, placeholders
```

Listing 1: A Function from WebDNN

As can be seen in Table 1, static inference tools such as Pysonar2 and Pytype fail to infer the type of the argument text. This is because arguments are generally at the beginning of the data flow without many static constraints. For the deep neural networks such as Typilus, they leverage the contextual information including the naming and similar cases in the training set for type prediction, thus they can accurately infer the argument type as str. Regarding the return value type, as shown in Table 1, we can observe that the DNN and static inference tools all predict wrong results. Although not 100% matching the ground truth, Pysonar2 and Pytype can at least infer the return value of parse() as a tuple containing a list and a dict. For Typilus, it can only predict the return value as a tuple but fails to produce the types inside the tuple. We can also notice that simply adding a type checker is not beneficial for the final result of Typilus since all the top 5 predictions are incorrect. The local variable pt shows a special case where its type changes from line 9 to line 12 in Listing 1. Pysonar2 identifies this change so it outputs two types for pt even if one type is unknown. However, Typilus combines all occurrences of pt in parse() so all its top-5 predictions contain only one type for pt. Therefore, combining all occurrences of one variable to avoid type drift hurts the performance of DL models.

**Observation 2:** Deep neural networks would perform well on predicting argument types, but poorly for the complex data types with strict constraints such as composite data types.
Based on the Observation 2, we propose HiTyper to integrate static type inference techniques with DL models for more accurate type prediction. HiTyper first creates the type graph of the function `parse()`, as shown in Figure 2, which indicates the type flow among variables. Based on the type graph, HiTyper statically infers the types of the variables with appropriate static constraints. For example, `shape = []` indicates that variable shape at line 5 in Code 1 is a list. Such type slots can be determined by static analysis without relying on neural networks, but other type slots such as the argument `text` still remain unsolved. The unsolved ones are referred to blank type slots in this paper. HiTyper collects all blank type slots and removes those that can be inferred from other blank type slots. For example, in Figure 2 node `text2(3)` is removed as it can be directly inferred from `text1: arg(2)`. The rest of the slots are named as hot type slots. HiTyper then employs a SOTA DL model, e.g., Typilus, to predict the types, and validates and corrects the recommended types by Typilus. For example, here Typilus recommends `str` as the type of `text`. HiTyper validates and accepts the result since it is a valid built-in type and does not violate any typing rules. HiTyper repeats the process until all the type slots are inferred.

3 PROPOSED APPROACH

HiTyper accepts Python source files as input and output JSON files storing the type assignments. Its overall architecture is shown in Figure 3, including three major components: type graph generation, static type inference and rejection, and type recommendation.

Given a Python source file, HiTyper first generates type graphs for each function and extracts all imported user-defined types. Then it conducts static type inference by walking through the type graph and by activating the typing rules stored in operation nodes. Some type slots are inferred at this step (black solid nodes in the type graphs in Fig. 3); however, most of them without appropriate static constraints remain unsolved, i.e., blank type slots. HiTyper identifies a key subset of them as hot type slots (red nodes in Fig.3) and then adopts a SOTA DL model to recommend candidate types for the hot type slots. To supplement user-defined types which are not in the DL model’s type set, HiTyper employs a user-defined type correction algorithm to find the most similar valid user-defined type and replace the one predicted by the DL model, thus indirectly enlarging the type set. The process could iterate until all the type slots in the type graph are completed with type inference.

In the following subsections, we will introduce the details of three main parts in HiTyper: type graph generation, static type inference and rejection, and type recommendation.

3.1 Type Graph Generation

The first part of HiTyper is the type graph generator. We define a graph $G = (N, E)$ where $N = \{n_i\}$ is a set of nodes representing all variables and operations in source code, and $E$ is a set of directed edges of $n_i \rightarrow n_j$ indicating the type of $n_j$ comes from the type of $n_i$ as a type graph. We also denote $n_i$ as the input node of $n_j$ and $n_j$ is the output node of $n_i$ here. A type graph is similar to a data flow diagram, but it only simulates the data flow, i.e., how does a type transfer from one variable to another one, instead of the specific data flow. An example of the type graph is shown in Fig. 2. HiTyper generates the type graph for each function in source code. Based on the type graphs HiTyper can conduct static type inference and select hot type slots for the DL model to recommend.

A type graph contains 4 kinds of nodes: Symbol nodes, operation nodes, branch nodes, and merge nodes.

**Symbol nodes.** A symbol node in a type graph represents a variable occurrence in source code. Like static single assignment (SSA), HiTyper labels each occurrence of a variable with the order of occurrences so that each symbol node in Fig. 2 has a format of `@name$order($lineno)` to uniquely indicate a variable occurrence. For example, for `t` in Listing. 1, there are two nodes “t1 (8)” and “t2 (9)” at the top of Figure 2 representing their occurrences in line 8 and 9, respectively. HiTyper does not combine all occurrences of one variable into one symbol node, since Python allows a variable to change its type at runtime. HiTyper infers types for every symbol node so each symbol node is also a type slot.

**Operation nodes.** We define operations here to indicate behaviors that may cause type changes. When an operation such as `a + b` occurs in source code, the type graph generator will add an operation node. Each operation node stores corresponding typing rules to infer the types of the result given types of the input nodes. A special operation node is `call` nodes (highlighted in red in Fig.2) which represent the function calls. Static inference can hardly handle function calls because many callees are not shown in the current source file or even not implemented in Python. Function calls are also the main source of user-defined types as developers often use `__init__` functions in classes to create a user-defined type such as the function call `Placeholder()` at line 9 of Listing 1.

**Branch and merge nodes.** These two kinds of nodes are introduced to handle possible branches in if statements, loop statements, and exception handling statements in source code. Branch nodes have only one input node and multiple output nodes. They split the types of the input node and then assign them to the output nodes. On the contrary, merge nodes have multiple input nodes but only one output node, they collect all types from the input nodes and assign them to the output node.

**Type graph generation.** Given a Python source file, HiTyper first transforms it into the standard abstract syntax tree (AST) using Python standard library `ast`. Then, it generates the type graphs by walking through the AST. HiTyper also maintains an operation stack and a variable stack to help generate the type graphs. When HiTyper visits an operation such as `Assign` in AST, it firstly pushes this operation to the operation stack and checks its operands. The operands can be operations again or just variables. For the operation’s operands, HiTyper continues to push them into the operation stack until it visits variables. For variable operands, it creates symbol nodes for them and checks the contexts (being read or written). If a variable is written, its type must come from an operation node because operations always indicate type changes, thus HiTyper sets the input node of this variable as the top operation in the operation stack. If this variable is read, its type directly comes from the last
3.2 Static Type Inference and Rejection

Given the type graph for each function, when an operation node is visited HiTyper examines all its input nodes. If the types of all input nodes are fully inferred, HiTyper infers the types of the operation result according to pre-defined typing rules and rejection rules. We divide all operations into three categories: 1) Unary operations with only one operand, 2) N-ary operations with multiple operands, 3) function calls, which are different from regular operations as their typing rules are defined by developers instead of Python standard specifications. Also, they can have one or multiple operands (arguments). Table 2 lists the input, output, example, typing rule, and rejection rule of these operations. Typing rules define the behavior of how an operation generates the output type given types of operands. Rejection rules define the behavior of how an operation rejects invalid types before conducting typing rules.

Table 1: Type prediction results of different techniques on parse() from Code. 1 (Pytype does not infer local variable types)

| Table 1: Type prediction results of different techniques on parse() from Code. 1 (Pytype does not infer local variable types) |
|---|---|---|
| Argument: text | Local Variable: pt | Return Value |
| Ground Truth: | str | Union[int, Placeholder] | Tuple[List[Union[int, Placeholder]], Dict[str, Placeholder]] |
| Pysonar2: | ? | Union[int, ?] | Tuple[List[int], Dict] |
| Pytype: | ? | - | Tuple[List] |
| Typilus: (Top-5) | 1. str | 1. torch.Tensor | Tuple[collections.OrderedDict[Text, List[DEAState]], Optional[Text]] |
| | 2. cate.core.types.DatasetLike | 2. Tuple[Text, Text, Text] | Tuple[Any, List[Tuple[Any], Any]] |
| | 3. Optional[dependencies.] | 3. Tuple[torch.Tensor] | Tuple[Text, Text] |
| | 4. Optional[datetime.datetime] | 4. Tuple[Any, list] | |
| | 5. Optional[int] | 5. Tuple[list, CommandOptionValues] | |
| HiTyper: | str | Union[int, Placeholder] | Tuple[List[Union[int, Placeholder]], Dict[str, Placeholder]] |

Figure 3: Overall architecture of HiTyper. Black solid nodes, hollow nodes, red nodes and yellow nodes in the type graphs represent inferred type slots, blank type slots, hot type slots, and the type slots recommended by DL model, respectively.

 occurrence of this variable in the same scope, so its input node is set as the top symbol node in the variable stack. Such connections avoid the type drift problem because the type of a variable will not change between its definition and its usage unless it is redefined again. HiTyper pushes a symbol node into the variable stack after it is created. After all operands of an operation are visited, HiTyper pops the operation from the operation stack and creates an operation node whose input nodes are its operands. When HiTyper visits a condition in branch statements indicating the true branch and the false branch have different types, HiTyper creates a branch node for the variables in the condition. For example, HiTyper creates a branch node for a in if isinstance(a, int) which directly indicates that a in the true branch should be an integer while a in the false branch should not have the type int. However, different from data flow diagram, HiTyper does not create a branch node for a in if a > 10 since both the true branch and the false branch can have type int. Merge nodes are created whenever more than one type flows merge, e.g., type flow of the true branch and the false branch merges after the if statement.

Table 2: Three kinds of operations

| Unary Op | N-ary Op | Function Call |
|---|---|---|
| Input | x | x1,x2... | arg1, arg2... |
| Output | z | z | z |
| Example | x = not x | z = x1 + x2 | z = func(arg1, arg2) |
| Typing Rule | \( T_z \leftarrow \phi(T_x, R) \) | \( T_z \leftarrow \phi(T_x, R_{\mathcal{E}}) \) | \( T_z \leftarrow \phi(F, U, T) \) |
| Rejection Rule | \( R_z \leftarrow R_x \) | \( R_z \leftarrow R_x \) | \( R_z \leftarrow \emptyset \) |

**Unary Operation.** Unary operation is the most simple operation in source code, as HiTyper does not need to consider the relationship between its operands. Suppose \( \phi_x(\cdot) \) represents the typing rule of operation \( o \) converting the types of operands into result types, and \( T_x \) represents the types of operands while \( T_z \) represents
the types of result. Then the typing rule of operation \( o \) can be written as \( T_x \leftarrow f(T_x \setminus RT_x) \) where \( RT_x \) represents the rejected input types. Suppose \( R_x \) represents the invalid input types of operation \( o \), e.g., \( List \) is a invalid type for operation \( int() \) as a list cannot be transferred to an integer. Because unary operations have only one operand, their rejection rules only consider the type of the operand is valid or not, which can be simply written as \( RT_x \leftarrow R_x \).

**N-ary Operation.** N-ary operation has more than one operand, thus not only types of each operand but also their relationship should be considered. For example, \( str \) and \( int \) both can be valid input types for operation \( + \) such as "a" + "b" and 1 + 2. However, "a" + 1 is illegal since \( str \) and \( int \) cannot be directly added to each other. Therefore, we introduce a new notion \( \times \) to indicate the relationship between multiple operands. \( \times \) can be \( \infty \), which requires that two operands have the same types, or \( \subseteq \) which requires that the type set of one operand be included in the type set of the other operand. Suppose \( RL_x \) represents all valid relationship of operands in operation \( o \), which are extracted from Python specification. Typing rule of N-ary operations can be written as \( T_x \leftarrow \phi_o(T_x \setminus RTx_i, ..., T_x \setminus RTx_n) \). Rejected types of operands are excluded before applying typing rules. Rejection rule can be written as \( RT_{x_i} \leftarrow RL_o \cap \{ t | \forall t_{\in} T_{x_i} \times \times T_{x_i} \setminus \{ t \} \times \times T_{x_n} \in RL_o, \forall t_{\times} \times \times T_{x_i} \times \times T_{x_n} \notin RL_o \} \). Apart from invalid type set \( R_o \), rejection rules also reject types resulting in invalid operand relationship. For input type sets resulting in invalid relationship, HiTyper gradually removes one type from the input type set and checks if the new input type set fulfills the requirement of valid relationship. If so, the removed type will be inserted into the rejected input type set \( RT_x \).

**Function Call.** Function calls do not have pre-defined typing rules as developers are free to customize their behaviors in the function body. Actually static analysis can hardly handle function calls as developers often use lots of APIs from standard or third-party modules. To mitigate this problem, HiTyper collects typing rules of some standard APIs such as \( len() \), and supports class instance definition such as \( a = \text{Placeholder}() \) as this is one of the major source of user-defined types. Suppose the built-in function set HiTyper collects is \( F \), and the user-defined type set collected by user-defined type extractor is \( U \). The typing rule of function calls can be written as \( T_x \leftarrow \phi_{F}(F, U, sig) \) where \( sig \) is the signature of the current function. HiTyper does not reject types when handling function calls since the pre-conditions of function calls are not strictly fixed even in built-in functions since developers may redefine a function with the same name with built-in functions.

After the entire type graph is visited, HiTyper checks if there are blank type slots. If not, the entire function is fully inferred and HiTyper generates the final type assignments. If there are still blank type slots, then HiTyper enters the next phase: type recommendation, in which HiTyper asks for recommendations from deep neural networks to infer the remaining type slots.

### 3.3 Type Recommendation

Data-driven methods such as deep neural networks have shown superior performance on type prediction than traditional static inference techniques. However, as discussed in Section 1, they are facing some challenges that can be well handled by static analysis. In our approach, we integrate the DL model as part of our framework by combining it with static inference to handle these challenges. Type recommendation phase is where HiTyper interacts with the DL model.

**Hot type slot identification.** Type correctness and the type drift problem hinder DL models from predicting more accurate types. Directly using DL models to fill all blank type slots can introduce many wrong types. Instead, HiTyper selects a key subset of blank type slots for DL models to predict, and this key subset is called hot type slot set (red nodes in Fig. 3). Given a set of blank type slots, HiTyper removes a type slot if all of its input nodes are also in the set, which means this slot can be inferred from other blank slots. This process significantly reduces the number of type slots accepting recommendations from deep neural networks, especially different occurrences of the same variables. By identifying hot type slots, HiTyper accepts recommendations of a few occurrences of the variable and infers others according to static typing rules. Consequently, it avoids type drift and improves the correctness of the predicted types.

**Algorithm 1 User-defined type correction algorithm of HiTyper**

**Input:** Variable name, name;
- Valid user defined type set, \( S \);
- Type String recommended by deep neural networks, \( t \);
- Penalty added for name-type similarity to align with type-type similarity, penalty;

**Output:** Corrected type of current variable, \( ct \);

1. if \( t \in S \) or \( isbuiltin(t) \) then
2. \( ct \leftarrow t \);
3. else
4. \( largest_sim \leftarrow 0 \); \( largest_type \leftarrow \text{None} \);
5. \( tw \leftarrow \text{BPE}(t); \ namew \leftarrow \text{BPE}(name); \)
6. for each \( pt \in S \) do
7. \( ptw \leftarrow \text{BPE}(pt); \)
8. if \( sim(ptw, tw) > largest_sim \) then
9. \( largest_sim \leftarrow sim(ptw, tw); \ largest_type \leftarrow pt; \)
10. end if
11. if \( sim(ptw, namew) + penalty > largest_sim \) then
12. \( largest_sim \leftarrow sim(ptw, namew); \ largest_type \leftarrow pt; \)
13. end if
14. end for
15. \( ct \leftarrow largest_type \);
16. end if

**User-defined type correction.** After identifying all hot type slots, HiTyper accepts recommendations from the top one or more types in the prediction list of the DL model (yellow nodes in Fig.3), depending on the strategy (Top-1, Top-3, or Top-5) HiTyper uses. As the fixed type set issue hinders DL models from precisely predicting user-defined types, HiTyper validates and corrects the recommended user-defined types by conducting a type correction algorithm shown in Algorithm 1. HiTyper only examines and corrects user-defined types at this phase since invalid built-in types are directly rejected by the rejection rules in operation nodes. If the recommended type is not a built-in type, HiTyper checks if
it appears in the user-defined type set (line 1). If not, this type is not valid and should be corrected. For such invalid types, HiTyper replaces them with the most similar candidate in the user-defined type set collected by a user-defined type extractor. Given two types, HiTyper uses Word2Vec [25] to embed them into two vectors and calculates their cosine similarity (lines 8 - 10). As Out-of-vocabulary (OOV) issue may occur in these types, HiTyper first splits them into subwords using BPE algorithm [10, 38] (line 5). BPE starts with characters in a string and merges characters into a new vocabulary if it has a high occurrence frequency, and finally HiTyper splits the original string into several subwords according to the new vocabularies it created. Thus after training on the corpus produced by BPE, our Word2Vec model rarely encounters OOV tokens when calculating the similarity between two types. In most cases DL models can find a quite similar user-defined type in the training set even if the ground truth type does not exist, but it is still possible that there is no user-defined type in the training set, which is similar to the correct one. To mitigate this issue, HiTyper also considers variable names in the algorithm to help correct user-defined types. For example, variable name placeholders in Listing 1 is quite similar to its value type Placeholder. Therefore, HiTyper also considers the similarity between the variable name and all valid user-defined types (lines 11 -13), but it adds a penalty to prevent the DL models’ recommendations from being easily overwritten, as DL models often consider much more information, not just the variable name. Finally HiTyper chooses the type candidate with the highest similarity to fill the hot type slot (line 15).

After all hot type slots are recommended, HiTyper goes back to the static type inference and rejection phase to 1) infer the remaining blank type slots; and 2) reject wrong built-in type predictions recommended by the DL models.

4 IMPLEMENTATION DETAILS

4.1 Dataset

We used the Python dataset released by Allamanis et al. [1] for evaluation. Based on the published scripts, we collect 12,071 Python source files of 592 Python projects which contain at least one type annotation from GitHub. Following the processing practice described by Allamanis et al. [1], we finally obtain 153,731 annotations which belong to 12,138 different types. We split the dataset by 7:1:2 to create the training set, validation set, and test set, respectively. Table 3 lists the information of the test set used for the evaluation in Section 5. We follow the guidance given by Allamanis et al. [1] and enlarge the training set and validation set using Pytype [34] to improve the performance of Typilus. However, we use the original test set which contains only human annotations for fair evaluation since we also compare with static inference tools.

### Table 3: Type distribution in the test set.

| Category | Total | Composite | User | Arg | Return |
|----------|-------|-----------|------|-----|--------|
| Count    | 20,456| 3,286     | 7,431| 14,367| 6,089  |
| Proportion | 100% | 16%       | 36%  | 70%  | 30%    |

Besides, for evaluating type correctness, we also compare the number of annotations added by different approaches. We calculate the average number of candidate types added for one type slot as $\frac{\sum n_i}{n_{slot}}$ where $n_i$ is the number of candidate types in one slot and $n_{slot}$ is the total number of type slots.

4.3 Baseline Approaches

To verify the effectiveness of the proposed HiTyper, we choose three baseline approaches for comparison:

1) A naive baseline. It represents a basic data-driven method. We build this baseline following TyperWriter [29], which considers the ten most frequent types in the dataset and makes predictions by sampling from the distribution of the ten types. This method is a simple data-driven method that does not consider code semantics or structure information.

2) Pysonar2 [33]. It is a static type inference tool for Python. We adopt it as our baseline as it also focuses on variable-level type inference and is a popular baseline for the task. Note that we do not adopt Pytype as our baseline since Typilus uses it to enlarge its training set.

3) Typilus [1]. It is the SOTA DL model which utilizes code structural information by converting source code into a graph with different kinds of edges. It is also the first DL model which handles the fixed type set problem by copying user-defined type from the training set. We use the Docker file provided by Typilus and strictly follow all the procedures mentioned in its documentation4 to train and validate it. We choose the model with the best performance on the validation set for comparison.

4.4 Implementation of HiTyper

The entire framework of HiTyper is implemented using Python, which contains more than 9,000 lines of code. We obtain all typing rules and rejection rules from Python’s official documentation [8] and its implementation CPython5. We use Word2Vec model from the gensim library [44] as the embedding when calculating the similarity between two types. We train the Word2Vec model by utilizing all the class names and variable names in the training set of Typilus. The dimension of the word embeddings and size of the context window are set as 256 and 10, respectively. Due to the small training corpus for Word2Vec, we choose Skip-Gram algorithm for model training [26]. We choose Typilus as the neural network model from which HiTyper accepts type recommendations.

5 EVALUATION

In this section we aim to answer the following research questions:

RQ1: How effective is HiTyper compared to baseline approaches?

---

4 https://github.com/typilus/typilus
5 https://github.com/python/cpython
RQ2: How well does HiTyper handle the challenges elaborated in Section 1?
RQ3: What is the impact of different components of HiTyper on the model performance?

5.1 RQ 1: Effectiveness of HiTyper compared to baseline approaches

We evaluate HiTyper and other baseline approaches considering different type categories, including arguments and return values, user-defined types and composite data types, and all the types. Table 4 shows the evaluation results. Since Pysonar2 only returns one exact type for each type slot, the top-3 and top-5 results cannot be calculated. The results of HiTyper are computed by accepting the top-1, 3, 5 candidate types generated by the DL model in the type recommendation phase.

All the types. As shown in the Table 4, HiTyper’s recall and F1 score outperforms all baseline approaches on all types. Pysonar2 outperforms HiTyper by 0.02 on top-1 precision, which is reasonable as purely static inference methods infer a type only with higher confidence. As a cost of high confidence in precision, Pysonar2’s recall and F1 score are much worse than other approaches, even the naive baseline, which shows data-driven methods can correctly infer much more variable types. For the top-1 results, HiTyper outperforms the SOTA DL-based approach Typilus by 12.7% on F1 score, which demonstrates the effectiveness of HiTyper. Besides, for top-5 results HiTyper still outperforms Typilus on F1 score by 9.2%, which indicates that HiTyper can infer types which are not recommended by Typilus. Such cases cannot be simply corrected using type checkers because type checkers only filter out those contradictory to typing rules without inferring new types. Therefore, our approach can better improve the performance of the DL model than simply adding a type checker for validation.

Argument and return value. Table 4 shows that HiTyper outperforms all baseline approaches on argument type prediction by at least 6.8%. The improvement of HiTyper on predicting argument types may be attributed to the involved user-defined type correction algorithm, as some wrong argument predictions by the DL model could be corrected. HiTyper’s F1 score on inferring the types of return value outperforms Typilus by 39.5%, which demonstrates the advantage of static inference in handling return values.

Inference ability. Table 5 shows the number of correct and wrong annotations inferred by HiTyper and Typilus. From this table we can see there are at least 2,300 type slots in the test set which can only be inferred by HiTyper, accounting for 11% of total slots, no matter how many predictions HiTyper accepts from Typilus (top-1, 3, 5). This result demonstrates that HiTyper can infer more types based on the recommended predictions from the DL model, not only by simply copying them. It is worth noting that there are also some type slots that can only be inferred by Typilus, towards which we conclude the possible reasons as follows: 1) Wrong candidate types from the DL model could make the type rejection rules reject correct types. For example, for an operation a[b], if the DL model recommends List[str] for a and int for b, then rejection rules will reject recommended types for both a and b even if one of the type may be correct. 2) Not all human annotations are correct. We will further discuss this in Section 7.

5.2 RQ 2: How well does HiTyper handle the challenges elaborated in Section 1?

We elaborate four challenges faced by DL models in Section 1. In this RQ we analyze whether the proposed HiTyper could alleviate these challenges. Since the type drift issue can be mitigated by type graphs, we only discuss the other three challenges, i.e., fixed type set, type correctness, and composite data type.

Fixed type set. It is expected that the naive baseline fails to predict all the user-defined types since the approach selects from the top ten types as the results and the user-defined types are generally not ranked in the top ten. For the DL models, they also show bad performance when the correct types are not included in the fixed type set. For example, Typilus only achieves a top-1 F1 score of user-defined type at 0.34 even if it copies types from the training set to enlarge the type set. Benefiting from static inference and user-defined type correction algorithm, HiTyper gets a top-1 F1 score of 0.47, outperforming Typilus by 38.2%. This shows that our approach can alleviate the fixed type set issue.

Composite type. Data-driven methods show poor performance on predicting composite types. For example, the naive baseline only achieves a top-1 F1 score of 0.01 and Typilus only gets a top-1 F1 score of 0.29. As static inference approach constructs composite types according to static constraints, instead of selecting one from a type set, it shows better performance on inferring such types. Therefore, HiTyper achieves a top-1 F1 score of 0.35, outperforming Typilus by 20.7%, alleviating the composite type prediction issue of DL models. We notice that the performance here still has much room to be improved. An approach needs to correctly infer all the types of elements before it can correctly infer the entire composite type. This hinders current approaches from precisely inferring composite types.

Type correctness. DL models often return many candidate types for a type slot without checking their correctness, which causes the type correctness issue. Table 6 shows the average number of candidate types added per type slot by HiTyper and Typilus, where the Top-1 result is not listed as HiTyper does not reject any type if only one type is recommended from the DL models. In the table we can see that HiTyper adds 2.42 and 2.22 candidate types for arguments and return value respectively, even if it accepts top-5 predictions from Typilus. This means that HiTyper rejects almost half of the types recommended by Typilus using the designed rejection rules, indicating that HiTyper can improve the correctness of the predicted types.

5.3 RQ3: Ablation Study

Table 7 shows ablation analysis result of HiTyper. The Overall framework entry indicates the results when enabling all parts of HiTyper. By disabling DL models in HiTyper (Only static inference), the F1 score on arguments drops dramatically from 0.63 to 0.13, which demonstrates the superior performance of DL models on predicting types of function arguments. The Top-1 F1 score on user-defined types drops significantly from 0.47 to 0.39 when
We conduct case studies to further analyze the advantage of HiTyper and why HiTyper fails in some cases.

5.4 Case Study

We conduct case studies to further analyze the advantage of HiTyper and why HiTyper fails in some cases.
Table 7: Ablations of HiTyper when removing its component or changing its type matching methods (F1 Score)

| Ablation                                      | Argument   | Return Value | User Defined |
|-----------------------------------------------|------------|--------------|--------------|
|                                              | Top-1 | Top-3 | Top-5 | Top-1 | Top-3 | Top-5 | Top-1 | Top-3 | Top-5 |
| Only Static Inference (No DL models)          | 0.13  | -    | -      | 0.52  | -    | -      | 0.08  | -    | -      |
| No Type Rejection                             | 0.63  | 0.72 | 0.74   | 0.60  | 0.65 | 0.66   | 0.48  | 0.56 | 0.60   |
| No Type Correction                            | 0.60  | 0.68 | 0.70   | 0.57  | 0.61 | 0.62   | 0.39  | 0.46 | 0.47   |
| Type Correction - Only Variable Names         | 0.57  | 0.64 | 0.64   | 0.58  | 0.63 | 0.65   | 0.35  | 0.39 | 0.40   |
| Type Correction - Only DL models              | 0.62  | 0.69 | 0.72   | 0.58  | 0.63 | 0.64   | 0.43  | 0.50 | 0.53   |
| Overall Framework                             | 0.63  | 0.71 | 0.74   | 0.60  | 0.65 | 0.66   | 0.47  | 0.55 | 0.59   |

from the argument cog_names. To infer the type of this return value, HiTyper first builds a type graph for function _load(), then it finds that the types of load...,failed2 need to be inferred before it infers the return value. However, the variable name is crucial to infer these types. name is derived from cog_names so HiTyper finally identifies cog_names as a hot type slot. Fortunately Typilus successfully predicts the type of cog_names as List[str] by leveraging the naming information. Given the prediction of cog_names from Typilus, HiTyper successfully infers the type of name as str, and further infers the type of load, failed, notfound, loaded as List[str] and the type of failed2 as List[Tuple[str]]. Thus the type of return value is Tuple[Union[List[str], List[Tuple[str]]]]. This case indicates that the return value of _load() cannot be correctly inferred without employing both static inference and DL models.

Listing 3 shows an example for which both HiTyper and Typilus fail to infer a variable type. The return statement at line 7 indicates that the type of return value is the same as the type of argument node. Typilus gives the prediction ForEachClauseNode for them, which is an invalid type since it is not imported in this file and is from other projects in the training set. HiTyper corrects it to AbstractNode as this type has the highest similarity with the variable name node and Typilus’s prediction. However, the ground truth type annotated by the developer is AbstractNode, which is a parent type of Node. The case implies a new challenge in the type prediction task: Subtype prediction. It is difficult for current approaches to accurately choose the correct type from several subtypes and natural language information also provides limited help as subtypes often share similar names.

6 RELATED WORK

Static and dynamic type inference. Existing static type inference techniques towards different programming languages, such as Java [4], JavaScript [17], Ruby [9], Python [14]. Inference tools used in industry such as Pytype [34], Pysonar2 [33] and Pyre [31] are sound by design with relatively high accuracy on some simple built-in types and generic types, but due to the dynamic feature of programming languages, they can hardly handle dynamically generated types such as user-defined types and some complicated generic types. Dynamic type inference techniques [3, 37] and type checkers such as Mypy [27], Pytype [34], Pyre Check [31], Pyright [32] calculate the type flow between functions and infer types according to several input cases and typing rules. Compared with static inference methods, they can generate more accurate type information but have limited code coverage and large time consumption. Thus, they encounter difficulties when adopted on large scales of code.

Machine learning in type inference. Traditional static and dynamic type inference techniques employ rule-based methods and give the only predicted type for each type slot. Xu et al. [42] introduce probabilistic type inference, which returns several candidate types for one variable. Hellendoorn et al. [16] regard types as word labels and build a sequence model DeepTyper to infer types. However, their model leads to “type drift” since it treats each variable occurrence as a new variable without strict constraints. Dash et al. [6] introduce conceptual types which divide a single type such as str to more detailed types such as url, phone, etc. Pradel et al. [29] design 4 separate sequence models for source code to infer function types in Python. They also add a validation phase to filter out most wrong predictions using type checkers. Allamanis et al. [1] propose a graph model to represent code and use KNN to predict type. This method enlarges type set but still fails when predicting types not occurring in the training set.

7 DISCUSSION

In the evaluation of our approach, we also identify several new challenges as follows:

Subtype prediction. When correcting types recommended from neural networks, HiTyper calculates the similarity between two type strings. However, we find that it is difficult for current approaches to identify the difference between two subtypes, such as Node and AbstractNode.

Soundness of human annotation. When checking the evaluation result of HiTyper, we find that human annotations are not always correct. For example, it’s quite common that developers declare a function, annotate the types of its arguments and the return
values but do not implement the function body for fast prototyping. They put a pass statement or raise a NoImplementedError as a place holder. This causes a contradiction between the function body and the annotation since a function containing only a pass statement or a NoImplementedError always returns None rather than the annotations added by developers.

**Dataset.** We directly adopt the dataset used by Typilus in order to fairly compare HiTyper with it. This dataset contains lots of popular Python projects at GitHub, thus it represents the general type distribution of real-world projects. However, the type distribution may vary across specific projects. What’s more, current datasets on the type prediction task lack annotations of local variables, which largely limit the performance of DL models in predicting the types of local variables. We believe there will be further improvements on the performance of HiTyper when more comprehensive datasets are made available.

## 8 CONCLUSIONS

We present HiTyper, a static type inference framework with neural predictions to infer types in Python source code. HiTyper leverages type graph, type correction, and rejection rules to mitigate the four challenges faced by current neural networks. Experiments show that HiTyper significantly improves the performance of the SOTA DL model Typilus.

**REFERENCES**

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

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In International Conference on Learning Representations, volume abs/1711.00740, 2018.

[3] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Collective intelligence for smarter api recommendations in python, 2016.

[4] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards typewriter: A Hybrid Static Type Inference Framework with Neural Prediction. In Springer-Verlag.

[5] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. Ccc: Simple, efficient, context sensitive code completion. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 71–80, 2014.

[6] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. Refinym: Using names to refine types. In Proceedings of the 2014 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pages 107–117, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Andrea Renika D’Souza, Di Yang, and Cristina V. Lopes. Collective intelligence for smarter api recommendations in python, 2016.

[8] Python Software Foundation. Official documentation of Python3. 2020. https://docs.python.org/3/.

[9] 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, pages 1859–1866, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Philip Gage. A new algorithm for data compression. C Users J., 12(2):23–38, February 1994.

[11] Z. Gao, C. Bird, and E. T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 758–769, 2017.

[12] Github. The 2020 state of the octoverse, 2020. https://octoverse.github.com/.

[13] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Eric Tanter, and Andreas Steiek. An empirical study on the impact of static typing on software maintainability. Empirical Software Engineering, 19, 10 2014.

[14] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. Maxsm-based type inference for python 3. In Hana Chockler and Georg Weissenbacher, editors, Computer Aided Verification, pages 12–19, Cham, 2018. Springer International Publishing.
[42] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, page 607–618, New York, NY, USA, 2016. Association for Computing Machinery.

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

[44] Radim Rehůřek and Petr Sojka. Software framework for topic modelling with large corpora. pages 45–50, 05 2010.