Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python

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

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

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

Qirun Zhang
Georgia Institute of Technology
United States
qrzhang@gatech.edu

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

ABSTRACT

Type inference for dynamic programming languages such as Python is an important yet challenging task. Static type inference techniques can precisely infer variables with enough static constraints but are unable to handle variables with dynamic features. Deep learning (DL) based approaches are feature-agnostic, but they cannot guarantee the correctness of the predicted types. Their performance significantly depends on the quality of the training data (i.e., DL models perform poorly on some common types that rarely appear in the training dataset). It is interesting to note that the static and DL-based approaches offer complementary benefits. Unfortunately, to our knowledge, precise type inference based on both static inference and neural predictions has not been exploited and remains an open challenge. In particular, it is hard to integrate DL models into the framework of rule-based static approaches.

This paper fills the gap and proposes a hybrid type inference approach named HiTyper based on both static inference and deep learning. Specifically, our key insight is to record type dependencies among variables in each function and encode the dependency information in *type dependency graphs* (TDGs). Based on TDGs, we can easily integrate type inference rules in the nodes to conduct static inference and type rejection rules to inspect the correctness of neural predictions. HiTyper iteratively conducts static inference and DL-based prediction until the TDG is fully inferred. Experiments on two benchmark datasets show that HiTyper outperforms state-of-the-art DL models by exactly matching 10% more human annotations. HiTyper also achieves an increase of more than 30% on inferring rare types. Considering only the static part of HiTyper, it infers 2× ∼ 3× more types than existing static type inference tools. Moreover, HiTyper successfully corrected seven wrong human annotations in six GitHub projects, and two of them have already been approved by the repository owners.

*Corresponding author

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510038

1 INTRODUCTION

Dynamically typed programming languages such as Python are becoming increasingly prevalent in recent years. According to GitHub Octoverse 2019 and 2020 [15], Python outranks Java and C/C++ and becomes one of the most popular programming languages. The dynamic features provide more flexible coding styles and enable fast prototyping. However, without concretely defined variable types, dynamically typed programming languages face challenges in ensuring security and compilation performance. According to a recent survey by Jetbrains [23], static typing or at least some strict type hints becomes the top 1 desired feature among Python developers. To address such problems, some research adopts design principles of statically typed programming languages [16, 25, 47]. For example, reusing compiler backend of the statically typed languages [26] and predicting types for most variables [2, 4, 11, 17, 18, 21, 39]. Moreover, Python officially supports type annotations in the Python Enhancement Proposals (PEP) [28, 29, 56, 61].

Type prediction is a popular task performed by existing work. Traditional *static type inference* approaches [4, 11, 17, 21, 48] and type inference tools such as Pytype [45], Pysonar2 [42], and Pyre Infer [40] can correctly infer types for the variables with enough static constraints, e.g., for `a = 1` we can know the type of `a` is `int`, but are unable to handle the variables with few static constraints, e.g. most function arguments or dynamic evaluations such as `eval()` [51].

With the recent development of *deep learning (DL) methods*, we can leverage more type hints such as identifiers and existing type annotations to predict types. Many DL-based methods [2, 18, 31, 35, 39, 59] have been proposed, and they show significant improvement

compared with static techniques [27]. While DL-based methods are effective, they face the following two major limitations:

(i) *No guarantee of the type correctness.* Pradel *et al.* [39] find that the predictions given by DL models are inherently imprecise as they return a list of type candidates for each variable, among which only one type is correct under a certain context. Besides, the predictions made by DL models may contradict the typing rules, leading to type errors. Even the state-of-the-art DL model Typilus [2] generates about 10% of predictions that cannot pass the test of a type checker. The type correctness issue makes the DL-based methods hard to be directly deployed into large codebases without validation. Recent work [2, 39] leverages a search-based validation in which a type checker is used to validate all combinations of types returned by DL models and remove those combinations containing wrong types. However, these approaches cannot correct the wrong types but only filter them out.

(ii) *Inaccurate prediction of rare types.* Rare types refer to the types with low occurrence frequencies in datasets [2]. Low-frequency problem has become one of the bottlenecks of DL-based methods [24, 30, 46, 50, 60]. For example, Typilus's accuracy drops by more than 50% for the types with occurrence frequencies fewer than 100, compared to the accuracy of the types with occurrence frequencies more than 10,000. More importantly, rare types totally account for a significant amount of annotations even though each of them rarely appears. We analyze the type frequencies of two benchmark datasets from Typilus [2] and Type4Py [34], and find a *long tail phenomenon*, i.e., the top 10 types in the two datasets already account for 54.8% and 67.8% of the total annotations, and more than 10,000 and 40,000 types in two datasets are rare types with frequency proportions less than 0.1%. They still occupy 35.5% and 25.5% of total annotations for the two respective datasets and become the long "tail" of type distributions.

To remedy the limitations of the previous studies, this paper proposes a hybrid type inference framework named HiTyper, which conducts static type inference and accepts recommendations from DL models (*Static+DL*). We propose a novel representation, named type dependency graph (TDG), for each function, where TDG records the type dependencies among variables. Based on TDG, we reformulate the type inference task into a blank filling problem where the "blanks" (variables) are connected with dependencies so that both static approaches and DL models can fill the types into "blanks".

HiTyper infers the "blanks" in TDG mainly based on static type inference, which automatically addresses DL models' rare type prediction problem since static type inference rules are insensitive to type occurrence frequencies. HiTyper extends the inference ability of static type inference by accepting recommendations from DL models when it encounters some "blanks" that cannot be statically inferred. Different from the search-based validation by Pradel *et al.* [39], HiTyper builds a series of type rejection rules to filter out all wrong predictions on TDG, and then continues to conduct static type inference based on the reserved correct predictions.

We evaluate HiTyper on two public datasets. One dataset is released by Allamanis *et al.* in the paper of Typilus [2], and the other is ManyTypes4Py [35], one large dataset recently released for this task. Experiment results show that HiTyper outperforms both SOTA DL models and static type inference tools. Compared

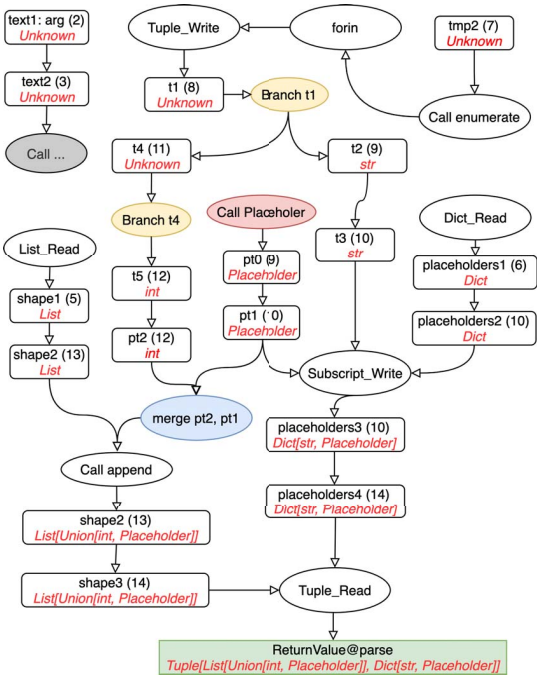

Figure 1: Type dependency graph of the parse() from Code.1.

with two SOTA DL models Typilus and Type4Py, HiTyper presents a 10%~12% boost on the performance of overall type inference, and a 6% ~ 71% boost on the performance of certain kinds of type inference such as return value type inference and user-defined type inference. Without the recommendations from neural networks and only looking at the static type inference part, HiTyper generally outputs 2× ~ 3× more annotations with higher precision than current static type inference tools Pyre [40] and Pytype [45]. HiTyper can also identify wrong human annotations in real-world projects. We identify seven wrong annotations in six projects of Typilus's dataset and submit pull requests to correct these annotations. Two project owners have approved our corrections.

Contributions. Our contributions can be concluded as follows:

- To the best of our knowledge, we are the first to propose a hybrid type inference framework that integrates static inference with DL for more accurate type prediction.
- We design an innovative type dependency graph to strictly maintain type dependencies of different variables.
- We tackle some challenges faced by previous studies and design a series of type rejection rules and a type correction algorithm to validate neural predictions.
- Extensive experiments demonstrate the superior performance of the proposed HiTyper than SOTA baseline models and static type inference tools in the task.

2 MOTIVATING EXAMPLE

Listing 1 illustrates an example of code snippet from the WebDNN project.[1] Results of several baselines, including static type inference

[1]https://github.com/mil-tokyo/webdnn

techniques - Pytype and Pysonar2, and state-of-the-art DL models - Typilus, are depicted in Table 1.

```
1  #src/graph_transpiler/webdnn/graph/shape.py
2  def parse(text):
3      normalized_text = _normalize_text(text)
4      tmp = ast.literal_eval(normalized_text)
5      shape = []
6      placeholders = {}
7      for i, t in enumerate(tmp):
8          if isinstance(t, str):
9              pt = Placeholder(label=t)
10             placeholders[t] = pt
11         elif isinstance(t, int):
12             pt = t
13         shape.append(pt)
14     return shape, placeholders
```

Listing 1: A Function from WebDNN.

Static Inference. According to Table 1, we can find that the static type inference techniques fail to infer the type of the argument `text` since the argument is at the beginning of data flow without any assignments or definitions. One common solution to infer the type is to use inter-procedural analysis and capture the functions that call `parse()` [52]. However, tracing the functions in programs, especially in some libraries, is not always feasible. As for the return value, by analyzing the data flow and dependencies between variables, static inference can easily identify that `shape` (line 5, 13) and `placeholders` (line 6, 10) consist of the return value. It can recursively analyze the types of the two variables, and finally output the accurate type of the return value. Indeed, both Pysonar2 and Pytype can correctly infer that the return value is a tuple containing a list and dict.

DL Approach. The DL model Typilus [2] accurately predicts the type as `str` according to the semantics delivered by the argument `text` and contextual information. The case illustrates that DL models can predict more types than static inference. However, Typilus fails to infer the type of the return value of `parse()`. Current DL models cannot maintain strict type dependencies between variables. Therefore, Typilus only infers the type as a tuple but cannot accurately predict the types inside the tuple. When adding a type checker to validate Typilus's predictions, its argument prediction is reserved since it does not violate any existing type inference rules. However, for the return value, its 2nd and 3rd type predictions in Table 1 by Typilus are rejected since the return value of `parse()` explicitly contains two elements with different types. The 1st prediction is also rejected because it contains the type `Optional[text]` that does not appear in the return value. In this case, the model does not produce any candidate type for the return value.

Static+DL Approach. For the code example, we find that static inference is superior than DL models when sufficient static constraints or dependencies are satisfied, while DL models are more applicable for the types lacking sufficient static constraints. Given the code, HiTyper first generates the TDG of it, as shown in Fig. 1, and tries to fill all nodes in TDG with corresponding types ("blank filling"). For the argument `text`, HiTyper identifies that the type cannot be inferred by static inference (it does not have any input edges) and asks DL for recommendations. HiTyper does not directly output the predictions from DL as final type assignments. Instead,

Table 1: Prediction results of different baselines for Listing 1.

Approach	Baseline	Argument	Return Value
	Ground Truth	str	Tuple[List[int, Placeholder], Dict[str, Placeholder]]
Static	Pysonar2	?	Tuple[List[int],Dict]
	Pytype	?	Tuple[List, Dict]
DL	Typilus	1. str	1. Tuple[collections.OrderedDict[Text, List[DFAState]], Optional[Text]],Tuple[Any, List[Tuple[Any]], Any] 2. Tuple[Text] 3. Tuple[torch.Tensor]
Static + DL	HiTyper (Typilus)	str	Tuple[List[int, Placeholder], Dict[str, Placeholder]]

HiTyper validates the prediction's correctness and accepts the result only if no type inference rules are violated. When predicting the return value, HiTyper captures its type dependencies based on the TDG (it connects with two input nodes) and directly leverages static inference to infer the type. For this case, DL predictions are not required, largely avoiding the imports of wrong types.

3 HITYPER

In this section, we first introduce the definitions used in HiTyper and then elaborate the details of HiTyper.

3.1 Definition of Types

Fig. 3 shows the definitions of different types according to the official documentation of Python [9] and its type checker `mypy` [8]. Note that we remove the `object` type and `Any` type since they are not strict static types. In general, all types can be classified into built-in types and user-defined types. Built-in types are predefined in the language specification of Python while user-defined types are created by developers. Developers can define the operations or methods supported by a user-defined type and overwrite some built-in operations for their user-defined types. For example, developers can define an `__add__()` method in a class so that two types derived from this class can be directly added together using the built-in operator `+`. The operation is called *operator overloading*. We create a subcategory for user-defined types with operator overloading behaviors since they have different type inference rules.

The type categories showed in Fig. 3 are widely used in most static type inference techniques [37, 40, 45]. Differently, DL-based studies [2, 35] generally categorize the types into *common types* and *rare types* based on a pre-defined threshold of occurrence frequencies (e.g., 100 in [2]). For a fair comparison, we also follow this definition for evaluation. By analyzing the rare types in two public datasets Typilus and ManyTypes4Py, we find that 79.02% and 99.7% of rare types actually are user-defined types. Because static inference technique is frequency-insensitive and cannot recognize rare types, we mainly add supports for user-defined types on static inference side of HiTyper.

3.2 Overview

HiTyper accepts Python source files as input and outputs JSON files recording the type assignment results. Fig. 2 illustrates its

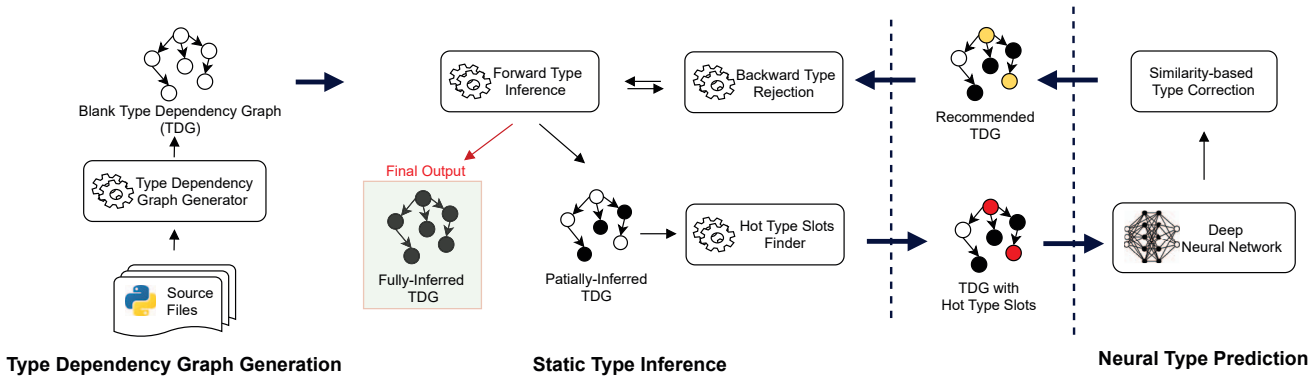

Figure 2: Overall architecture of HiTyper. Black solid nodes, hollow nodes, red nodes and yellow nodes in the type dependency graphs represent inferred type slots, blank type slots, hot type slots, and the type slots recommended by DL model, respectively.

$$\theta \in \textit{Type}\ (\Theta) ::= \gamma \mid \alpha[\theta, ..., \theta] \mid u \mid \textbf{None} \mid \textbf{type}$$
$$\gamma \in \textit{Elementary Type}\ (\Gamma) ::= \textbf{int} \mid \textbf{float} \mid \textbf{str} \mid \textbf{bool} \mid \textbf{bytes}$$
$$\alpha \in \textit{Generic Type}\ (A) ::= \textbf{List} \mid \textbf{Tuple} \mid \textbf{Dict} \mid \textbf{Set} \mid \textbf{Callable} \mid \textbf{Generator} \mid \textbf{Union}$$
$$b \in \textit{Builtin Type}\ (B) ::= \gamma \mid \alpha[\theta]$$
$$u \in \textit{User Defined Type}\ (U) ::= \textit{all classes and named tuples in code}$$
$$o \in \textit{Overloading User Defined Type}\ (O) ::= \textit{all classes with operator overloading in code}$$

Figure 3: Types in Python.

overall architecture. HiTyper includes three major components: type dependency graph generation, static type inference, and neural type prediction. The static type inference component comprises two main steps, i.e., forward type inference and backward type rejection.

Type Dependency Graph (TDG) Generation. Specifically, given a Python source file, HiTyper first generates TDGs for each function and identifies all the imported user-defined types (Sec. 3.3). TDG transforms every variable occurrence and expression into nodes and maintains type dependencies between them so that static inference and DL models can work together to fill types into it.

Static Type Inference - Forward Type Inference. To maintain the correctness of prediction results, HiTyper focuses on inferring types using static inference. Given a TDG, HiTyper conducts forward type inference by walking through the graph and implementing the type inference rules saved in each expression node (Sec. 3.4). However, due to the limitation of static inference, in most cases HiTyper can only infer partial type slots, i.e., variables, indicated as black solid nodes in the *partially-inferred TDG* in Fig. 2; while the blank nodes denote the type slots without sufficient static constraints and remaining unsolved. To strengthen the inference ability of HiTyper, we ask DL models for recommendations.

Neural Type Recommendation. Through the *hot type slot finder*, HiTyper identifies a key subset of the blank nodes as hot type slots, marked as red nodes in Fig. 2, for obtaining recommendations from DL models. HiTyper also employs a similarity-based type correction algorithm to supplement the prediction of user-defined types, which are the primary source of rare types (Sec. 3.5). The types recommended by the neural type prediction component are filled into the graph, resulting in the *recommended TDG*.

Static Type Inference - Backward Type Rejection. HiTyper utilizes type rejection rules to validate the neural predictions in hot type slots (Sec. 3.4). Then it traverses the whole TDG to transmit the rejected predictions from output nodes to input nodes so that all nodes in TDG can be validated. Finally HiTyper invokes forward type inference again to infer new types based on the validated recommendations.

The interactions between forward type inference and backward type rejection could iterate until the TDG reaches a fixed point, i.e., the types of all nodes do not change any more. Meanwhile, the iterations between static inference and neural prediction can repeat several times until all type slots are inferred, or a maximum iteration limit is reached.

3.3 Type Dependency Graph Generation

This section introduces the creation of type dependency graph (TDG), which describes the type dependencies between different variables in programs. Fig. 4 presents the syntax of all the expressions that generate types in Python, where each expression corresponds to a node in the AST (Abstract Syntax Tree). Given the AST of a program, HiTyper can quickly identify these expressions. The expression nodes constitute a major part of TDG. We define TDG as below.

Definition. We define a graph $G = (N, E)$ as a type dependency graph (TDG), where $N = \{n_i\}$ is a set of nodes representing all variables and expressions in source code, and E is a set of directed edges of $n_i \rightarrow n_j$ indicating the type of n_j can be solved based on the type of n_i by type inference rules. We also denote n_i is the input node of n_j and n_j is the output node of n_i here.

The TDG contains four kinds of nodes:

- *symbol* nodes represent all the variables for which the types need to be inferred. We also use *type slots* to indicate symbol nodes in the following sections.
- *expression* nodes represent all the expressions that generate types as shown in Fig. 4.
- *branch* nodes represent the branch of data flows.

$$
\begin{aligned}
e \in Expr ::= {} & v \mid c \mid e \ \textbf{blop}\ e \mid e \ \textbf{numop}\ e \mid \\
& e \ \textbf{cmpop}\ e \mid e \ \textbf{bitop}\ e \mid \\
& (e, ..., e) \mid [e, ..., e] \mid \\
& \{e : e, ..., e : e\} \mid \{e, ..., e\} \mid \\
& [e \ \textbf{for}\ e \ \textbf{in}\ e] \mid \{e \ \textbf{for}\ e \ \textbf{in}\ e\} \mid \\
& \{e : e \ \textbf{for}\ e, e \ \textbf{in}\ e\} \mid (e \ \textbf{for}\ e \ \textbf{in}\ e) \mid \\
& e(e, ..., e) \mid e[e : e : e] \mid e.v \\
v \in Variables ::= {} & \textit{all identifiers in code} \\
c \in Constants ::= {} & \textit{all literals in code} \\
blop \in Boolean\ Operations ::= {} & And \mid Or \mid Not \\
numop \in Numeric\ Operations ::= {} & Add \mid Sub \mid Mult \mid Div \mid Mod \mid \\
& UAdd \mid USub \\
bitop \in Bitwise\ Operations ::= {} & LShift \mid RShift \mid BitOr \mid BitAnd \mid \\
& BitXor \mid FloorDiv \mid Invert \\
cmpop \in Compare\ Operations ::= {} & Eq \mid NotEq \mid Lt \mid LtE \mid Gt \mid GtE \mid \\
& Is \mid IsNot \mid In \mid NotIn
\end{aligned}
$$

Figure 4: The syntax of expressions for typing in Python

- *merge* nodes represent the merge of data flows.

HiTyper creates a node for every variable occurrence instead of every variable in TDG because Python's type system allows variables to change their types at run-time. Similar to static single assignment (SSA), HiTyper labels each occurrence of a variable with the order of occurrences, so that each symbol node in the TDG has a format of `$name$order($lineno)` to uniquely indicate a variable occurrence. For example, in Fig. 1, we create three symbol nodes (`pt0(9)`, `pt1(10)`, `pt2(12)`) for variable `pt` as it appears three times in Listing 1 (Line 9, 10, and 12).

Import Analysis. Before establishing TDG for every input function, HiTyper first conducts import analysis to extract all user-defined types so that it can distinguish the initialization of user-defined types from regular function calls. HiTyper first collects all classes in source files, which constitute the initial set of user-defined types. Then it analyzes all local import statements such as "*from package import class*", and adds the imported classes into the user-defined type set. For all global import statements such as "*import package*", HiTyper locates the source of this package and adds all the classes and named tuples in the source into the user-defined type set. For each imported class, HiTyper solves the location of external source files and checks whether operator overloading methods exist in this class.

Type Dependency Graph Generation. Given the AST of input code and all the user-defined types extracted by import analysis, HiTyper creates TDG for each function based on the main logic shown in Alg. 1. HiTyper first locates all the variables and expressions in the code by traversing the whole AST. Specifically, to visit each AST node, HiTyper employs the ASTVisitor provided by Python's module *ast* [43]. HiTyper identifies expressions according to the definitions of expression nodes in Python (as depicted in Fig. 4) and records every visited expression node using an expression stack. Whenever HiTyper identifies an expression node (Line 3), it builds a same node in the current TDG and pushes it into the expression stack. HiTyper will then recursively visits the expression's operands to capture new expression nodes until it encounters a variable node (Line 12), which is the leaf node of the AST.

Algorithm 1 Type Dependency Graph Generation

Input: The AST of given function, *func_ast*;
Output: Type dependency graph of the given function, *tg*

```
    Initialize an expression stack ex_stack
    Initialize a variable stack var_stack
 1: for all node ∈ func_ast && node is not visited do
 2:     // handle expression nodes
 3:     if node.type ∈ Expressions then
 4:         ex_stack.push(node); ex_node ← new ex(node)
 5:         visit(node.operands); ex_stack.pop(node)
 6:         if not ex_stack.empty() then
 7:             tg.addEdge(ex_node → ex_stack.top())
 8:         end if
 9:         tg.addNode(ex_node)
10:     end if
11:     // handle symbol nodes
12:     if node.type == ast.Name then
13:         sym_node ← new symbol(node)
14:         if node.ctx == write then
15:             tg.addEdge(ex_stack.top() → sym_node)
16:         else
17:             tg.addEdge(var_stack.top() → sym_node)
18:             tg.addEdge(sym_node → ex_stack.top())
19:         end if
20:         var_stack.push(sym_node); tg.addNode(sym_node)
21:     end if
22:     // handle branch and merge nodes
23:     if checkTypeBranch(node) then
24:         branch_node ← new branch(node)
25:         tg.addNode(branch_node)
26:         ctx1, ctx2 ← Branch(ctx)
27:         visit(node.left, ctx1); visit(node.right, ctx2)
28:     end if
29:     if checkTypeMerge(node) then
30:         merge_node ← new merge(node)
31:         tg.addNode(merge_node)
32:         ctx ← Merge(ctx1, ctx2)
33:     end if
34: end for
```

HiTyper builds a symbol node in TDG for each visited identifier node of AST, and maintains a variable map to record all the occurrences of each variable. The AST already indicates the context of each variable occurrence, i.e., whether *read* or *write*.

(i) If the variable context is *read*, HiTyper will obtain the last occurrence of the variable according to the maintained variable map under the current context. It then creates an edge from the symbol node of the last occurrence to the symbol node of the current variable (Line 16 - 18).

(ii) If the variable context is *write*, HiTyper will fetch the value from the last expression in the expression stack and build an edge

connecting from the expression node to the symbol node of the current variable (Line 14 - 15).

Analogous to regular data flow analysis, HiTyper also checks whether the data flow branches (Line 23 - 27) or merges at certain locations (Line 29 - 32).

In TDG, each symbol node keeps a list of candidate types while each expression node includes type inference rules and type rejection rules. When HiTyper walks through TDG, the rules will be activated to produce new types. Thus, types can *flow* from arguments to return values. By traversal, HiTyper obtains the types of each symbol node and outputs the type assignment. The leveraged type inference rules and type rejection rules are detailed in the next subsections.

3.4 Static Type Inference

This section describes the type inference and rejection rules integrated in expression nodes, which are the key component of our static type inference. Fig. 5 denotes all the type inference and rejection rules used in static type inference. Each rule consists of some premises (contents above the line) and conclusions (contents below the line). They obey the following form:

$$\pi \vdash e : \theta.$$

In this form, π is called the context, which includes lists that assign types to expression patterns. e is the expression showed in Fig. 4, and we use $e_1, ..., e_n$ to represent different expressions. θ is the type showed in Fig. 3. We use $\theta_1, ..., \theta_n$ to represent different types. A rule under this form is called a *type judgement* or *type assignment*. Our goal is to get the context π that assigns types to all the variables in code.

The premises of each rule in Fig. 5 are the types of input nodes $\theta_1, \theta_2, ...$ that constructs an expression, and the valid type set $\widetilde{\theta}$ for the current operation. Usually type inference rules only have one conclusion, which is the result type of current expression. However, as we also involve neural predictions in TDG and use type rejection rules to validate them, the conclusions of each rule in Fig. 5 have two parts: 1) the result type of current expression node and 2) the validated types of input nodes. (Some rules may not have the second part because they accept any input types.)

The result type of the current expression node is what traditional static type inference techniques usually infer. We denote it as *forward type inference*. However, there exist types that are not allowed to conduct certain operations, which are guided by *type constraints*. When a type constraint is violated, e.g., adding an integer to a string, traditional static inference techniques [37, 40, 45] throw type errors. For the wrongly predicted cases, HiTyper does not directly throw a type error since it accepts recommendations from DL models. To "sanitize" the recommendations from DL models, we create type rejection rules to validate and remove the wrong predictions in input nodes. We call this as *backward type rejection*.

Forward Type Inference. HiTyper starts forward type inference with the nodes that do not have input nodes in TDG. It gradually visits all nodes in the graph and activates corresponding type inference rules if their premises are satisfied, i.e., all input nodes are fully inferred. This is the forward traversal of TDGs. As forward type inference in HiTyper is similar to traditional static type inference techniques, we only discuss the [Call] rule for which HiTyper has a special strategy. The premise of the [Call] rule requires the type of callees, which is beyond the scope of current functions. This premise is one major barrier for most static inference techniques to fully infer a program due to a large number of external APIs in Python programs [19, 52]. HiTyper only focuses on inferring the types of functions with explicit implementation in the current source code, in which the TDGs of the functions are connected. HiTyper does not infer external calls for two reasons: 1) DL models perform well on predicting the types of commonly-used APIs that frequently occur in the training set; 2) Python maintains a *typeshed* [44] project to collect the type annotations of frequently-used modules, so HiTyper can directly access the types.

Backward Type Rejection. An input type in an expression must fulfill two constraints before it can conduct the expression: 1) it must be the valid type to conduct a certain expression, 2) it must have a valid relationship with other input types. HiTyper rejects the input types that violate these two constraints. It first checks whether the type is valid for an expression. We indicates valid types for each expression as $\widetilde{\theta}$ in Fig. 5. For example, in [In, NotIn] rule, the types of e_2 must be iterable so `int` is not allowed and should not be in the valid type set $\widetilde{\theta}$. Then HiTyper checks whether the relationships between all inputs are valid. Apart from valid types for a certain operation, some operations also require the inputs to satisfy a certain relationship. For example, in [Add] rule, the two operands must have the same type. For types of two inputs `int` and `str`, even though they are in the valid type set of this operation, they are still rejected because they are not the same type. Therefore, in the [Add] rule, the final valid input types are the intersection of all input type sets θ_1, θ_2 and valid type set $\widetilde{\theta}$.

Type Rejection rules can validate and reject the input types of an operation. However, the input types are the results of previous operations, so the type rejection process will also affect the input types of previous operations. To thoroughly remove the influence of wrong types, HiTyper also rejects the input types that result in the rejected types according to forward type inference rules. HiTyper gradually validates all type slots by starting from the type slots without output edges and producing the rejected input types. Then it traverses other slots until the whole TDG is visited. This is the backward traversal of TDGs.

Correctness. Different from the DL-based approaches [18, 35], HiTyper can always guarantee the correctness of its type assignments based on static inference. According to the architecture of HiTyper in Fig. 2, the type assignments generated by HiTyper have two cases: 1) If the static inference can successfully handle a program, HiTyper does not need to invoke DL models to give type recommendations. Consequently, the type assignments fully based on the inference rules (Fig. 5) are sound because they are collected from the Python official implementation CPython [10]; and 2) If the static inference cannot fully infer a program and the DL models are invoked to provide type recommendations (Sec. 3.5), HiTyper utilizes type rejection rules to validate the recommendations and then calls the type inference rules again to infer the remaining types. In this case, our rejection rules thoroughly eliminate the influence of wrong recommendations, and the final results are also produced by static inference. Therefore, HiTyper always maintains the type correctness.

$$\frac{v \in \mathbf{Dom}(\pi)}{\pi \vdash v : \theta} \quad \text{(Variable)}$$

$$\frac{}{\pi \vdash c : \theta} \quad \text{(Constant)}$$

$$\frac{\pi \vdash e : \theta}{\pi \vdash e.v : \theta.v} \quad \text{(Attribute)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\mathbf{bool},\ \mathbf{int},\ O\}}{\pi \vdash e_1\ \mathbf{bitop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \quad \text{(LShift, RShift)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \widetilde{\theta} = \{\mathbf{bool},\ \mathbf{int},\ \mathbf{float},\ O\} \quad \pi \vdash e_2 : \theta_2 \quad \theta' = getMorePreciseType(\theta_1 \wedge \widetilde{\theta}, \theta_2 \wedge \widetilde{\theta})}{\pi \vdash e_1\ \mathbf{numop}\ e_2 : \theta' \quad \pi \vdash \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash \theta_2 \wedge \widetilde{\theta}} \quad \text{(Numeric Operations)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \widetilde{\theta_1} = \{\mathbf{int}, \mathbf{bool}\} \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta_2} = \{\Gamma,\ \mathbf{List},\ \mathbf{Tuple},\ O\}}{\pi \vdash e_1\ \mathbf{numop}\ e_2 : \theta_2 \wedge \widetilde{\theta_2} \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta_2}} \quad \text{(Mult)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\Gamma,\ \mathbf{List},\ \mathbf{Tuple},\ O\}}{\pi \vdash e_1\ \mathbf{cmpop}\ e_2 : \mathbf{bool} \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \quad \text{(Lt,LtE,Gt,GtE)}$$

$$\frac{}{\pi \vdash u(e_1, ..., e_n) : u} \quad \text{(Class Instantiation)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad ... \quad \pi \vdash e_n : \theta_n}{\pi \vdash (e_1, ..., e_n) : \mathbf{Tuple}[\theta_1, ..., \theta_n] \quad \pi \vdash [e_1, ..., e_n] : \mathbf{List}[\theta_1, ..., \theta_n] \quad \pi \vdash \{e_1, ..., e_n\} : \mathbf{Set}[\theta_1, ..., \theta_n]} \quad \text{(Tuple, List, Set)}$$

$$\frac{\pi \vdash e : \theta \quad \widetilde{\theta} = \{A, \mathbf{str}, \mathbf{bytes}\} \quad \theta' = getElementType(\theta \wedge \widetilde{\theta})}{\pi \vdash \mathbf{for}\ v\ \mathbf{in}\ e : \theta' \quad \pi \vdash e : \theta \wedge \widetilde{\theta}} \quad \text{(Comprehension)}$$

$$\frac{\pi \vdash \mathbf{for}\ v\ \mathbf{in}\ e_1 : \theta_1 \quad \pi \vdash e_2[v] : \theta_2}{\pi \vdash (e_2[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1) : \mathbf{Generator}[\theta_2]} \quad \text{(Generator)}$$

$$\frac{\pi \vdash \mathbf{for}\ v\ \mathbf{in}\ e_1 : \theta_1 \quad \pi \vdash e_2[v] : \theta_2}{\pi \vdash [e_2[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1] : \mathbf{List}[\theta_2] \quad \pi \vdash \{e_2[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1\} : \mathbf{Set}[\theta_2]} \quad \text{(List, Set Comprehension)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2}{\pi \vdash e_1\ \mathbf{blop}\ e_2 : \mathbf{Union}[\theta_1, \theta_2]} \quad \text{(Boolean Operations)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\mathbf{bool},\ \mathbf{int},\ \mathbf{Set},\ O\}}{\pi \vdash e_1\ \mathbf{bitop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \quad \text{(BitOr, BitAnd, BitXor)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\Gamma,\ \mathbf{List},\ \mathbf{Tuple},\ O\}}{\pi \vdash e_1\ \mathbf{numop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \quad \text{(Add)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\Gamma,\ \mathbf{Set},\ O\}}{\pi \vdash e_1\ \mathbf{numop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \quad \text{(Sub)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2}{\pi \vdash e_1\ \mathbf{cmpop}\ e_2 : \mathbf{bool}} \quad \text{(Eq,NotEq,Is,IsNot)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\mathbf{str}, \mathbf{bytes}, \mathbf{List},\ \mathbf{Tuple},\ \mathbf{Set},\ \mathbf{Dict},\ \mathbf{Generator}\}}{\pi \vdash e_1\ \mathbf{cmpop}\ e_2 : \mathbf{bool} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \quad \text{(In,NotIn)}$$

$$\frac{\pi \vdash e : \theta \quad \pi \vdash e_1 : \theta_1 \quad ... \quad \pi \vdash e_n : \theta_n \quad \widetilde{\theta} = \{\mathbf{Callable}[[\theta_1, ..., \theta_n], \theta]\} \quad \theta' = getReturnType(\theta \wedge \widetilde{\theta})}{\pi \vdash e(e_1, ..., e_n) : \theta} \quad \text{(Call)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad ... \quad \pi \vdash e_n : \theta_n}{\pi \vdash \{e_1 : e_2, ..., e_{n-1} : e_n\} : \mathbf{Dict}[\theta_1 : \theta_2, ..., \theta_{n-1} : \theta_n]} \quad \text{(Dict)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta_1} = \{\mathbf{Dict}\} \quad \theta' = getValueType(\theta_1 \wedge \widetilde{\theta_1})}{\pi \vdash e_1[e_2] : \theta' \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1}} \quad \text{(SubScript)}$$

$$\frac{\pi \vdash \mathbf{for}\ v\ \mathbf{in}\ e_1 : \theta_1 \quad \pi \vdash e_2[v] : \theta_2 \quad \pi \vdash e_3[v] : \theta_3}{\pi \vdash \{e_2[v] : e_3[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1\} : \mathbf{Dict}[\theta_2 : \theta_3]} \quad \text{(Dict Comprehension)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta_1} = \{A, \mathbf{str}, \mathbf{bytes}\} \quad \widetilde{\theta_2} = \{\mathbf{int}, \mathbf{bool}\} \quad \theta' = getElementType(\theta_1 \wedge \theta_2)}{\pi \vdash e_1[e_2] : \theta' \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta_2}} \quad \text{(Slice)}$$

Figure 5: Type inference and rejection rules of expressions in Python

3.5 Neural Type Recommendation

HiTyper conducts static type inference based on type inference rules When static type inference can fully infer all the variables in TDG. However, some variables are hard to be statically typed so that HiTyper only gets a partially-inferred TDG. In this case, HiTyper asks DL models for recommendations. The neural type recommendation part of HiTyper includes two procedures: hot type slot identification and similarity-based type correction.

Hot Type Slot Identification. Some variables can impact the types of many other variables because they locate at the beginning of the data flow or possess type dependencies with many variables. We call these variables as *hot type slots*. Given the types of hot type slots, static type inference techniques can infer the remaining type slots. Therefore, to optimize the type correctness of HiTyper, DL models are only invoked on the hot type slots instead of all the blank type slots.

To identify the hot type slots, HiTyper first removes slots already filled by static type inference and obtains a sub-graph with all the blank type slots. Then HiTyper employs a commonly-used dominator identification algorithm semi-NCA [14] to capture all dominators in the sub-graph. A node X dominiating another node Y in a graph means that each entry node to Y must pass X. Thus, if a type slot X dominates another type slot Y, Y's type can be inferred from X's type. HiTyper gradually removes the type slots Y from

Table 2: Comparison with the baseline approaches. Top-1,3,5 of HiTyper means it accepts 1,3,5 candidates from deep neural networks in type recommendation phase. The neural network in HiTyper is the corresponding comparison DL model.

Dataset	Type Category	Approach	Top-1		Top-3		Top-5	
			Exact Match	Match to Parametric	Exact Match	Match to Parametric	Exact Match	Match to Parametric
ManyTypes4Py	Argument	Naive Baseline	0.14	0.16	0.33	0.38	0.43	0.51
		Type4Py	0.61	0.62	0.64	0.66	0.65	0.68
		HiTyper	**0.65**	**0.67**	**0.70**	**0.74**	**0.72**	**0.76**
	Return Value	Naive Baseline	0.07	0.10	0.19	0.28	0.28	0.42
		Type4Py	0.49	0.52	0.53	0.59	0.54	0.63
		HiTyper	**0.60**	**0.72**	**0.63**	**0.76**	**0.65**	**0.77**
	Local Variable	Naive Baseline	0.13	0.17	0.33	0.45	0.47	0.65
		Type4Py	0.67	0.73	0.71	0.78	0.72	0.79
		HiTyper	**0.73**	**0.85**	**0.74**	**0.86**	**0.75**	**0.86**
	All	Naive Baseline	0.13	0.16	0.31	0.40	0.43	0.57
		Type4Py	0.62	0.66	0.66	0.72	0.67	0.73
		HiTyper	**0.69**	**0.77**	**0.72**	**0.81**	**0.72**	**0.82**
Typilus's Dataset	Argument	Naive Baseline	0.19	0.20	0.38	0.42	0.46	0.50
		Typilus	0.60	0.65	0.69	0.74	0.71	0.76
		HiTyper	**0.63**	**0.68**	**0.72**	**0.76**	**0.76**	**0.79**
	Return Value	Naive Baseline	0.11	0.11	0.28	0.31	0.36	0.43
		Typilus	0.41	0.57	0.48	0.62	0.50	0.64
		HiTyper	**0.57**	**0.70**	**0.63**	**0.75**	**0.64**	**0.77**
	All	Naive Baseline	0.17	0.18	0.35	0.39	0.44	0.48
		Typilus	0.54	0.62	0.63	0.70	0.65	0.72
		HiTyper	**0.61**	**0.69**	**0.69**	**0.76**	**0.72**	**0.78**

Algorithm 2 Type correction of user-defined types

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 $\leftarrow None$;
5: $tw \leftarrow$ BPE(t); $namew \leftarrow$ BPE(*name*);
6: **for** each $pt \in S$ **do**
7: $ptw \leftarrow$ 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**

the sub-graph until no type slots can be removed. In the smallest sub-graph, each type slot is not dominated by other type slots, and all the slots are hot type slots. For these type slots, HiTyper accepts type recommendations from DL models.

Similarity-based Type Correction for User-defined Types. DL models provide one or more type recommendations for each hot type slot, depending on the strategy (Top-1, -3, or -5) HiTyper uses. Some DL models [18, 39] treat user-defined types as OOV tokens and do not predict the types, while other models [2, 45] directly copy user-defined types from the training set but fail to predict those never appearing in the training set. We propose to complement the recommendation of user-defined types using the similarity-based type correction algorithm shown in Alg. 2. Note that HiTyper only focuses on replacing the explicitly incorrect user-defined types, i.e., those never imported or defined in current source file, with the most similar user-defined types collected by import analysis.

Specifically, if the recommended type does not belong to built-in types, HiTyper checks whether the type appears in the user-defined type set collected from import analysis (Line 1). If the check result is False, the type will be regarded as explicitly incorrect and should be corrected. For these incorrect user-defined types, HiTyper replaces them with the most similar candidate in the user-defined type set. HiTyper employs Word2Vec [32] to embed two types and the variable name into word embeddings, and calculates the cosine distance as the similarity of the two types (Line 6-12). For the OOV tokens, HiTyper splits them into subtokens using the BPE algorithm [12, 53] (Line 5). Finally, HiTyper chooses the type candidate with the largest similarity to fill the user-defined type (Line 15).

4 EVALUATION

In the section, we answer the following research questions:
RQ1: How effective is HiTyper compared to baseline approaches?
RQ2: Can HiTyper well predict the rare types?

Table 3: Type distribution in the test set. "Rare" indicates rare types and "User" indicates user-defined types.

	Category	Total	Rare	User	Arg	Return	Local
Typilus	**Count**	15,772	7,103	5,572	11,261	4,511	-
	Prop.	100%	45%	35%	71%	29%	-
Type4Py	**Count**	37,408	14,035	10,023	11,807	5,491	20,110
	Prop.	100%	37%	27%	32%	15%	53%

RQ3: What is the performance of the static type inference component in HiTyper?

4.1 Experimental Setup

Dataset. We used the two Python datasets mentioned in Sec. 2 for evaluation. One is the *Typilus's Dataset* released by Allamanis *et al.* [2]; and the other one is *ManyTypes4Py* released by Mir *et al.* [34], with the number of different types in the test set and more detailed statistics shown in Table 3 and Sec. 2, respectively.

Evaluation Metrics. Following the previous work [2, 35], we choose two metrics *Exact Match* and *Match to Parametric* for evaluation. The two metrics compute the ratio of results that: 1) *Exact Match*: completely matches human annotations. 2) *Match to Parametric*: satisfy exact match when ignoring all the type parameters. For example, `List[int]` and `List[str]` are considered as matched under this metric.

Baseline Approaches. To verify the effectiveness of the proposed HiTyper, we choose five baseline approaches for comparison:

1) A naive baseline. It represents a basic data-driven method. We build this baseline following the work [39], which makes predictions by sampling form the distribution of the most frequent ten types.

2) Pytype [45] and Pyre Infer [40]. They are two popular Python static type inference tools from Google and Facebook, respectively.

3) Typilus [2] and Type4Py [35]. Typilus is a graph model that utilizes code structural information. Type4Py is a hierarchical neural network that uses type clusters to predict types.

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 [9] and its implementation CPython[2]. We use Word2Vec model from the gensim library [62] 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 [33]. We choose Typilus and Type4Py as the neural network model from which HiTyper accepts type recommendations. We choose the exact hyper-parameters for Typilus and Type4Py used in the original papers. We run all experiments on Ubuntu 18.04. The system has a Intel(R) Xeon(R) CPU (@2.4GHz) with 32GB RAM and 2 NVIDIA TiTAN V GPUs with 12GB RAM.

4.2 RQ 1: Effectiveness of HiTyper

We evaluate the effectiveness of HiTyper considering different type categories, including arguments, local variables, and return values. The results are depicted in Table 2.

[2]https://github.com/python/cpython

Overall performance. The naive baseline achieves high scores regarding the top-5 exact match metric for different type categories, some of which are even close to the performance of DL models. Since the naive baseline only predicts types with high occurrence frequencies in the dataset, the results indicate the challenge of accurately predicting rare types. Typilus and Type4Py mitigate the challenge by using similarity learning and type clusters and achieve ~0.6 regarding the top-1 exact match metric. HiTyper further improves the metric by 11% and 15% compared with Typilus and Type4Py, respectively. HiTyper also enhances the top-1 match to parametric metric by 17% and 11% compared with Typilus and Type4Py, respectively. The improvement indicates the effectiveness of HiTyper in accurate type prediction. Besides, HiTyper presents better performance than the respective DL models regarding the top-3,5 metrics, demonstrating that HiTyper infers new results based on the static type inference rules, instead of just filtering out or reordering the predictions of DL models.

Type categories. Both Type4Py and Typilus perform better on the argument category than the return value category, which may reflect the difficulty of predicting the types of return values. By building upon type inference rules and TDGs, HiTyper can handle the complicated type dependencies of return values and thereby improve Type4Py and Typilus by 22% and 39%, respectively, w.r.t. the Top-1 exact match metric. HiTyper also slightly meliorates the prediction of the argument category by 7% and 5% compared with Type4Py and Typilus, respectively. The improvement may be attributed to the type correction for user-defined types. Moreover, HiTyper outperforms Type4Py by 9% for predicting local variables.

Answer to RQ1: HiTyper shows great improvement (11% ~ 15%) on overall type inference performance, and the most significant improvement is on return value inference (22% ~ 39%).

4.3 RQ 2: Prediction of Rare Types

Rare types are defined as the types with proportions less than 0.1% among the annotations in the datasets, and we observe that 99.7% and 79.0% of rare types are user-defined types in ManyTypes4Py and Typilus's dataset, respectively. Table 4 illustrates the prediction results of rare types and user-defined types. We can observe that the naive baseline barely infers rare types and user-defined types. Besides, the performance of Type4Py and Typilus drops significantly for the two type categories, which indicates that type occurrence frequencies can impact the performance of DL models. HiTyper shows the best performance on predicting the two type categories. Specifically, for inferring the rare types, HiTyper outperforms Type4Py and Typilus by 31% and 34%, respectively, w.r.t. the top-1 exact match metric. Regarding the prediction of user-defined types, HiTyper increases the performance of Type4Py and Typilus by 69% and 47%, respectively.

Answer to RQ2: HiTyper greatly alleviates the prediction issue of rare types faced by DL models by achieving a > 30% boost, taking the advantage of the static type inference component.

Table 4: Comparison with the baseline DL approaches.

Dataset	Type Category	Approach	Top-1		Top-3		Top-5	
			Exact Match	Match to Parametric	Exact Match	Match to Parametric	Exact Match	Match to Parametric
ManyTypes4Py	User-defined Types	Naive Baseline	0.00	0.00	0.00	0.00	0.00	0.00
		Type4Py	0.29	0.29	0.34	0.34	0.36	0.36
		HiTyper	**0.49**	**0.49**	**0.56**	**0.56**	**0.58**	**0.58**
	Rare Types	Naive Baseline	0.03	0.07	0.08	0.21	0.13	0.35
		Type4Py	0.39	0.46	0.45	0.54	0.47	0.57
		HiTyper	**0.51**	**0.66**	**0.56**	**0.72**	**0.58**	**0.73**
Typilus's Dataset	User-defined Types	Naive Baseline	0.00	0.00	0.00	0.00	0.00	0.00
		Typilus	0.32	0.32	0.40	0.40	0.42	0.42
		HiTyper	**0.47**	**0.47**	**0.56**	**0.56**	**0.60**	**0.60**
	Rare Types	Naive Baseline	0.00	0.01	0.01	0.03	0.03	0.09
		Typilus	0.32	0.43	0.41	0.53	0.43	0.55
		HiTyper	**0.43**	**0.55**	**0.52**	**0.63**	**0.56**	**0.67**

Table 5: Comparison with static type inference tools.

Dataset	Type Category	Approach	Exact Match	#Correct Annotations
ManyTypes4Py	Argument	Pytype	-	0
		Pyre Infer	**0.96**	613
		HiTyper	0.94	**1060**
	Return Value	Pytype	0.81	777
		Pyre Infer	0.84	662
		HiTyper	**0.86**	**2603**
	All	Pytype	0.81	777
		Pyre Infer	**0.89**	1275
		HiTyper	0.88	**3663 (16918*)**
Typilus's Dataset	Argument	Pytype	-	0
		Pyre Infer	**0.96**	543
		HiTyper	0.88	**983**
	Return Value	Pytype	0.79	552
		Pyre Infer	0.71	484
		HiTyper	**0.91**	**2461**
	All	Pytype	0.79	552
		Pyre Infer	0.82	1027
		HiTyper	**0.90**	**3444**

* The number of correct annotations when including local variables.

4.4 RQ 3: Performance of the Static Type Inference Component

In this RQ, we evaluate the performance of the static type inference component in HiTyper compared with popular static type inference tools Pytype [45] and Pyre [40]. The results are shown in Table 5. We only consider the type categories of argument and return value for comparison since Pyre and Pytype do not infer types for local variables. We use the metric *number of correct annotations* to replace the metric *match to parametric* that is usually used to evaluate DL models, considering that the results of static inference are exact and not recommendations.

As shown in Table 5, the exact match scores of all the static tools are greatly high, and HiTyper achieves the best performance. The results indicate the effectiveness of the static type inference component in HiTyper. We also find that there remains ~10% of the results inconsistent with human annotations in the datasets. By using Python's official type checker *mypy* to check these results, we observe that all the types annotated by HiTyper do not produce type errors, which reflects the correctness of the proposed HiTyper. After manual checking of these inconsistent types, we find this inconsistency is caused by subtypes, we further discuss them in Sec. 5. Besides, *mypy*'s results indicate very few inconsistent cases are caused by incorrect human annotations. To test whether HiTyper can rectify the incorrect annotations, we replace the original annotations with the results inferred by HiTyper, and inspect whether the original type errors are fixed. We finally correct 7 annotations on 6 GitHub repositories, including memsource-wrap [13], MatasanoCrypto [1], metadata-check [58], coach [20], cauldron [54], growser [55], and submit pull requests to these repository owners. The owners of MatasanoCrypto and cauldron have approved our corrections.

While Pytype and Pyre present high exact match scores, the numbers of variables they can accurately infer are small. Table 5 shows that HiTyper generally outputs 2x argument types and 3x return value types compared with them in both datasets, which suggests HiTyper's stronger inference ability than Pyre and Pytype. Such improvements attribute to HiTyper's import analysis and [Class Instantiation] rule on supporting the inference of user-defined types, and inter-procedural analysis on supporting the inference of class attributes and functions.

Answer to RQ3: Only considering the static inference part, HiTyper still outperforms current static type inference tools by inferring 2× ~ 3× more variables with higher accuracy.

5 DISCUSSION

Inference of subtypes. Although HiTyper achieves promising results for type prediction and passes the check of *mypy*, it is still unable to infer some variable types (around 10%). The failure mainly occurs in the inference of subtypes.

```
#File: miyakogi.wdom/wdom/node.py
#Human annotation: AbstractNode
#Typilus: ForeachClauseNode      HiTyper: Node
def _append_element(self, node: AbstractNode) ->
     AbstractNode:
    if node.parentNode:
        node.parentNode.removeChild(node)
    self.__children.append(node)
    node.__parent = self
```

```
9       return node
10  def _append_child(self, node):
11      if not isinstance(node, Node):
12          raise TypeError
13      ...
14      return self._append_element(node)
```

Listing 2: An example HiTyper fails to infer.

Listing 2 shows an example for which HiTyper's result is inconsistent with the original annotations but still passes the check of *mypy*. The return statement at Line 9 indicates that the type of return value is the same as the type of argument `node`. Typilus predicts the type as `ForeachClauseNode`, which is invalid since it is not imported in the code and is from other projects in the training set. HiTyper infers the type as `xml.dom.Node`, because the function is called by another function named *_append_child* in the same file and the caller transmits a variable with type `Node`. However, developers annotate the variable as `AbstractNode`, the parent type. Such behavior is common in practice and poses a challenge for accurate type prediction.

6 RELATED WORK

Static and dynamic type inference. Existing static type inference techniques towards different programming languages, such as Java [4], JavaScript [21], Ruby [11], Python [17] or using different static analysis techniques [5, 7, 38], and inference tools used in industry such as Pytype [45], Pysonar2 [42] and Pyre [40] are correct by design with relatively high accuracy on some simple builtin types and generic types, but due to the dynamic feature [51] of programming languages, they can hardly handle user-defined types and some complicated generic types. HiTyper extends the inference ability of static inference techniques by conducting import analysis and inter-procedural analysis to handle the user-defined types, class attributes and functions in code. Dynamic type inference techniques [3, 36, 49] and type checkers such as Mypy [37], Pytype [45], Pyre Check [40], Pyright [41] calculate the data flow between functions and infer types according to several input cases. They can more accurately predict types than static type inference techniques but have limited code coverage and large time consumption. Thus, they encounter difficulties when deployed on large scales of code.

Machine learning in type inference. Traditional static and dynamic type inference techniques employ rule-based methods and give the exact predicted type for each type slot. Xu *et al.* [59] introduce probabilistic type inference, which returns several candidate types for one variable. Hellendoorn *et al.* [18] regard types as word labels and build a sequence model DeepTyper to infer types. However, their model 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.* [39] design 4 separate sequence models to infer function types in Python. They also add a validation phase to filter out most wrong predictions using type checkers. Allamanis *et al.* [2] propose a graph model to represent code and use KNN to predict the types. The method enlarges type set but still fails when the predicted types are not occurring in the training set. Although DL models have shown great improvement in this task, it still faces the type correctness and rare type prediction problem, HiTyper addresses these two problems by integrating DL models into the framework of static inference since static inference is data-insensitive and implemented on type inference rules that are sound by design. Despite efforts on Python type inference, there are also a bunch of work on type inference of other dynamically typed programming languages. Wei *et al.* [57] propose a neural graph network named LambdaNet to conduct probabilistic type inference on JavaScript programs. Jesse *et al.* [22] propose a BERT-style model named TypeBert that obtains better performance on type inference of JavaScript than most sophisticated models.

7 CONCLUSION

In the work, we propose HiTyper, a hybrid type inference framework which iteratively integrates DL models and static analysis for type inference. HiTyper creates TDG for each function and validates predictions from DL models based on typing rules and type rejection rules. Experiments demonstrate the effectiveness of HiTyper in type inference, enhancement for predicting rare types, and advantage of the static type inference component in HiTyper. HiTyper is open-sourced at https://github.com/JohnnyPeng18/HiTyper.

8 ACKNOWLEDGMENTS

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 14210920 of the General Research Fund), National Natural Science Foundation of China under project No. 62002084, Stable support plan for colleges and universities in Shenzhen under project No. GXWD20201230155427003-20200730101839009, and supported, in part, by Amazon under an Amazon Research Award in automated reasoning; by the United States National Science Foundation (NSF) under grants No. 1917924 and No. 2114627; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. 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.

REFERENCES

[1] aldur. The return value at line 295., 2021. https://github.com/aldur/MatasanoCrypto/blob/master/matasano/blocks.py.
[2] 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.
[3] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. *SIGPLAN Not.*, 46(1):459–472, January 2011.
[4] 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.
[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] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. Refinym: Using names to refine types. 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 107–117, New York, NY, USA, 2018. Association for Computing Machinery.
[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] Python Software Foundation. Official documentation of Mypy, 2020. https://mypy.readthedocs.io/en/stable/builtin_types.html.

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

[10] Python Software Foundation. Cpython. python's official implementation, 2021. https://github.com/python/cpython.

[11] 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.

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

[13] gengo. The return value at line 853., 2021. https://github.com/gengo/memsource-wrap/blob/master/memsource/api.py.

[14] Loukas Georgiadis, Robert Tarjan, and Renato Werneck. Finding dominators in practice. volume 10, pages 69–94, 01 2006.

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

[16] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19, 10 2013.

[17] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. Maxsmt-based type inference for python 3. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 12–19, Cham, 2018. Springer International Publishing.

[18] 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.

[19] Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. Static type inference for foreign functions of python. In *32nd International Symposium on Software Reliability Engineering*, October 2021.

[20] IntelLabs. The return value at line 95. https://github.com/IntelLabs/coach/blob/master/rl_coach/memories/non_episodic/experience_replay.py, 2021.

[21] 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.

[22] 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.

[23] Jetbrains. Python developer survey conducted by jetbrains and python software foundation, 2020. https://www.jetbrains.com/lp/python-developers-survey-2020/.

[24] Bingyi Kang, Saining Xie, Marcus Rohrbach, Zhicheng Yan, Albert Gordo, Jiashi Feng, and Yannis Kalantidis. Decoupling representation and classifier for long-tailed recognition, 2020.

[25] C. M. Khaled Saifullah, M. Asaduzzaman, and C. K. Roy. Exploring type inference techniques of dynamically typed languages. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 70–80, 2020.

[26] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *2nd LLVM Workshop on the LLVM Compiler Infrastructure in HPC*.

[27] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Comput. Surv.*, 53(3), June 2020.

[28] 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/.

[29] 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/.

[30] Jialun Liu, Yifan Sun, Chuchu Han, Zhaopeng Dou, and Wenhui Li. Deep representation learning on long-tailed data: A learnable embedding augmentation perspective, 2020.

[31] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 304–315. IEEE Press, 2019.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[33] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013.

[34] Amir M. Mir, Evaldas Latoskinas, and Georgios Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. *CoRR*, abs/2104.04706, 2021.

[35] Amir M Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Deep similarity learning-based type inference for python. *arXiv preprint arXiv:2101.04470*, 2021.

[36] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic type inference for gradual hindley-milner typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, 2019.

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

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

[39] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. *TypeWriter: Neural Type Prediction with Search-Based Validation*, page 209–220. Association for Computing Machinery, New York, NY, USA, 2020.

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

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

[42] Pysonar2. https://github.com/yinwang0/pysonar2.

[43] Python. The python ast module, 2021. https://github.com/python/cpython/blob/3.9/Lib/ast.py.

[44] Python. The typeshed project, 2021. https://github.com/python/typeshed.

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

[46] Vikas Raunak, Siddharth Dalmia, Vivek Gupta, and Florian Metze. On long-tailed phenomena in neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3088–3095, Online, November 2020. Association for Computational Linguistics.

[47] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, September 2017.

[48] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *SIGPLAN Not.*, 50(1):111–124, January 2015.

[49] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1565–1572, New York, NY, USA, 2013. Association for Computing Machinery.

[50] Jiawei Ren, Cunjun Yu, Shunan Sheng, Xiao Ma, Haiyu Zhao, Shuai Yi, and Hongsheng Li. Balanced meta-softmax for long-tailed visual recognition, 2020.

[51] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery.

[52] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1646–1657. IEEE, 2021.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.

[54] sernst. The return value at line 35., 2021. https://github.com/sernst/cauldron/blob/master/cauldron/steptest/functional.py.

[55] tomdean. The return value at line 56., 2021. https://github.com/tomdean/growser/blob/master/growser/handlers/rankings.py.

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

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

[58] wtsi hgi. The return value at line 151., 2021. https://github.com/wtsi-hgi/metadata-check/blob/master/mcheck/metadata/seqscape_metadata/seqscape_metadata.py.

[59] 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.

[60] Ningyu Zhang, Shumin Deng, Zhanlin Sun, Guanying Wang, Xi Chen, Wei Zhang, and Huajun Chen. Long-tail relation extraction via knowledge graph embeddings and graph convolution networks. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3016–3025, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

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

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