QuAC: Quick Attribute-Centric Type Inference for Python

JIFENG WU, University of British Columbia, Canada
CAROLINE LEMIEUX, University of British Columbia, Canada

Python's dynamic typing facilitates rapid prototyping and underlies its popularity in many domains. However, dynamic typing reduces the power of many static checking and bug-finding tools. Python type annotations can make these tools more useful. *Type inference tools* aim to reduce developers' burden of adding them. However, existing type inference tools struggle to support dynamic features, infer correct types (especially container type parameters and non-builtin types), and run in reasonable time. Inspired by Python's duck typing, where the attributes accessed on Python expressions characterize their implicit interfaces, we propose QuAC (Quick Attribute-Centric Type Inference for Python). At its core, QuAC collects attribute sets for Python expressions and leverages information retrieval techniques to predict classes from these attribute sets. It also recursively predicts container type parameters. We evaluate QuAC's performance on popular Python projects. Compared to state-of-the-art non-LLM baselines, QuAC predicts types with high accuracy complementary to those predicted by the baselines while not sacrificing coverage. It also demonstrates clear advantages in predicting container type parameters and non-builtin types and reduces run times. Furthermore, QuAC is nearly two orders of magnitude faster than an LLM-based method while covering nearly half of its errorless non-trivial type predictions. It is also significantly more consistent at predicting container type parameters and non-builtin types than the LLM-based method, regardless of whether the project has ground-truth type annotations.

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

Additional Key Words and Phrases: Python, Type Inference, Gradual Typing, Static Analysis

ACM Reference Format:
Jifeng Wu and Caroline Lemieux. 2024. QuAC: Quick Attribute-Centric Type Inference for Python. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 343 (October 2024), 30 pages. https://doi.org/10.1145/3689783

1 Introduction

According to analyses from GitHub Octoverse [GitHub 2023] and IEEE Spectrum [IEEE Spectrum 2023], Python is one of the most favored programming languages since 2018, surpassing stalwarts such as Java and C/C++. Unlike these languages, Python is dynamically typed, facilitating rapid prototyping and making it particularly attractive in diverse fields, including data science, web development, and IoT. However, as Python has become increasingly pervasive, the disadvantages of dynamic typing have become more salient. Amongst other things, static types enable more meaningful static analyses, in-IDE error checks, and refactoring passes. Thus, in 2014, PEP 484 [van Rossum et al. 2014] introduced a standard syntax for Python type annotations. These type annotations are not checked by Python itself, but are used by IDEs, linters, and static type checkers—such as mypy [mypy Developers 2024] and Pytype [Google 2024]—to find errors before code runs.

Despite the advantages of static typing, only a very small proportion of Python code is annotated. A 2020 study of Python types in the wild [Rak-amnouykit et al. 2020] found that six years after

Authors' Contact Information: Jifeng Wu, University of British Columbia, Vancouver, Canada, jifengwu2k@gmail.com; Caroline Lemieux, University of British Columbia, Vancouver, Canada, clemieux@cs.ubc.ca.

ACM 2475-1421/2024/10-ART343
https://doi.org/10.1145/3689783

introducing PEP 484, only 2,678 of 70,000 analyzed repositories had type annotations. Further, on average, 1,144 repositories have less than 1 type annotation per file. This conflict—the clear advantages of type annotations but the apparent reluctance of developers to add them to Python code—has led to the development of several *type inference* tools that aim to reduce developers' burden of adding type annotations by automatically annotating untyped Python files. A recent study of the utility of type inference tools finds that they can reduce the time it takes to annotate Python code with type annotations by 40% [Guo et al. 2024].

We believe a good Python type inference tool should satisfy, at the very least, two core criteria. First, its predictions should be *correct*. For untyped Python code, there may not be a ground truth, and in this case, the predicted types should at least be *correct modulo type checker* [Allamanis et al. 2020; Yee and Guha 2023], i.e., raise no type checking errors. Second, the type inference tool should output *as many type predictions as possible*, i.e., achieve high *coverage* of the code being analyzed. A type inference tool that only gives suggestions for 20 of 1000 typing slots has limited utility.

The landscape of type inference for Python (and other dynamically typed languages) is defined by a contrast between traditional static type inference methods and emerging machine learning-based methods. Static type inference methods [Cannon 2005; Google 2024; Hassan et al. 2018; Maia et al. 2012; Meta 2024; Microsoft 2024; Salib 2004; Sun et al. 2022; Vitousek et al. 2014; Wang 2022] utilize rule-based approaches, data-flow analysis, and heuristics to create and solve typing constraints. They aim for correctness and achieve high accuracy with simple types in straightforward contexts. However, they often only support a subset of their target languages [Anderson et al. 2005; Chandra et al. 2016] and can struggle with dynamic features [Richards et al. 2010], affecting their *coverage*. Moreover, the computational effort required to generate and solve their constraints can limit their usage in large-scale codebases. Conversely, machine learning-based approaches [Allamanis et al. 2020; Dash et al. 2018; Hellendoorn et al. 2018; Peng et al. 2022; Pradel et al. 2020; Wei et al. 2023; Xu et al. 2016; Yan et al. 2023] use natural language cues and context with various machine learning models (e.g., sequence models, graph models) to improve coverage and accuracy in type inference. These methods can handle the complexities of dynamic languages and provide multiple candidate types, enhancing inference flexibility. However, they cannot guarantee type correctness and struggle with rare types [Mir et al. 2021]. Despite recent advances in hybrid models that statically validate type predictions [Allamanis et al. 2020; Peng et al. 2022; Pradel et al. 2020; Yan et al. 2023], their validation processes can only eliminate invalid types suggested by machine learning models without correcting them, leading to potential drops in coverage. Furthermore, Large Language Model (LLM)-based techniques [Wei et al. 2023] present immense requirements for various resources such as computation and energy [Chien et al. 2023; Šakota et al. 2024; Samsi et al. 2023]. Moreover, even with extensive code datasets for pre-training these models, it is still difficult in practice to fully cover the code distribution. This results in out-of-distribution (OOD) generalization challenges and unpredictable model inference behaviors [Hajipour et al. 2024]. Finding a balance between correctness, coverage, and performance remains challenging.

We believe Python's duck typing presents new opportunities for type inference. From its inception, Python has endorsed *duck typing* [Milojkovic et al. 2017]: the *attributes* (fields, methods) accessed on expressions *implicitly define interfaces* that valid types should implement. If the type of an expression satisfies that interface (i.e., "quacks like a duck"), the program should run fine.

For example, consider the code fragment in Listing 1, which defines a `Point` type and a global function `maximize`. The parameter `points` of the global function `maximize` can be any type providing the method `__getitem__` for indexing on Line 6 and for slicing on Line 7. Furthermore, given the `for`-loop on Line 7 iterating over `points[1:]`, the type of `points[1:]` (and thus of `points`) should also provide the method `__iter__` supporting iteration. Thus, `points` could be a `list`, a

Listing 1. `maximize` is an example of a duck-typed Python function. Adapted from the bm_float benchmark in the Python Benchmark Suite [Python Core Developers 2024]

```
1 class Point(object): # A 3D Point Class
2   def __init__(self, i): ...
3   def maximize(self, other): ... # Sets values to the max in each dimension
4
5 def maximize(points): # Return the maximal point for a set of 3D points
6   elem = points[0]
7   for p in points[1:]:
8     elem = elem.maximize(p)
9   return elem
```

`tuple`, an `array.array`[1], or any other *sequence type*. Python's `typing` module defines an interface that covers all such types: `Sequence`. Note that providing the *attribute set* {`__getitem__`, `__iter__`} is necessary for the type of `points`. Moreover, the element accessed through indexing on Line 6, `elem`, calls its method `maximize` on Line 8. If `Point` is the only type providing this attribute in the context of Listing 1, then `elem` should be annotated with the type `Point`. Again, note that providing the *attribute set* {`maximize`} is necessary for the type of `elem`.

We find that existing state-of-the-art approaches struggle with this example. The industrial static type inference tool Pytype [Google 2024], which aims for soundness, does not make predictions for the parameter `points` of the global function `maximize` and predicts its return value to be the trivial `Any`. The academic static type inference tool Stray [Sun et al. 2022] also fails to predict types for the parameter `points` and the return value. The machine learning-based type inference technique incorporating a static validation process HiTyper [Peng et al. 2022] predicts the type of `points` to be `tuple`, which is technically correct but is *over-constrained* (`points` could also be some other sequence type such as `list`) and does not predict `tuple` 's type parameters (the type of the elements `points` contains). Furthermore, HiTyper erroneously predicts the return value of the global function `maximize` to be `str`. Similarly, the recent large language model (LLM)-based type inference method TypeT5 [Wei et al. 2023], though making the correct prediction, `Point`, for the global function `maximize`'s return value, also makes an over-constrained prediction lacking type parameters, `list`, for the parameter `points`.

Observe, above, that the *attribute set* accessed on a Python expression characterizes the expression's *implicit interface* that a valid type must provide. We believe *finding the simplest types that satisfy this attribute set* may be a robust, high-coverage way of conducting type inference.

Based on this intuition, we propose QuAC (Quick Attribute-Centric Type Inference for Python). QuAC combines simple static analysis techniques with information retrieval techniques to try and find a balance between correctness, coverage, and performance. QuAC desugars Python's syntactic constructs into attribute accesses and collects attribute sets like {`__getitem__`, `__iter__`} for the parameter `points` and {`maximize`} for the return value of the global function `maximize` in Listing 1. For built-in functions whose implementations are not available in Python, QuAC leverages extra type information from Typeshed [Typeshed Contributors 2024]. Then, it queries classes implementing the given attribute set. Considering that rare attributes are more suggestive of specific classes, QuAC uses BM25 queries, a standard information retrieval technique [Robertson et al. 2009]. Additionally, QuAC recursively applies its attribute collection and class querying technique to predict container type parameters. For example, QuAC can successfully predict the types of the parameter `points` and the return value of the global function `maximize` in Listing 1 to be `Sequence[Point]` and `Point`, respectively.

[1]The class `array` in Python standard library's `array` module.

To evaluate QuAC's performance, we compare QuAC to state-of-the-art non-LLM approaches Stray [Sun et al. 2022] and HiTyper [Peng et al. 2022]. We chose these as examples of static and machine learning techniques capable of predicting both parameters and return values found to outperform other approaches in their evaluations. Considering that a major goal of Python type inference is to predict types for Python code *without type annotations* and facilitate migrating untyped Python codebases to typed ones, we evaluate QuAC and the baselines on a set of popular Python projects including *untyped* ones. Inspired by a similar evaluation of TypeScript type inference methods for migrating untyped JavaScript codebases [Yee and Guha 2023], we evaluate the correctness of type predictions by running mypy [mypy Developers 2024] on each typing slot individually. We evaluate the coverage by counting the number of non-trivial (i.e., not `None` or `Any`) predictions. Further, we compare QuAC's ability to predict container type parameters and non-builtin types against Stray and HiTyper and compare their run times on benchmarks of different sizes. In addition, we also compare QuAC against TypeT5 [Wei et al. 2023], a recent LLM-based approach that fine-tunes CodeT5 [Wang et al. 2021], a pretrained LLM for code.

In total over all benchmarks, QuAC achieves type prediction correctness higher than Stray and HiTyper while retaining a competitive type prediction coverage. Moreover, it can predict container type parameters with high correctness and coverage. By analyzing the typing slots where Stray, HiTyper, and QuAC predict correct types, we find that QuAC excels on typing slots where a non-builtin type is correct, overcoming the rare types issue faced by machine learning-based type inference methods. Its typing slots with correct type predictions, in general, complement Stray and HiTyper, suggesting its potential to be used in an ensemble type inference method. Moreover, QuAC is substantially faster than Stray and HiTyper. Further, compared to the LLM-based method TypeT5 [Wei et al. 2023], QuAC covers a significant share of TypeT5's errorless non-trivial type predictions (47.8% on average) with much greater efficiency (92× faster on average) and retains significant advantages in predicting container type parameters and non-builtin types, delivering consistent results regardless of whether the benchmarks had type annotations in training data.

Overall, we make the following contributions:

- We introduce QuAC (ref. Section 4), a type inference method that collects attribute sets for Python expressions, employs information retrieval methods for class prediction, and recursively predicts container type parameters.
- We implement QuAC for Python (ref. Section 5) and distribute its implementation as open source on Zenodo [Wu and Lemieux 2024].
- We evaluate QuAC and non-LLM baseline techniques on a set of popular Python projects (ref. Section 6), demonstrating QuAC's advantages in overall accuracy, container type parameters, non-builtin types, and run times, while not sacrificing coverage.
- We compare QuAC to an LLM-based technique TypeT5 [Wei et al. 2023] on the same Python projects and provide a contrastive evaluation of TypeT5's performance on both untyped and typed benchmarks. We find QuAC is nearly two orders of magnitude faster while covering nearly half of TypeT5's errorless non-trivial type predictions and has significantly more consistent performance in predicting container type parameters and non-builtin types.

2 High-Level Overview

We provide a high-level overview of QuAC in this section. QuAC works by translating Python's expressions and statements to attribute accesses and collecting *attribute sets* for Python expressions, including the parameters and return values of functions. It also populates a *class query database* including concrete classes available under the given typing context and *protocols* (abstract base classes) in the Python standard library. Afterward, QuAC queries its database for the most likely class (Section 4.2.4) and recursively predicts type parameters for containers (Section 4.3). We illustrate QuAC's type prediction process with the example in Listing 2.

Listing 2. Motivating example adapted from the fasta Python 3 #3 program in *The Computer Language Benchmarks Game* [The Computer Language Benchmarks Game Team 2023].

```
1  import bisect
2
3  def make_cumulative(table):
4    P = []; C = []; prob = 0.
5    for char, p in table:
6      prob += p; P += [prob]; C += [ord(char)]
7    return (P, C)
8
9  def random_fasta(table, n, seed):
10   width = 60; im = 139968.0
11   # ...
12   if n % width: ...
13   probs, chars = make_cumulative(table)
14   count = 0.0; end = (n / float(width)) - count_modifier
15   while count < end:
16     for i in range(width):
17       seed = (seed * 3877.0 + 29573.0) % 139968.0
18       line[i] = chars[bisect.bisect(probs, seed / im)]
```

Predicting Basic Types. In Listing 2, the parameter `n` of `random_fasta` is involved in a modulo operation with an `int` (`n % width` on Line 12) and is divided by a `float` (`n / float(width)` on Line 14). This requires that `n` has the attributes `__mod__` and `__truediv__`. To retrieve a type for `n`, QuAC queries its class database (as defined in Section 4.2.4) with the attribute set {`__mod__`, `__truediv__`}. The query returns the numeric protocol `numbers.Real` (whose concrete subclasses include `int` and `float`) as the highest-ranked type. So, QuAC predicts `n`'s type annotation as `numbers.Real`. Similarly, the parameter `seed` of `random_fasta` has the attribute set {`__mul__`,`__truediv__`} from the operations `seed * 3877.0` on Line 17 and `seed / im` on Line 18. From this, QuAC, following the same querying procedure as before, predicts `seed`'s annotation as `numbers.Real`.

Predicting Container Type Parameters. On Line 5 of the function `make_cumulative`, we iterate over the parameter `table`. This means that `table` must support the method `__iter__`. Given the attribute set {`__iter__`}, QuAC predicts `table`'s type as `Iterable[T]`, where `T` is a *type parameter* representing the type of the items iterated over it. To predict `T`, we recursively invoke QuAC to predict the type of the *iteration target* of `table`. In the for-loop on Line 5, the iteration target is the 2-tuple `char,p`. QuAC predicts its type to be `tuple`. Finishing the prediction requires recursively calling QuAC to predict types for the first (`char`) and second (`p`) elements of the 2-tuple.

We observe that `char` is passed to the built-in function `ord`, which, from a Typeshed lookup, accepts a one-character `str` and returns an `int`. Thus, QuAC populates `char`'s attributes with the attributes of `str`, and predicts `char`'s type as `str`. Given `prob = 0.`, we know `prob` is an instance of type `float`. Given `prob += p`, QuAC populates `p`'s attribute set with the attributes of `prob`. This leads QuAC to predict `p`'s type as `float`. Linking these together, QuAC predicts `char,p` as `tuple[str,float]`. With this type for `T`, the prediction is complete: QuAC predicts the parameter `table` of `make_cumulative` to be `Iterable[tuple[str,float]]`.

Related Expression Propagation. In some code, the attributes accessed on an expression are sparse. In these situations, it may be possible to populate their attribute sets with those of *related expressions*. For example, when predicting the type of the parameter `table` of `random_fasta`, we observe that it is not operated on except to be passed to the parameter `table` of `make_cumulative`. In this case, it is meaningful to perform an *interprocedural analysis* and adopt the attributes and information about type parameters from `table` in `make_cumulative`.

Listing 3. Examples of Python Type Annotations

```
def greeting(name: str) -> str:
  return 'Hello ' + nam

def f(x: typing.Any) -> None:
  y = x.foo(); z = y.bar()
```

After propagating the information, QuAC knows that the parameter `table` of `random_fasta` should have the attribute `__iter__` and its *iteration target* is the 2-tuple `char,p` on Line 5. Following the logic above, QuAC predicts its type to also be `Iterable[tuple[str,float]]`. This demonstrates the utility of augmenting the attribute sets and information about type parameters of expressions with those of *interprocedurally related expressions*. We also augment expression typing constraints with those of *intraprocedural* related expressions in assignment statements, arithmetic and logical operations, and comparisons, as hinted above and detailed in Section 4.2.2.

3 Background

This section provides additional background on topics necessary to precisely define QuAC: readers familiar with these concepts may skip directly to Section 4 for the details of QuAC.

3.1 Python Type Annotations

PEP 484 [van Rossum et al. 2014] brought optional type annotations to Python 3.5 for enhanced code completion in IDEs, static analysis, refactoring, and code generation.

In its simplest form, Python type annotations denote *classes* for function parameters and return values. For instance, the function `greeting` in Listing 3 expects the parameter `name` and the return value to be of class `str`. Beyond classes, Python type annotations also allow a variety of other constructs. The singleton `None` indicates that a parameter or return value is expected to be this singleton object. Similarly, the singleton `Any` represents a dynamically typed value of an arbitrary type. In Listing 3, function `f` accepts a parameter `x` of any type and returns the singleton object `None`—the default behavior for Python functions without an explicit return statement.

Furthermore, a category of classes, known as generic classes, permits *parameterization*. For instance, `dict[int,str]` represents a `dict` with keys of type `int` and values of type `str`. Different generic classes follow different parameterization syntax and semantics. For example, `list[str]` denotes a list containing strings, while `tuple[int,int,str]` signifies a 3-tuple containing two integers and a string.

Over time, Python's type annotation framework has been enriched through a series of PEPs, including union types, literal types denoting that a variable's value must correspond to one of the specified literals, and annotated types which add context-specific metadata (such as the value range of a variable) to an annotation. QuAC aims to predict stable and frequently used types for type annotations: classes (including primitives such as `int`), and parameterized standard library containers.

3.2 Special Methods

Python uses objects as its primary data abstraction method. Each object has a *class*, which can define *special methods* [Python Software Foundation 2020] (also known as *magic methods* or *dunder methods*) invoked by Python operators. For example, a class implementing the `__getitem__` method enables its instances to use the indexing notation (`x[i]`), while the methods `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__` are invoked by the binary arithmetic operations `+`, `-`, `*`, `/`, `//`. Conversely, the presence of Python operators in source code also implies the existence of relevant special methods in the classes of their operands. These special methods are an important constitutive part of the attribute sets we collect for expressions to predict their classes.

3.3 Typeshed

Typeshed [Typeshed Contributors 2024] is an officially maintained repository of stub files for the Python standard library. A stub file outlines the public interface (classes, variables, and functions) of a Python module and contains type annotations. It adheres to Python syntax but replaces variable initializers, function bodies, and default arguments with ellipsis expressions. Moreover, stub files may contain circular imports, cannot be imported as Python modules, and have to be manually parsed using Python's `ast` module. We use Typeshed stub files in our project to determine the attribute requirements of parameters and return values of functions within the Python standard library. As much of the Python standard library is written in C, such information would be difficult to acquire without analysis of non-Python code.

4 Method

4.1 Overview

Algorithm 1 shows the overall QuAC algorithm. QuAC begins its analysis on a set of Python AST module nodes $\mathcal{M} = \{m_1, \ldots, m_n\}$. First, QuAC conducts pre-analysis to collect base information for type prediction. It constructs C, a database of *candidate classes* (Section 4.2.4). Then it generates $\mathcal{D}$, a map of name nodes in the AST to their definition nodes, following Python's scoping rules. The last part of the pre-analysis is creating $\boldsymbol{\rho}$, which maps each function definition to a *symbolic return value* to handle regular and async functions and generators. After this pre-analysis is done, it collects *typing constraints* (Line 4, Algorithm 1), and predicts types for all functions in the input modules (Line 5, Algorithm 1) based on these typing constraints. The meat of QuAC's analysis lies in collecting the typing constraints.

Algorithm 1: Overall Algorithm

Data: Module nodes $\mathcal{M} = \{m_1, \ldots, m_n\}$
Result: Type predictions $\mathcal{P}$
1 $C \leftarrow$ construct a database of candidate classes;
2 $\mathcal{D} \leftarrow$ map each name to its definition via Python name resolution;
3 $\boldsymbol{\rho} \leftarrow$ map each function definition to a symbolic return value;
4 $\mathcal{A}, \mathcal{G}_r, \mathcal{G}_S \leftarrow$ CollectTypingConstraints$(\mathcal{M}, \mathcal{D}, \boldsymbol{\rho})$;
5 $\mathcal{P} \leftarrow$ TypePrediction$(\mathcal{M}, \boldsymbol{\rho}, \mathcal{A}, \mathcal{G}_r, \mathcal{G}_S)$;
6 **return** $\mathcal{P}$

Algorithm 2: Type Prediction

Data: Module nodes $\mathcal{M}$, Candidate classes C, Function definition to symbolic return value mapping $\boldsymbol{\rho}$, Node to attributes mapping $\mathcal{A}$, Directed graphs storing node relations and typing constraint subsets $\mathcal{G}_r, \mathcal{G}_S$
Result: Type predictions $\mathcal{P}$
`// E is a set of expression nodes`
Function PredictTypeOfExprSet$(E, C, \mathcal{A}, \mathcal{G}_r, \mathcal{G}_S)$**:**
  $E' \leftarrow \{e' | e' \in \mathcal{G}_S, \exists e \in E \text{ s.t. } e' \text{ reachable in } \mathcal{G}_S \text{ from } e\}$;
  $a \leftarrow \bigcup_{e' \in E'} \mathcal{A}[e']$; $c \leftarrow$ **if** $a \neq \emptyset$ **then** $\arg\max_{c' \in C} \text{score}(c', a)$ **else** Any; $\mathcal{T} \leftarrow [\,]$;
  for $\mathcal{R} \in$ GetRelationSetsOfTypeParameters(c) **do**
    $E'' \leftarrow \{e'' | e' \in E', r \in \mathcal{R}, (e', e'', r) \in \mathcal{G}_r\}$;
    $t' \leftarrow$ PredictTypeOfExprSet$(E'', C, \mathcal{A}, \mathcal{G}_r, \mathcal{G}_S)$; $\mathcal{T}.add(t')$;
  return if $\mathcal{T} \neq [\,]$ **then** Parameterize$(c, \mathcal{T})$ **else** c;
`// Main algorithm`
$\mathcal{P} \leftarrow \emptyset$;
foreach $m \in \mathcal{M}$ **do**
  foreach $f = \texttt{def}\ g(x_1, \ldots, x_n) \in \mathcal{M}$ **do**
    `// Predict type of each function parameter and return value`
    foreach $e \in \{x_1, \ldots, x_n, \boldsymbol{\rho}[g]\}$ **do** $\mathcal{P}[e] \leftarrow$ PredictTypeOfExprSet$(\{e\}, C, \mathcal{A}, \mathcal{G}_r, \mathcal{G}_S)$;
return $\mathcal{P}$

QuAC's *typing constraints* are collected in three main data structures. The first is $\mathcal{A}$, a mapping from AST expression nodes to attributes (Section 4.2.1). $\mathcal{A}$ is the intuitive base behind QuAC, answering the question: which attributes are accessed on each AST expression node? To predict container type parameters, QuAC also builds $\mathcal{G}_r$ during its analysis. $\mathcal{G}_r$ a directed graph storing *relations* among nodes, i.e., tracking which AST nodes correspond to the key of a dictionary node (Section 4.3.1). Finally, QuAC builds $\mathcal{G}_S$, which links different AST nodes whose types should be compatible. In particular, an edge (e_1, e_2) in $\mathcal{G}_S$ means e_2's typing constraints are a *subset* of e_1's (Section 4.2.2). A formal description of how we collect these typing constraints is presented in Algorithm 3. We explain the type constraint collection in detail in the next sections.

With these typing constraints, QuAC runs Algorithm 2 to predict types. In Lines 9-12 of Algorithm 2, QuAC loops over all function definitions in the input Python modules (Line 10), and runs PredictTypeOfExprSet (Line 11) to predict the type for each function parameter and return value. Given a set of expression nodes E to predict types for, PredictTypeOfExprSet first *collects* the set of nodes E' whose *typing constraints* are a subset of those in E, by finding the nodes reachable from any $e \in E$ in $\mathcal{G}_S$ (Line 2). On Line 3, it then *merges* the *attribute sets* (Section 4.2.1) of all nodes in E' with a simple union, and predicts a class c based on the merged attribute set (Section 4.2.4). If c is a *generic container* (e.g., `dict`), QuAC predicts its *type parameters*; otherwise, it returns c as the type prediction. Specifically, for each type parameter of a generic container, QuAC obtains its *relation set* $\mathcal{R}$. Then, QuAC identifies the set of nodes E'' capturing the usages of that type parameter using $\mathcal{G}_r$ (Line 5, Section 4.3.1) and runs PredictTypeOfExprSet recursively on E'' to predict the parameter type t' (Line 6). Then, it *parameterizes* the predicted generic container with the predicted parameter types to derive the final type prediction result (Line 7). This recursive algorithm allows QuAC to predict non-parametric types (e.g., `int`) and arbitrary nested parametric types (e.g., `dict[str,list[list[int]]]`) in a unified manner.

4.2 Predicting Classes

The main step in our type prediction procedure is predicting what *class* an expression is. To do this, we assign each Python expression an *attribute set* representing the attributes in an unknown class. We populate $\mathcal{A}$, which maps each expression to its attribute set, by *collecting attributes based on syntactic constructs.* The attribute sets of some expressions are *subsets* of others, as expressed by edges in $\mathcal{G}_S$. Then, we *query classes* based on these attribute sets.

4.2.1 Collecting Attributes. We perform attribute collection by walking the nodes of the target Python AST in evaluation order, and adding attributes to $\mathcal{A}$ in a *syntax-directed* manner. We add attributes directly accessed on expressions (e.g., `x.y` accesses the attribute `y` on variable `x`), and *special methods* (Section 3.2) accessed internally by the Python interpreter through syntactical constructs. For example, the indexing expression `x[y]` requires that `x` supports indexing via the `__getitem__` method. The `with` statement `with x as y` requires that `x` is a *context manager* providing the `__enter__` and `__exit__` methods. Python's Language Reference [Python Software Foundation 2020] lists the special methods each Python expression and statement implies. Algorithm 3 shows which attributes are added to $\mathcal{A}$ for different syntax constructs.

4.2.2 Typing Constraint Subsets. Some expressions have no attribute accesses in a function body. In such cases, it may be possible to populate their attribute sets through other expressions whose attribute sets we assume to be *subsets* of these expressions'. We store this information as a *typing constraint subset graph* $\mathcal{G}_S$. $\mathcal{G}_S$ is a directed graph where an edge (e_1, e_2) indicates that e_2's attribute set is a subset of e_1's, i.e., $\mathcal{A}[e_2] \subseteq \mathcal{A}[e_1]$. This happens at:

Assignments. Given $x = y$, we consider $(x, y) \in \mathcal{G}_S$ (Lines 48 and 38, Algorithm 3).

Function return values. A Python function may have multiple return statements and return a *generator* or *coroutine*. To handle these complexities, we introduce a *symbolic return value* ρ for each function. For ordinary functions, we consider ρ's attribute set a *superset* of the attribute sets of expressions returned at different return statements (Line 47, Algorithm 3). For functions returning a generator g or coroutine c, we initialize ρ's attribute set with the attribute set of g or c, and add *relations* (Section 4.3.1) between ρ and returned, yielded, and awaited expressions to predict the type parameters of g or c (Line 41,45, Algorithm 3).

Algorithm 3: Collecting Typing Constraints

Input: Module nodes $\mathcal{M}$, Name to definition mapping $\mathcal{D}$, Function definition to symbolic return value mapping $\boldsymbol{\rho}$
Result: $\mathcal{N}$ is all AST nodes in $\mathcal{M}$. $\mathcal{A} : \mathcal{N} \to$ *attribute sets*; $\mathcal{G}_r : \mathcal{N} \times \mathcal{N} \times$ relation type; $\mathcal{G}_S : \mathcal{N} \times \mathcal{N}$

1 $\mathcal{RT} \leftarrow \emptyset$;
2 $\mathit{SetInstance}(n, T) \equiv \mathcal{RT}[n].\mathit{add}(\{\texttt{Instance}(T)\}); \mathcal{A}[n].\mathit{add}(\texttt{attrs}(T))$;
3 **foreach** $m \in \mathcal{M}$ **do**
4   populate $\mathcal{RT}$ with names corresponding to modules, classes, global functions, and instances in m;
5   **foreach** $n \in$ visit child nodes of m in evaluation order **do**
6   **switch** n **do**
7   **case** Name **do** $d \leftarrow \mathcal{D}[n]; \mathcal{RT}[n].\mathit{add}(\mathcal{RT}[d]); \mathcal{G}_S.\mathit{add}(\{(n, d), (d, n)\})$;
8   **case** num, str, bytes **do** $\mathit{SetInstance}(n, \texttt{typeof}(n))$;
9   **case** $[e_1, \ldots, e_n], (e_1, \ldots, e_n), \{e_1, \ldots, e_n\}, \{k_1 : v_1, \ldots, k_n : v_n\}$ **do**
10                 $\mathit{SetInstance}(n, \texttt{typeof}(n))$;
11   Add to $\mathcal{G}_r$ $\{(n, e_i, \{\texttt{ValueOf}, \texttt{IterTargetOf}\}) | i \in [1, n]\}$ (list), $\{(n, e_i, \{\texttt{Element i Of}\}) | i \in [1, n]\}$ (tuple), $\{(n, e_i, \{\texttt{IterTargetOf}\}) | i \in [1, n]\}$ (set), $\{(n, k_i, \{\texttt{KeyOf}, \texttt{IterTargetOf}\}), (n, v_i, \{\texttt{ValueOf}\}) | i \in [1, n]\}$ (dict);
12   **case** unary e **do** $\mathcal{A}[e].\mathit{add}(\{\texttt{attr}(\text{unary})\}); \mathcal{G}_S.\mathit{add}(\{(n, e), (e, n)\})$;
13   **case** e_1 binop e_2 **do**
14                 $\mathcal{A}[e_1].\mathit{add}(\{\texttt{attr}(\text{binop})\})$;
15   **if** binop $\neq *$ **then**
16                     $\mathcal{G}_S.\mathit{add}(\{(n, e_1), (e_1, n)\})$;
17   **if** binop $\neq$ % **then** $\mathcal{G}_S.\mathit{add}(\{(e_1, e_2), (e_2, e_1)\})$;
18   **case** e_1 cmpop e_2 **do**
19   **if** cmpop $\in \{==, !=, <, <=, >, >=\}$ **then** $\mathcal{A}[e_1].\mathit{add}(\{\texttt{attr}(\text{cmpop})\}); \mathcal{G}_S.\mathit{add}(\{(e_1, e_2), (e_2, e_1)\})$;
20   **if** cmpop $\in \{\texttt{in}, \texttt{not in}\}$ **then**
21                     $\mathcal{A}[e_2].\mathit{add}(\{\texttt{\_\_contains\_\_}, \texttt{\_\_iter\_\_}\}); \mathcal{G}_r.\mathit{add}((e_2, e_1, \{\texttt{IterTargetOf}\}))$;
22   **case** $e(e_1, \ldots, e_n)$ **do**
23   **case** $f = \texttt{def}\ g(x_1, \ldots, x_n) \in \mathcal{RT}[e]$ **do**
24                     $\mathcal{G}_S.\mathit{add}(\{(e_i, x_i) | i \in [1, n]\} \cup \{(n, \boldsymbol{\rho}[g])\})$;
25   **if** g is in the Python standard library **then** populate $\mathcal{RT}[n]$ via Typeshed lookup;
26   **case** $m = \texttt{InstanceMethod}(\texttt{def}\ g(\text{self}, x_1, \ldots, x_n)) \in \mathcal{RT}[e]$ **do**
27                     $\mathcal{G}_S.\mathit{add}(\{(e_i, x_i) | i \in [1, n]\} \cup \{(n, \boldsymbol{\rho}[g])\})$;
28   **if** g is in the Python standard library **then** populate $\mathcal{RT}[n]$ via Typeshed lookup;
29   **case** $c = \texttt{class}\ C \in \mathcal{RT}[e]$ **do**
30                     $\texttt{def}\ g(\text{self}, x_1, \ldots, x_n) \leftarrow$ constructor of c; $\mathcal{G}_S.\mathit{add}(\{(e_i, x_i) | i \in [1, n]\}); \mathit{SetInstance}(n, c)$;
31                 $\mathcal{A}[e].\mathit{add}(\{\texttt{\_\_call\_\_}\}); \mathcal{G}_r.\mathit{add}(\{(e, e_i, \{\texttt{Parameter i Of}\}) | i \in [1, n]\} \cup \{(e, n, \{\texttt{ReturnValueOf}\})\})$;
32   **case** $e_1[e_2]$ **do**
33                 $\mathcal{A}[e_1].\mathit{add}(\{\texttt{\_\_getitem\_\_}\})$;
34   **if** $\texttt{typeof}(e_2) \in \{\texttt{slice}, \texttt{tuple}\}$ **then** $\mathcal{G}_S.\mathit{add}(\{(n, e_1), (e_1, n)\})$;
35   **else** $\mathcal{G}_r.\mathit{add}(\{(e_1, e_2, \{\texttt{KeyOf}\}), (e_1, n, \{\texttt{ValueOf}\})\})$;
36   **case** $e.x$ **do** $\mathcal{A}[e].\mathit{add}(\{x\}); \mathcal{RT}[n].\mathit{add}(\{R.x | R \in \mathcal{RT}[e]\})$;
37   **case** $\texttt{def}\ f(x_1 = e_1, \ldots, x_n = e_n) : \ldots$ **do**
38                 $\mathcal{G}_S.\mathit{add}(\{(x_i, e_i) | i \in [1, n]\}); \rho \leftarrow \boldsymbol{\rho}[n]$;
39                 $\mathcal{G}_r.\mathit{add}(\{(n, x_i, \{\texttt{Parameter i Of}\}) | i \in [1, n]\} \cup \{(n, \rho, \{\texttt{ReturnValueOf}\})\})$;
40                 $\mathbf{r}, \mathbf{s}, \mathbf{y} \leftarrow$ nodes returned from, sent to, yielded from f;
41   **if** f is a generator **then**
42                     $\mathcal{G}_r.\mathit{add}(\{(\rho, y, \{\texttt{IterTargetOf}\}) | y \in \mathbf{y}\} \cup \{(\rho, s, \{\texttt{SendTargetOf}\}) | s \in \mathbf{s}\} \cup \{(\rho, r, \{\texttt{YieldFromAwaitTargetOf}\}) | r \in \mathbf{r}\})$;
43                     $\mathit{SetInstance}(\rho,$ **if** f is async **then** Generator **else** AsyncGenerator$)$;
44   **else**
45   **if** f is async **then**
46                         $\mathcal{G}_r.\mathit{add}(\{(\rho, r, \{\texttt{YieldFromAwaitTargetOf}\}) | r \in \mathbf{r}\}); \mathit{SetInstance}(\rho, \texttt{Coroutine})$;
47   **else** **if** $\mathbf{r} \neq \emptyset$ **then** $\mathcal{G}_S.\mathit{add}(\{(\rho, r) | r \in \mathbf{r}\})$ **else** $\mathit{SetInstance}(\rho, \texttt{None})$;
48   **case** $e_1 = e_2$ **do** $\mathcal{G}_S.\mathit{add}(\{(e_1, e_2)\}); \mathcal{RT}[e_1].\mathit{add}(\mathcal{RT}[e_2])$;
49   **case** for e_1 in e_2 **do** $\mathcal{A}[e_2].\mathit{add}(\{\texttt{\_\_iter\_\_}\}); \mathcal{G}_r.\mathit{add}(\{(e_2, e_1, \{\texttt{IterTargetOf}\})\})$;
50   **case** with e_1 as e_2 **do** $\mathcal{A}[e_1].\mathit{add}(\{\texttt{\_\_enter\_\_}, \texttt{\_\_exit\_\_}\})$;
51   **case** yield from e **do** $\mathcal{A}[e].\mathit{add}(\{\texttt{\_\_iter\_\_}\}); \mathcal{G}_r.\mathit{add}(\{(e, n, \{\texttt{YieldFromAwaitTargetOf}\})\})$;
52   **case** await e **do** $\mathcal{A}[e].\mathit{add}(\{\texttt{\_\_await\_\_}\}); \mathcal{G}_r.\mathit{add}(\{(e, n, \{\texttt{YieldFromAwaitTargetOf}\})\})$;
53 **return** $\mathcal{A}, \mathcal{G}_r, \mathcal{G}_S$;

Function calls. If we can precisely determine which function is called at a call site (Section 4.2.3), we consider the attribute sets of *parameters* within the function definition to be *subsets* of those of *arguments* passed at the call site, and the attribute sets of *symbolic return values* to be *subsets* of those of function call results. For example, given a call site $\mathtt{f}(y_1, y_2)$ and function definition $\mathtt{def\ f}(x_1, x_2)$ with a symbolic return value ρ, then $(y_1, x_1), (y_2, x_2), (\mathtt{f}(y_1, y_2), \rho) \in \mathcal{G}_S$. This process is formalized in Line 22 of Algorithm 3.

Two expressions may also have their attribute sets to be mutual subsets (i.e., $(e_1, e_2), (e_2, e_1) \in \mathcal{G}_S$) when they are involved in the following syntactical constructs:

- The operands and results of *arithmetic and logical operations* (except `*`, which allows multiplying sequences and integers, and `%`, which allows formatting strings, Lines 12 and 13, Algorithm 3).
- The left and right hand sides of *comparisons* (Line 19, Algorithm 3).
- An expression that is *sliced* (indexed by a `slice` or `tuple` object, e.g., `y[1:10]`, `X[1:10, :5]`), and the result of slicing (Line 34, Algorithm 3).
- Accessing a previously defined name based on name resolution (Line 7, Algorithm 3).

4.2.3 Resolving Function Calls. If we can resolve function calls, we can propagate attribute set relationships between function parameters and arguments, and between symbolic return values and values returned from function calls. To resolve function calls, we associate a *run-time term set* $\mathcal{RT}[e]$ with each Python expression e. Based on these run-time term sets, we can then resolve most calls either to user-defined code or the Python standard library. Run-time terms include *modules*, *classes*, *functions*, *instances*, and *methods*.[2]

$\mathcal{RT}$ is populated in Algorithm 3 in sync with the other typing constraints. To build $\mathcal{RT}$, we first populate the run-time terms of names corresponding to *modules, classes, functions, and instances* in each module (Line 4, Algorithm 3). Then, we populate the run-time terms of *derivative expressions* resulting from the following rules:

R-1. Accessing modules, classes, functions, and instances on modules, functions on classes, functions and methods on instances (Line 36, Algorithm 3).
R-2. Initializing an instance explicitly via literal notation or calling a class (Lines 8, 9, 29, Algorithm 3), or implicitly via returning a generator or coroutine instance or `None` from a regular function with no explicit return value (Lines 41, 45, 47, Algorithm 3).
R-3. Calling a function or method in the Python standard library results in an instance determined via a Typeshed (Section 3.3) lookup (Lines 25, 28, Algorithm 3).
R-4. Copying run-time terms via an assignment or by accessing a previously defined name (Lines 7, 48, Algorithm 3).

Example. Fig. 1 shows sample code (left), and the run-time term sets (right) QuAC populates for different expressions in the code. We walk through QuAC's run-time term collection procedure in this example. From `import re as r`, we add Python's `re` module as a run-time term for `r`. Then, we apply the above rules to populate the run-time terms of derivative expressions:

(R-1) We add the function `re.compile` as a run-time term for `r.compile`.
(R-3) Typeshed says `re.compile`'s return value is an instance of `re.Pattern`: we add this as a run-time term for `r.compile(pattern)` (and `regex`, R-4).
(R-1) We add the method `re.Pattern.match` as a run-time term for `regex.match`.
(R-3) Typeshed says `re.Pattern.match` returns an instance of `None` or `re.Match`: we add both as run-time terms for `regex.match(characters, pos)` (and `match`, R-4).
(R-1) We add the instance method `re.Match.group` as a run-time term for `match.group`.

[2]Methods include bound instance methods and class methods, while unbound methods are considered to be functions.

```python
import re as r
def lex(characters, token_exprs):
  pos = 0; tokens = []
  while pos < len(characters):
    match = None
    for token_expr in token_exprs:
      pattern, tag = token_expr
      regex = r.compile(pattern)
      match = regex.match(characters, pos)
      if match:
        text = match.group(0)
```

Expression	**Run-time Term Set**
`r`	module `re`
`r.compile`	global function `re.compile`
`regex`	instance of `re.Pattern`
`regex.match`	instance method `re.Pattern.match`
`match`	`None`, instance of `re.Match`
`match.group`	instance method `re.Match.group`
`text`	instance of `str`, instance of `bytes`

Fig. 1. To resolve calls, QuAC finds the run-time terms associated with each Python expression. The table on the right gives the run-time term sets QuAC populates for Python expressions in the code listing.

(R-3) Typeshed says `re.Match.group` returns an instance of `str` or `bytes`: we add both as run-time terms for `match.group(0)` (and `text`, R-4).

Through this procedure, we determine what is being called at a large number of call sites. If the *called function or method*[3] is *user-defined*, we add subset relations between the attribute sets of parameters/arguments and symbolic return values/call results (Section 4.2.1). If the called function or method is from the Python standard library, QuAC initializes *dummy parameters and symbolic return values* for the callable and initializes their attribute sets by looking up Typeshed (Section 3.3), before adding subset relations as for user-defined callables.

4.2.4 Querying Classes. The last step in class prediction is querying classes (Line 3, Algorithm 2) for a given attribute set. We first construct C in Algorithm 1, a set of *candidate classes*:

- Built-in classes such as `int`, `str`, and `list`.
- *Protocols* [van Rossum et al. 2018] (abstract base classes) in the Python standard library, such as `Iterable` representing any object supporting iteration and `Callable` representing any object that can be called. These are useful when the attribute requirements of an expression point to an *interface requirement* (e.g., supporting iteration) rather than concrete classes satisfying that interface requirement (`list`, etc.) This is a common situation given Python's duck typing.
- User-defined classes within the Python files being analyzed and classes within imported external modules (both Python standard library and third-party).

From this database, we query candidate classes using the Okapi BM25 ranking function [Robertson et al. 2009]. Given an attribute set $a = \{\alpha_1, \ldots, \alpha_n\}$, the BM25 score of a class $c \in C$ is:

$$\text{score}(c, a) = \sum_{i=1}^{n} \text{IDF}(\alpha_i) \cdot \frac{f(\alpha_i, c) \cdot (k_1 + 1)}{f(\alpha_i, c) + k_1 \cdot (1 - b + b \cdot \frac{|c|}{avgcl})} \tag{1}$$

where $f(\alpha_i, c)$ is the number of times[4] α_i occurs in c, $|c|$ is the length of c in attributes, and $avgcl$ is the average class length in the class query database. k_1 and b are free parameters. Based on the guidelines in [Manning et al. 2008], we use $k_1 = 1.50$ and $b = 0.75$. $\text{IDF}(\alpha_i)$ is the IDF (inverse document frequency) weight of the attribute α_i. It captures the *rarity* of the attribute, or how much *information* the attribute provides [Robertson 2004]. Given $C = \{c_1, c_2, \ldots c_n\}$, the IDF of α_i is:

$$\text{IDF}(\alpha_i) = \ln\left(\frac{|C| - n(\alpha_i) + 0.5}{n(\alpha_i) + 0.5} + 1\right) \tag{2}$$

[3]Calling a class boils down to calling its constructor while calling an instance boils down to calling its `__call__` method.
[4]As Python classes do not include duplicate attributes, this is either 1 if the attribute is present, or 0 if it is absent.

where $n(\alpha_i) = |\{c | c \in C, \alpha_i \in c\}|$ is the number of classes containing α_i.

The rationale for using IDF weighting is that not all attributes are equal in class prediction, with rare attributes more suggestive of specific classes. For example, `object` is at the top of Python's class inheritance hierarchy, and every class in Python has `object`'s attributes. Thus, those attributes cannot be used to discern classes. In contrast, `str` is the only built-in class providing the attribute encode. Thus, when only considering built-in classes, the attribute encode within an attribute set suggests str due to the attribute's high IDF weight.

4.3 Predicting Type Parameters for Containers

The procedure above allows us to predict classes. However, in addition to classes themselves, *generic classes* (e.g. `dict`) parameterized by *type parameters* (e.g. `K`, `V` in `dict[K,V]` assigned `K=str`, `V=int` in `dict[str,int]`, Section 3.1) are pervasive in Python. Through a quantitative analysis of the 2083 type annotations present in the ten most popular *typed* pure-Python packages [Libraries.io 2023], we found that 1036 (49.74%) contained *parameterized generic classes*. Due to their ubiquity, especially for denoting *container element types* [van Rossum et al. 2014], predicting type parameters for generic classes such as containers is essential for usability.

However, this is challenging in an unconstrained setting. In Python, type parameters can be used *anywhere* in generic class definitions, including in the type annotations of fields and method parameters and return values. If the types of all variables are known beforehand, it is easy to infer and check the types of type parameters based on the usage of fields and methods. This is what *type checkers* do, given existing type annotations and soundly inferred types.

However, in *type prediction* on untyped codebases, the types of a large number of variables are *not known a priori* and *cannot be soundly inferred*. In this case, accurately predicting the type parameters of one expression's predicted class entails accurately predicting the types of *related expressions* associated with those type parameters. But that set of related expressions—the ones representing the use of a type parameter—cannot be determined before the base class is predicted! For instance, consider the statements `a = x[y]; a += 1`. If `x` is a `dict`, this tells us `x`'s *second* type parameter should be an `int`, say `dict[?,int]`. On the other hand, if `x` is a `list`, this gives us information about its *first* type parameter (`list[int]`). Further, `x` could be some user-defined class that extends `list[int]` but does not contain type parameters itself.

However, compared with arbitrary type parameters, a large portion of type parameters are used in *containers*, the designated use case of generics in PEP 484. Specifically, within the 1558 parameterized generic classes in the type annotations of the ten most popular typed pure-Python packages mentioned above, 1114 (71.50%) were parameterizations of *containers*, including concrete classes such as `list` and `dict`, and protocols such as `Iterable` and `Callable`.[5] Although generics were designed to express "type information about objects kept in containers that cannot be statically inferred generically" in PEP 484, many container type parameters have semantics corresponding to specific *syntactical constructs* in Python code. For example, given that `y : list[T]`, both `y[i]` (for `i: int`) and the `x` in `for x in y` have types equivalent to the type variable `T`. We exploit this to predict container type parameters in a *syntax-directed* manner.

4.3.1 Modeling Container Type Parameter Semantics. Based on the insight above, we model the *semantics* of container type parameters using *relations*. For example, in `dict[K,V]`, the type parameter `K` has the type of the *keys* and *iteration targets* of the dictionary, while `V` has the type of the *values* of the dictionary. We represent K's and V's semantics with the *relation sets* $\mathcal{R}(K) = \{$`KeyOf`, `IterTargetOf`$\}$ and $\mathcal{R}(V) = \{$`ValueOf`$\}$, respectively. A complete description of

[5]The top five remaining parameterized non-container generic classes were 9.76% `Optional` for optional types, 6.80% `Union` for union types, 5.32% `Type` for class objects, 1.60% `IO` for IO streams, and 1.60% `Literal` for literal types.

all relations can be found in Section 4.3.2 below. For each standard library container,[6] we have specified its number of type parameters and the *relation sets* for each type parameter. QuAC retrieves these in the call to GetRelationSetsOfTypeParameters on Line 4, Algorithm 2.

When analyzing the code, we associate (potential) container expressions with *semantically related* expressions based on *syntax-directed*, *type-agnostic* association rules for each relation. We store these associations in the directed graph $\mathcal{G}_r$. As an example, given $e_3 = e_1[e_2]$ in source code, $(e_1, e_2, \{\texttt{KeyOf}\}), (e_1, e_3, \{\texttt{ValueOf}\}) \in \mathcal{G}_r$, even if the types of e_1, e_2, and e_3 are unknown.

We also extend the notion of our typing constraint subsets (Section 4.2.2) to include node relations. For instance, if $(e_1, e_2) \in \mathcal{G}_S$ and $(e_2, e_2^{'}, \mathcal{R}) \in \mathcal{G}_r$, we assume e_1 is also related to $e_2^{'}$ via the relations in $\mathcal{R}$. In Algorithm 2, given a set of expression nodes E, which is extended to E' (Line 2), we may predict c to be a container class. In this case, for the relation set $\mathcal{R}$ of a type parameter of c, we can obtain E'', the set of all expressions associated with the nodes in E' via the relations in $\mathcal{R}$ (Line 5). This allows us to recursively predict the type of the type parameter (Line 6).

4.3.2 Supported Relations. For each relation, we describe its semantics and provide example containers whose type parameters have this relation. We also describe when we associate a potential container expression e with another expression e' via the relation, with reference to Algorithm 3.

- `KeyOf`, `ValueOf`. A type parameter has `KeyOf` or `ValueOf` if it is the type of the *indexing expression* or *indexed result*, respectively, in non-slicing indexing operations. For example, `ValueOf` $\in \mathcal{R}$(`T`) for `list[T]`, `KeyOf` $\in \mathcal{R}$(`K`), `ValueOf` $\in \mathcal{R}$(`V`) for `dict[K,V]`. In Algorithm 3, we make associations for `list` and `dict` construction (Line 9) and non-slicing indexing operations (Line 35).
- `IterTargetOf`. A type parameter has `IterTargetOf` if it is the type of the *iteration target* of an instance of that container: given the for-loop `for x in y`, `x` is the *iteration target* of `y`. For example, `IterTargetOf` $\in \mathcal{R}$(`T`) for `list[T]`, `IterTargetOf` $\in \mathcal{R}$(`K`) for `dict[K,V]`, and `IterTargetOf` $\in \mathcal{R}$(`Y`) for `Generator[Y,S,R]`. In Algorithm 3, we make associations for `list`, `set`, and `dict` construction (Line 9), container membership checks (Line 20), values yielded from generators (Line 42), and `for`-loops (Line 49).
- `Element i Of`. In Python, tuples are immutable and usually contain *heterogeneous* elements. Reflecting this usage pattern, tuples are frequently constructed using the *literal notation* (e.g., `(a, b, c)`). Python's type annotation for tuples requires specifying the type of each tuple element — an n-tuple with elements of types $T_1, \ldots, T_n$ has the type `tuple`$[T_1, \ldots, T_n]$, where `Element i Of` $\in \mathcal{R}(T_i)$. In Algorithm 3, we make associations for `tuple` construction (Line 9).
- `Parameter i Of`, `ReturnValueOf`. Python allows annotating simple *callable objects* (no variadic arguments, keyword-only parameters) using `Callable`. Specifically, an object called with n positional parameters of types $T_1, \ldots, T_n$ and returning a value of type T_r can be annotated as `Callable`$[[T_1, \ldots, T_n], T_r]$, where `Parameter i Of` $\in \mathcal{R}(T_i)$, `ReturnValueOf` $\in \mathcal{R}(T_r)$. In Algorithm 3, we make associations for calls (Line 31) and function definitions (Line 39).
- `SendTargetOf`. PEP 342 [Ewing and van Rossum 2005] allows values to be sent to generators, which then become the results of `yield` expressions within the generator. `S` of `Generator[Y,S,R]` captures the type of these values, i.e., `SendTargetOf` $\in \mathcal{R}(S)$. In Algorithm 3, we make associations for values sent to generators (Line 42).
- `YieldFromAwaitTargetOf`. PEP 380 [Ewing 2009] allowed one generator to delegate part of its operations to *another* through the `yield from` expression. Later on, PEP 492 [Selivanov 2015] introduced *coroutines* to Python, allowing one coroutine to obtain the result of *another* through the `await` expression. In both cases, a value in one of the `return` statements of the *second* generator or coroutine is assigned to the `yield from` or `await` expression of the *first*. R in `Generator[Y,S,R]`

[6] This includes typical containers (`list`, `dict`, etc.) as well as protocols such as `Callable` and `Generator`, which are not strictly containers but are parameterized types.

or `Coroutine[Y,S,R]` represents the type of this value, i.e., `YieldFromAwaitTargetOf` $\in \mathcal{R}(R)$. In Algorithm 3, we make associations for values "returned from" generators and coroutines (Lines 42, 45) and `yield from` and `await` expressions (Lines 51, 52).

5 Implementation

QuAC is implemented in around 9k lines of Python code. Its core component is a Python AST visitor that walks all statements and expressions to collect attributes (Section 4.2.1) and resolves function calls (Section 4.2.3). We support all statements and expressions defined in Python 3.9 [Python Software Foundation 2020]. Moreover, to resolve imports in the Python files being analyzed and to add classes within imported standard library and third-party modules to the class query database (Section 4.2.4), we also let the Python interpreter import the Python files being analyzed as *modules* and use Python's live object introspection capabilities. To build the class query database, we store all candidate classes and their attributes in a *document-term matrix* [Anandarajan et al. 2019], which we implement our class query BM25 ranking function on top of. We also implement a Typeshed lookup library based on `typeshed_client` [Zijlstra 2024] that parses the relevant Typeshed type stubs on demand whenever a Typeshed lookup is required (ref. Section 4.2.3). We used the provided reproduction packages to run the non-LLM baseline methods Stray [Sun et al. 2022] and HiTyper [Peng et al. 2022] and the LLM-based type inference method TypeT5 [Wei et al. 2023]. We ran all methods within a Docker container on Ubuntu 20.04. The system has an Intel(R) Core(TM) i7-12700K CPU (@3.6GHz) with 64GB RAM. Stray, QuAC, and TypeT5 produce varying results from run to run due to selecting different top-ranking type annotations. The run times of the methods also vary from run to run. Thus, we have run each method five times and averaged the results from each run to reduce variability. The code, benchmarks, and data replication scripts are available on Zenodo [Wu and Lemieux 2024].

6 Evaluation

6.1 Research Questions

We investigate the following questions in our evaluation. RQs (1) and (2) measure our core criterion of coverage and accuracy for type inference tools. RQs (3) and (4) evaluate QuAC's ability to predict container type parameters and non-builtin types. RQ (5) evaluates QuAC's run-time performance. RQs (6) and (7) evaluate whether QuAC and the non-LLM baselines display similar or distinct patterns in making type predictions. RQ (8) evaluates how predictions would differ from human-written annotations on annotated projects. Finally, RQ (9) investigates whether QuAC still maintains competitiveness in the era of LLM-based type inference methods.

(1) What coverage can QuAC achieve?
(2) What accuracy can QuAC achieve?
(3) How well does QuAC predict container type parameters?
(4) How well does QuAC predict non-builtin types?
(5) What is the run-time performace of QuAC?
(6) Do QuAC and the non-LLM baselines make correct type predictions for the same or different typing slots?
(7) What are the main failure modes of QuAC?
(8) How well does QuAC match existing type annotations on typed benchmarks?
(9) How does QuAC compare to an LLM-based method?

6.2 Baselines and Benchmarks

We evaluate QuAC against the state-of-the-art non-LLM methods Stray [Sun et al. 2022] (a static method) and HiTyper [Peng et al. 2022] (a hybrid ML method) using popular pure-Python projects [Libraries.io 2023] with greatly varying project sizes as benchmarks. We believe they are representative of real-world Python projects. Table 1 describes key statistics of the benchmarks. We evaluate RQs (1)-(6) on untyped and typed benchmarks, RQ (7) on only the untyped benchmarks, and RQ (8) on only the typed benchmarks. We compare QuAC to the recent LLM-based type inference method TypeT5 [Wei et al. 2023] on untyped and typed benchmarks in RQ (9).

Table 1. Statistics of the benchmarks used in our evaluation.

	Repository	Version	Lines of Code	Typing Slots	GitHub Stars	Dependent Packages
Untyped	requests	2.31.0	5963	861	51K	60.2K
	Pygments	2.15.1	104475	2135	1.5K	3.66K
	boto3	1.28.10	7625	1319	8.61K	7.02K
	gunicorn	21.2.0	6279	893	9.38K	1.31K
	python-dateutil	2.8.2	15277	2133	1.93K	6.14K
	pytz	2023.3	4961	374	297	6.36K
	six	1.16.0	755	93	949	14.2K
	pytest-cov	4.1.0	1358	228	1.64K	14.8K
	notebook	7.0.0	306	30	11K	1.08K
	peewee	3.16.2	6352	2083	10.6K	532
	seaborn	0.12.2	25616	3436	11.6K	5.42K
Typed	click	8.1.6	7465	1208	15.1K	23.7K
	flake8	4.0.1	4431	558	3.34K	11.9K
	Flask	2.3.2	6412	840	66.9K	8.22K
	ipython	8.14.0	50979	5845	16.2K	5.7K
	Jinja2	3.1.2	10993	1859	9.97K	10.7K
	pre_commit	3.3.3	5038	833	12.3K	0
	pylint	2.17.4	38784	4552	5.18K	5.05K
	sphinx	7.1.0	52021	9644	6.15K	82
	urllib3	2.0.4	7009	969	3.7K	10.3K
	Werkzeug	2.3.6	17774	2432	6.56K	2.37K

6.3 Evaluation Criteria

Previous work on type inference for Python [Allamanis et al. 2020; Mir et al. 2022; Peng et al. 2022] have evaluated their methods on Python projects *with* type annotations, using two main criteria for correctness. First, Exact Match: a type prediction completely matches an existing type annotation. Second, Match to Parametric: a type prediction completely matches an existing type annotation *when ignoring all type parameters* (i.e., `list[int]` and `list[str]`).

However, these may be *too strict* for Python's duck typing philosophy. For example, a value passed to the parameter `params` in Listing 4 need not exactly be `dict[str,bool]`: it could be any "dict-like" type whose `items` method returns a `Iterable[tuple[str,bool]]`. Thus, `Mapping[str,bool]`, which parameterizes the protocol `Mapping` for "dict-like" classes, would be a perfectly valid type prediction. However, this type prediction would be *incorrect* based on the criteria above.

Listing 4. Example of too-strict annotation inspired by method `keyword_arguments_for` of class FileProcessor in module flake8.processor; some code simplified and some elided for brevity.

```
def keyword_args_for(params: dict[str,bool], args: ...) -> ...:
  for param, required in params.items():
    args[param] = getattr(self, param)
```

Typilus [Allamanis et al. 2020] also proposed a third criterion, Type Neutral. Type Neutral means that a type prediction is correct if replacing the ground truth with it does not yield a type error.

Typilus *approximates* type neutrality by building a type hierarchy for the types in its training corpus, assuming universal covariance of type parameters. In this approximation, they say a prediction is Type Neutral if the predicted type is a non-`object` supertype of the type annotation. This approximation is not robust as a non-`object` supertype may not provide all the attributes being accessed on an expression and its derived expressions. For example, while `Mapping[str,bool]` is a correct type prediction for `params` under this approximation, so would `Mapping[object,object]` and `Container[object]`. The former is wrong since it suggests that the type of its keys—`param` in Listing 4—is `object`. However, given the usage `getattr(self, param)`, `param` must be of the more specific type `str`. The latter, `Container[object]`, is wrong as it doesn't provide the `items` method called on `params`.

More importantly, the Exact Match, Match to Parametric, and Type Neutral metrics all require Python projects to have existing type annotations, but a major goal for type inference for Python is to predict types for *previously unannotated projects*. Thus, we need a metric to assess the correctness of type predictions that is based on how expressions are actually used within the project, and would work even if type annotations are unavailable.

To achieve this goal, in addition to using the Exact Match and Match to Parametric metrics on typed benchmarks, we also adapt the Correctness Modulo Type Checker metric proposed in Typilus [Allamanis et al. 2020] and used in a recent evaluation of type inference methods for TypeScript [Yee and Guha 2023], on both untyped and typed benchmarks. This approach delegates the task of checking type predictions to type checkers, whose best effort has been demonstrated to be reasonably effective in practice [Gao et al. 2017]. Specifically, we use mypy [mypy Developers 2024], which introduced optional typing into Python and strongly inspired Python's type annotation syntax. This can be seen as an alternative implementation of the Type Neutral metric in Typilus that does not require ground truth type annotations.

6.4 Results

We run QuAC and the baselines on the benchmarks in Table 1. The results are as follows.

6.4.1 What coverage can QuAC achieve? To investigate QuAC's ability to predict type annotations, we analyze the number of typing slots with non-trivial (not `None` or `Any`) type predictions, as presented in the "# Type Preds." column in Table 2. We see that Stray lags behind both HiTyper and QuAC regarding the total number of non-trivial type predictions, indicating Stray's relative ineffectiveness in achieving high coverage. On the other hand, QuAC and HiTyper make a comparable number of non-trivial type predictions on all benchmarks, with QuAC having a significant edge on some benchmarks (`peewee`, `seaborn`, `Werkzeug`). On `peewee`, HiTyper fails to generate a type dependency graph, leading to no predictions for this benchmark. These results show that despite having a relatively simple design, QuAC is robust and on par with a non-LLM ML method at achieving high coverage. In fact, in terms of *total errorless* non-trivial type predictions across our benchmarks, QuAC exceeds HiTyper. On average, over all benchmarks, QuAC adds *errorless* non-trivial types to 34% of the typing slots in Table 1, compared to 28% by HiTyper.

6.4.2 What accuracy can QuAC achieve? We then investigate the correctness of these non-trivial type predictions by examining the percentages of them that are correct via Correctness Modulo Type Checker, as presented in the "% Errorless" column in Table 2. Although QuAC does not have a clear advantage over HiTyper in the total number of non-trivial type predictions it makes, it does consistently achieve a higher (or at least equal) errorless percentage on all benchmarks, as well as the *highest* errorless percentage on all but three benchmarks (`gunicorn`, `seaborn`, `Jinja2`) where Stray is higher. However, on these benchmarks, QuAC achieves 3.2×, 15.8×, 2.7× more errorless non-trivial type predictions than Stray. These results suggest that QuAC's design focuses

Table 2. The total number of non-trivial (i.e., not `None` or `Any`) type predictions by each technique on each benchmark. *% Errorless* is the percent of those predictions on which mypy raises no errors (— means divide by zero).

	# Type Preds.			% Errorless		
Repository	S	H	Q	S	H	Q
requests	0	**334**	283	—	83.8	**85.1**
Pygments	31	1127	**1133**	87.1	72.1	**90.5**
boto3	249	**565**	396	89.8	72.4	**91.5**
gunicorn	104	**387**	350	**88.5**	76.5	83.7
python-dateutil	0	363	**526**	—	81.3	**86.3**
pytz	6	**119**	102	66.7	81.5	**86.3**
six	0	0	**26**	—	—	**92.3**
pytest-cov	0	**40**	38	—	67.5	**94.7**
notebook	0	**19**	7	—	**100**	**100**
peewee	0	0	**726**	—	—	**92.0**
seaborn	101	1199	**1738**	**94.1**	80.4	86.5
click	0	**694**	603	—	84.3	**92.4**
flake8	109	210	**226**	80.9	71.9	**84.5**
Flask	40	226	**305**	76.5	82.7	**87.7**
ipython	0	2188	**2367**	—	80.4	**88.5**
Jinja2	310	816	**886**	**92.8**	81.0	86.9
pre_commit	0	**584**	453	—	75.5	**85.0**
pylint	0	1992	**2294**	—	64.1	**82.8**
sphinx	0	474	**3755**	—	95.2	**99.1**
urllib3	0	415	**451**	—	83.6	**89.4**
Werkzeug	0	626	**1190**	—	86.3	**90.8**

Table 3. Total number of container type predictions with non-trivial type parameters (i.e., `list[int]` rather than `list`). *% Errorless* is the percent of those predictions on which mypy raises no errors (— means divide by zero).

	# Param'd. Preds.			% Errorless		
Repository	S	H	Q	S	H	Q
requests	0	22	**34**	—	59.1	**67.4**
Pygments	5	205	**395**	**100**	26.8	91.1
boto3	29	**51**	45	**84.8**	72.6	80.4
gunicorn	4	**37**	33	50.0	48.7	**78.8**
python-dateutil	0	32	**63**	—	50.0	**77.8**
pytz	0	4	**12**	—	0.0	**83.3**
six	0	0	**3**	—	—	**100**
pytest-cov	0	**6**	5	—	33.3	**100**
notebook	0	**4**	**4**	—	**100**	**100**
peewee	0	0	**71**	—	—	**85.9**
seaborn	8	150	**309**	**87.5**	70.7	75.1
click	0	51	**69**	—	62.8	**89.9**
flake8	29	**39**	38	62.8	61.5	**79.0**
Flask	6	17	**30**	**76.7**	52.9	56.7
ipython	0	224	**314**	—	63.0	**82.7**
Jinja2	20	83	**89**	80.0	53.0	**81.7**
pre_commit	0	77	**82**	—	67.5	**81.7**
pylint	0	204	**265**	—	48.5	**80.6**
sphinx	0	29	**667**	—	96.6	**99.1**
urllib3	0	16	**39**	—	25.0	**84.6**
Werkzeug	0	42	**147**	—	52.4	**83.7**

on accuracy and, compared to Stray and HiTyper, predicts type annotations with higher overall accuracy while not sacrificing the absolute number of predictions.

6.4.3 How well does QuAC predict container type parameters? Recall QuAC has special handling for containers: recursively predicting their type parameters (ref. Section 4.3). We evaluate QuAC on this front by recording the (1) total number of container types with non-trivial type parameters predicted, and (2) the percentage of those which are errorless in Table 3.

Regarding the total number of predicted containers with non-trivial type parameters, QuAC and HiTyper outperform Stray on all benchmarks. QuAC further outperforms HiTyper on all but four benchmarks. Out of these predictions, QuAC's are most likely to be errorless, exceeding HiTyper on all benchmarks. Stray achieves slightly higher correctness than QuAC on four benchmarks, but QuAC obtains higher coverage on these. This data suggests that QuAC's approach to predicting container type parameters is more effective than Stray and HiTyper.

6.4.4 How well does QuAC predict non-builtin types? Besides container type parameters, we also study QuAC's trends in predicting *non-builtin types*. By *builtin types*, we mean standard types built into the interpreter and usable anywhere without the need for imports, such as `int`, `list`, and `str`. Investigating such a research question is meaningful as static type inference methods may prioritize builtin types [Sun et al. 2022]. Further, predicting non-builtin types is also one of the bottlenecks of machine learning-based type inference methods. This is because each non-builtin type tends to have low occurrence frequencies in their training sets, yet all such rare non-builtin types account for a significant amount of annotations [Peng et al. 2022].

Table 4 shows the percentage of errorless non-trivial type predictions that are non-builtin types, as well as the number of errorless non-builtin type predictions. Compared with Stray and HiTyper, QuAC has a higher percentage of correct type predictions that are non-builtin on all benchmarks. QuAC also has a higher absolute number of correct non-builtin type predictions on all but two benchmarks. This is in contrast with Stray and HiTyper, which fail to generate *any* errorless

Table 4. Percent (left) and total number of (right) errorless type predictions that are non-builtin types. — means divide by zero.

	% Preds. non-builtin			# Non-builtin preds.		
Repository	S	H	Q	S	H	Q
requests	—	12.5	**45.2**	0	51	**109**
Pygments	6.06	8.5	**59.9**	2	85	**614**
boto3	6.9	10.1	**64.4**	22	62	**233**
gunicorn	7.3	8.6	**42.0**	10	42	**123**
python-dateutil	—	5.1	**46.0**	0	67	**209**
pytz	11.1	29.5	**55.7**	3	**54**	49
six	—	—	**29.2**	0	0	**7**
pytest-cov	—	16.4	**69.4**	0	9	**25**
notebook	—	7.4	**14.3**	0	**2**	1
peewee	—	—	**56.6**	0	0	**378**
seaborn	9.3	19.0	**41.4**	10	274	**623**
click	—	21.5	**34.7**	0	178	**193**
flake8	7.4	11.3	**38.7**	10	30	**74**
Flask	11.7	23.2	**49.5**	5	72	**132**
ipython	0.0	12.1	**31.4**	0	428	**658**
Jinja2	11.4	20.2	**55.9**	40	183	**430**
pre_commit	—	9.2	**45.5**	0	50	**175**
pylint	—	18.6	**64.4**	0	412	**1222**
sphinx	—	20.8	**52.6**	0	126	**1956**
urllib3	—	8.1	**29.5**	0	51	**119**
Werkzeug	—	13.3	**30.3**	0	119	**327**

Table 5. Run times of each technique in seconds.

	Run Time (s)		
Repository	S	H	Q
requests	80.7	48.5	**10.4**
Pygments	2,606.1	365.4	**62.9**
boto3	28,534.6	79.2	**7.8**
gunicorn	193.3	81.5	**22.9**
python-dateutil	270.9	154.8	**19.6**
pytz	38.1	24.3	**2.8**
six	6.6	1.9	**1.5**
pytest-cov	38.2	9.9	**2.0**
notebook	13.3	9.4	**1.6**
peewee	3.2	**2.1**	20.2
seaborn	5,400.0	259.4	**193.9**
click	75.4	62.3	**12.9**
flake8	1025.1	52.7	**7.2**
Flask	944.4	56.3	**11.5**
ipython	3179.7	625.2	**229.9**
Jinja2	15166.2	81.5	**23.8**
pre_commit	255.7	138.7	**13.1**
pylint	1860.8	511.3	**206.3**
sphinx	2603.5	3892.9	**426.7**
urllib3	148.0	76.9	**14.8**
Werkzeug	352.6	138.5	**51.9**

non-builtin type predictions on several benchmarks. Overall, the results demonstrate QuAC's propensity towards predicting correct non-builtin types, suggesting QuAC does not face the same low-frequency non-builtin type bottleneck that many baseline techniques have.

6.4.5 What is the run-time performace of QuAC?. Table 5 presents the run times of each method on each benchmark. We can see that QuAC outperforms HiTyper and Stray on all but one benchmark and achieves geometric mean speedups of 3×, 14× over HiTyper and Stray on the untyped benchmarks (4×, 18× on all benchmarks). On the benchmark where QuAC is slower, `peewee`, QuAC takes 20 seconds to run and predicts 726 non-trivial types; Stray and HiTyper run in 2-3 seconds but predict no non-trivial types. Overall, QuAC's design makes it more scalable in terms of project size compared with Stray and HiTyper.

6.4.6 Do QuAC and the non-LLM baselines make correct type predictions for the same or different typing slots? Continuing on this note, we look at whether Stray, HiTyper, and QuAC make correct type predictions for the *same* or *different* typing slots. We analyzed the sizes of the sets of errorless non-trivial typing slots for each method and each combination of methods over all benchmarks, as shown in Table 6. Table 6 tells a story of there being little overlap between the typing slots where Stray, HiTyper, and QuAC make errorless, non-trivial type predictions. This suggests that QuAC, in general, makes accurate predictions on typing slots *distinct* from Stray and HiTyper. Following the last research question, this difference may be partly driven by QuAC excelling at typing slots where a non-builtin type prediction is correct. Overall, these results suggest it is worthwhile to include QuAC in an *ensemble* complementing other type inference methods.

6.4.7 What are the main failure modes of QuAC?. We now investigate the main failure modes of QuAC and other non-LLM methods. We present failure modes appearing more than once within the five most error-prone typing slots for each method and each benchmark in Table 7.

Predictions Lacking Accessed Attributes. One of the main failure modes of Stray and HiTyper is the inability to reject type predictions that do not provide accessed attributes. For example, the parameter `request` of `requests.cookies.MockRequest`'s constructor (depicted below) is assigned

Table 6. Sizes of the sets of errorless non-trivial typing slots for each method and each combination of methods. The total number of typing slots for each benchmark is in Table 1.

Repository	S	H	Q	S,H (IoU)	S,Q (IoU)	H,Q (IoU)	S,H,Q
requests	0	**280**	241	0 (0.00)	0 (0.00)	169 (0.48)	0
Pygments	27	813	**1026**	25 (0.03)	15 (0.01)	485 (0.36)	15
boto3	224	**409**	362	122 (0.24)	64 (0.12)	183 (0.31)	49
gunicorn	92	**296**	293	34 (0.10)	57 (0.17)	160 (0.37)	21
python-dateutil	0	295	**454**	0 (0.00)	0 (0.00)	172 (0.30)	0
pytz	4	**97**	88	2 (0.02)	4 (0.05)	59 (0.47)	2
six	0	0	**24**	0 (0.00)	0 (0.00)	0 (0.00)	0
pytest-cov	0	27	**36**	0 (0.00)	0 (0.00)	14 (0.29)	0
notebook	0	**19**	7	0 (0.00)	0 (0.00)	6 (0.30)	0
peewee	0	0	**668**	0 (0.00)	0 (0.00)	0 (0.00)	0
seaborn	95	964	**1503**	73 (0.07)	59 (0.04)	620 (0.34)	44
click	0	**585**	557	0 (0.00)	0 (0.00)	334 (0.41)	0
flake8	88	151	**191**	36 (0.18)	48 (0.21)	71 (0.26)	19
Flask	31	187	**267**	9 (0.04)	20 (0.07)	105 (0.30)	8
ipython	0	1758	**2094**	0 (0.00)	0 (0.00)	1063 (0.38)	0
Jinja2	288	661	**770**	183 (0.24)	125 (0.13)	374 (0.35)	92
pre_commit	0	**441**	385	0 (0.00)	0 (0.00)	203 (0.33)	0
pylint	0	1277	**1898**	0 (0.00)	0 (0.00)	663 (0.26)	0
sphinx	0	451	**3722**	0 (0.00)	0 (0.00)	271 (0.07)	0
urllib3	0	347	**403**	0 (0.00)	0 (0.00)	232 (0.45)	0
Werkzeug	0	540	**1080**	0 (0.00)	0 (0.00)	365 (0.29)	0

Table 7. Categorized failure modes for the top five error-prone typing slots.

	Freq.		
Failure Mode	S	H	Q
Preds. Lacking Accessed Attrs.	10	26	
Built-in & Standard Lib. Constraints	6		
Over-constr'd Preds.		9	
Over-reliance on Param. Def. Values		6	
Union Types	1	1	13
Variable Changing Type			8
Class Query Database			7
Query Algorithm			6
Sparse, Generic Attributes			5
Complex Python Oper. Semantics			4
Attribute Types	1		3
Instance Variables			3

to the instance variable `self._r`, on which the attribute `url` is accessed. HiTyper's type prediction, `dict[str,Any]`, is wrong as `dict` does not provide the attribute `url`.

```
def __init__(self, request):
  self._r = request
  self._new_headers = {}
  self.type = urlparse(self._r.url).scheme
```

Built-in and Standard Library Constraints. Stray also struggles with built-in and standard library constraints. There are several aspects to this failure mode. First, Stray cannot handle some of the semantics of built-in types: e.g., the result of addition operations on `strs` should be of type `str`. Second, Stray sometimes produces errors with container type parameter semantics. For example, given that Stray predicts the class of a parameter `d` as `dict` and the result of `d.get('path')` as `Any`, Stray may predict the type of `d` as `dict[Any,str]` instead of `dict[str,Any]`, putting the type of `dict`'s keys in the *second*—not the first—type parameter. Further, Stray does not consider typing information for standard library callables. For example, given `prefix = os.path.commonprefix(strs)`, Stray cannot determine that `strs` is a `list`, even though the standard library function `os.path.commonprefix` accepts a `list` of path names.

Over-constrained Predictions. HiTyper sometimes makes predictions that are over-constrained given the *intraprocedural* typing context of the current function or method, and not generalizable to *interprocedural* typing constraints. For example, HiTyper's prediction of `int` as the type of the `reprname` parameter of `USTimeZone`'s constructor is not *wrong* within the scope of the constructor and class definition, but is *overconstrained* as objects of other types can be (and are) also passed to that parameter, such as the string `'Eastern'` later in the same file.

```
class USTimeZone(tzinfo):
  def __init__(self, hours, reprname, stdname, dstname):
    self.stdoffset = timedelta(hours=hours)
    self.reprname = reprname
  # ...
Eastern = USTimeZone(-5, 'Eastern', 'EST', 'EDT')
```

Over-reliance on Parameter Default Values. HiTyper also tends to be over-reliant on parameter default values, even if those default values are used as placeholders processed in separate code paths and are not the same type as typical values passed to that parameter. This error mode frequently occurs with *Predictions Lacking Accessed Attributes* or *Over-constrained Predictions*. For example, HiTyper predicts the parameter `fill_iter` of `pytz.lazy.LazyList` to be of type `None` given that it has the default value `None`. However, this would result in typing errors given usages where the parameter is passed non-`None` iterators, such as in the `setUp` method below.

```
class LazyList:
  def __new__(cls, fill_iter=None):
    if fill_iter is None: return set()
    class LazySet(set): ...
    fill_iter = [fill_iter]
    # ...
class LazyListTestCase(unittest.TestCase):
  def setUp(self):
    self.base = [3, 2, 1]
    self.lazy = LazyList(iter(list(self.base)))
```

Union Types. A failure mode affecting QuAC, and to a lesser extent, Stray and HiTyper, is the inability to predict union types. This often occurs when a parameter is involved in `isinstance` checks guarding different branches (such as in the `handle_error` below), or when a function returns values of different types from different branches. In this situation, Stray and HiTyper might only return one of the constituting types as its type prediction. In contrast, QuAC pools the attributes from different constituting types together and makes a type prediction based on that merged attribute set, which may or may not be a constituting type. This is because QuAC's analysis is *control-flow insensitive* and does not support *type narrowing* [mypy Developers 2024] (narrowing a broader type to a more specific type on program branches).

```
def handle_error(self, req, client, addr, exc):
  if isinstance(exc, InvalidRequestLine): ...
  elif isinstance(exc, InvalidRequestMethod): ...
  elif isinstance(exc, InvalidHTTPVersion): ...
```

Variable Changing Type. In Python, a variable can be transformed to a different type. For instance, a parameter `X` may originally be a `list`, but after `X = torch.Tensor(X)`, `X` is now a `torch.Tensor`. In QuAC, this may lead to both the attributes of `list` and `torch.Tensor` being in `X`'s attribute set, and as a result, the query algorithm may determine the type of the parameter `X` to be `torch.Tensor` instead of `list`.

Class Query Database. QuAC's class query database for each project records the attributes of built-in classes, standard library protocols, and other classes defined in, or accessible via inputs, within that project. This is not enough for some use cases. For instance, Python's standard library doesn't include all possible protocols that may be used in real-world projects, such as a hypothetical generic container protocol supporting indexing that could be seen as an abstract base type for both sequence (e.g., `list`) and mapping (e.g., `dict`) types. On the other hand, our class query database doesn't record possible *dynamic attributes* accessed on class instances via the `__getattr__` or `__getattribute__` methods, and a large portion of such dynamic attributes in an attribute set would lead to inaccuracies in a class query.

Query Algorithm. QuAC's use of BM25 in the class query process also has drawbacks. Given a relatively small attribute set, it may rank a smaller class missing some attributes higher than a larger class containing all the attributes. This can be attributed to the small attribute set (small n) exacerbating the effect of $|C|$ (class length) on the class's BM25 score in Equation 1.

Sparse, Generic Attributes. In some cases, a very limited number of attributes not indicative of a particular class are accessed on a variable. For example, in the function `sep` below, the parameter `s`'s attribute set only contains `__mul__`, occurring in both numeric and sequence types in the Python standard library. Given this single attribute, it is challenging for QuAC to accurately predict that `s` should be of the type `str`, a conclusion that one can reach by considering the *natural language semantics* of the function name `sep` and the names of its variable `stream`, `sep_total`, etc.

```
def sep(stream, s, txt):
  if hasattr(stream, 'sep'): stream.sep(s, txt)
  else:
    sep_total = max(70 - 2 - len(txt), 2)
    sep_len = sep_total // 2; sep_extra = sep_total % 2
    out = f'{s␣*␣sep_len}␣{txt}␣{s␣*␣(sep_len␣+␣sep_extra)}\n'
    stream.write(out)
```

Complex Python Operational Semantics. Python's operators have complex run-time behavior that can only be precisely determined given the operands' types, and, in some cases, even the values. For example, a class may define methods supporting binary arithmetic or comparison operations where the left and right-hand sides are not the same type, such as a `datetime.datetime` object defining `__add__` (the method supporting addition) accepting a `datetime.timedelta` object—not a `datetime.datetime` object—as its right-hand side. Furthermore, although both sequence and mapping types support indexing operations, indexing a sequence object with a `range` or `tuple` (e.g., `['a', 'b', 'c'][1:2]`) performs *slicing*, while indexing a mapping object (e.g. `dict`) with a `range` or `tuple` treats the `range` or `tuple` as a key and looks up its value.

Attribute Types. QuAC associates expressions with attribute sets, considering the *presence* of attributes but not their types. This sometimes leads to errors. For example, when predicting the type of the return value of the method `_filter_subplot_data` depicted below, QuAC determines it has the attribute set {`columns`, `index`, `__getitem__`} (`df` is returned from the function, and these attributes are accessed on `df`), and then predicts the class `os.terminal_size`. However, given `df.columns.intersection(['col', 'row'])`, `df`'s `columns` attribute should be a type that provides the `intersection` method accepting a `list` of `str` objects, while `os.terminal_size`'s `columns` attribute is simply a property of type `int`. Thus, predicting `os.terminal_size` is wrong.

```
def _filter_subplot_data(self, df, subplot):
  dims = df.columns.intersection(['col', 'row'])
  if dims.empty: return df
  keep_rows = pd.Series(True, df.index, dtype=bool)
  for dim in dims: keep_rows &= df[dim] == subplot[dim]
  return df[keep_rows]
```

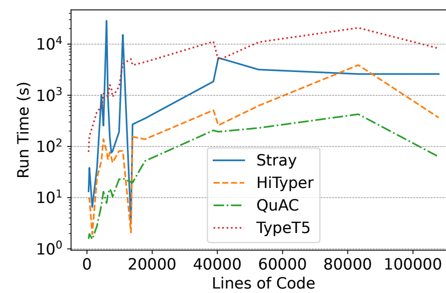
Instance Variables. Many classes have *instance variables* initialized from constructor parameters and accessed via *name lookups* on `self`. However, QuAC does not construct equivalence relationships between the constructor's parameters and the instance variables accessed later. This makes QuAC unable to associate the attribute requirements of the instance variables with those of their corresponding constructor parameters. For example, when predicting the type of the parameter `session` of `ServiceDocumenter`'s constructor initializing `self._boto3_session`, we can only record that `session` has the attribute `_session` (Line 3), but miss out the attributes `client`, `get_available_resources`, and `resource` (Lines 5,7,8). This leads to inaccuracies in predicting the types of such constructor parameters.

```
1 class ServiceDocumenter(BaseServiceDocumenter):
2   def __init__(self, service_name, session, root_docs_path):
3     super().__init__(service_name=service_name, session=session._session,
          root_docs_path=root_docs_path)
4     self._boto3_session = session
5     self._client = self._boto3_session.client(service_name)
6     self._service_resource = None
7     if self._service_name in self._boto3_session.get_available_resources():
8       self._service_resource = self._boto3_session.resource(service_name)
```

Table 8. Exact Match and Match to Parametric metrics for non-trivial type annotations.

Repository	% Exact Match S	H	Q	% Match to Parametric S	H	Q
click	0.0	26.4	**34.7**	0.0	30.7	**39.4**
flake8	15.8	19.1	**21.1**	19.5	25.4	**32.4**
Flask	2.1	17.8	**24.3**	2.6	19.3	**27.4**
ipython	0.0	20.6	**42.0**	0.0	23.7	**48.5**
Jinja2	8.6	25.5	**31.1**	8.9	28.4	**35.9**
pre_commit	0.0	**36.6**	26.9	0.0	**45.0**	36.6
pylint	0.0	20.8	**25.2**	0.0	24.0	**29.9**
sphinx	0.0	3.0	**22.0**	0.0	3.5	**29.6**
urllib3	0.0	26.8	**28.5**	0.0	27.8	**30.0**
Werkzeug	0.0	20.1	**33.6**	0.0	21.4	**37.1**

Fig. 2. Log scale run time of each technique (y-axis) and lines of code in each benchmark (x-axis).



6.4.8 How well does QuAC match existing type annotations on typed benchmarks? We compare Stray, HiTyper, and QuAC on matching existing type annotations on typed benchmarks using the Exact Match and Match to Parametric metrics for non-trivial annotations in Table 8. QuAC demonstrates a significant improvement in Exact Match and Match to Parametric performance over both Stray and HiTyper, outperforming Stray on 10/10 and HiTyper on 9/10 benchmarks. QuAC averages net percent increases of 7.3% (Exact Match) and 9.8% (Match to Parametric) over HiTyper, and 26.3% (Exact Match) and 31.6% (Match to Parametric) over Stray.

Table 9. Comprehensive comparison of QuAC and TypeT5 under Correctness Modulo Type Checker.

Repository	# Type Preds. Q	T	% Preds. Errorless Q	T	Run Time (s) Q	T	# Param'd. Preds. Q	T	% Param'd. Errorless Q	T	% Non-builtin Q	T
requests	283	**491**	85.1	**94.5**	**10.4**	932.1	**34**	31	67.4	**83.0**	**45.2**	22.8
Pygments	1133	**1637**	90.5	**92.2**	**62.9**	8237.0	**395**	58	**91.1**	76.7	**59.9**	33.5
boto3	396	**871**	91.5	**96.3**	**7.8**	1015.0	**45**	11	80.4	**96.4**	**64.4**	27.2
gunicorn	350	**605**	83.7	**94.0**	**22.9**	1476.7	**33**	8	78.8	**90.0**	**42.0**	22.0
python-dateutil	526	**839**	86.3	**91.4**	**19.6**	3878.6	**63**	17	77.8	**82.8**	**46.0**	11.6
pytz	102	**148**	86.3	**87.3**	**2.8**	486.1	**12**	2	83.3	**100.0**	**55.7**	26.6
six	26	**73**	92.3	**94.5**	**1.5**	235.9	**3**	**3**	**100.0**	**100.0**	**29.2**	19.5
pytest-cov	38	**77**	**94.7**	81.6	**2.0**	158.6	**5**	4	**100.0**	**100.0**	**69.4**	28.1
notebook	7	**27**	**100.0**	**100.0**	**1.6**	78.4	**4**	3	**100.0**	**100.0**	14.3	**23.3**
peewee	726	**1699**	92.0	**94.3**	**20.2**	5113.6	**71**	63	**85.9**	82.0	**56.6**	30.1
seaborn	1738	**2709**	86.5	**94.9**	**193.9**	4854.1	**309**	142	75.1	**83.5**	**41.4**	36.6
click	603	**1087**	92.4	**98.2**	**12.9**	1401.2	69	**275**	89.9	**100.0**	34.6	**55.0**
flake8	226	**435**	84.5	**93.7**	**7.2**	584.2	38	**128**	78.9	**90.5**	**38.7**	27.5
Flask	305	**751**	87.7	**95.0**	**11.5**	942.3	30	**135**	56.7	**99.7**	49.5	**62.6**
ipython	2367	**3774**	88.4	**91.8**	**229.9**	10914.0	314	**402**	82.7	**83.3**	**31.4**	13.7
Jinja2	886	**1651**	86.9	**97.3**	**23.8**	4119.7	89	**263**	81.7	**99.9**	55.9	**58.7**
pre_commit	453	**748**	85.0	**97.0**	**13.1**	1071.6	82	**232**	81.7	**94.2**	**45.5**	29.6
pylint	2294	**3400**	82.8	**90.6**	**206.3**	11228.7	265	**616**	80.6	**84.2**	**64.3**	38.8
sphinx	3755	**7182**	99.1	**99.7**	**426.7**	20808.5	667	**1128**	99.1	**99.9**	**52.6**	29.7
urllib3	451	**769**	89.4	**91.9**	**14.8**	1590.7	39	**172**	84.6	**88.1**	29.5	**30.4**
Werkzeug	1190	**2102**	90.8	**97.4**	**51.9**	4456.6	147	**516**	83.7	**99.4**	30.3	**42.6**

6.4.9 How does QuAC compare to an LLM-based technique? Recently, Large Language Models (LLMs) trained on code have been applied to various software analysis tasks, including type prediction. In particular, TypeT5 [Wei et al. 2023] was recently proposed for type prediction targeting both Python and JavaScript. TypeT5 is based on CodeT5 [Wang et al. 2021], an LLM trained by SalesForce. CodeT5 is pre-trained on the CodeSearchNet [Husain et al. 2019] corpus, which itself is extracted from open-source projects from GitHub, featuring popular packages from Libraries.io [Libraries.io 2023]. In the evaluation on their datasets, CodeT5 gets a net 25-34% increase in accuracy over HiTyper, and TypeT5 a further 4-5% increase in accuracy over CodeT5.

Table 10. Intersections between the errorless non-trivial typing slots of TypeT5 and other methods, and the percentages of TypeT5's covered by other methods.

Repository	T	S,T	H,T	Q,T	% S,T/T	% H,T/T	% Q,T/T
requests	464	0	**273**	231	0.0	58.9	49.8
Pygments	1509	26	715	**940**	1.7	47.4	62.3
boto3	839	208	**406**	350	24.8	48.4	41.7
gunicorn	569	89	**290**	283	15.7	51.0	49.8
python-dateutil	767	0	293	**441**	0.0	38.2	57.5
pytz	129	4	71	**72**	3.1	55.0	55.7
six	69	0	0	**23**	0.0	0.0	33.3
pytest-cov	63	0	25	**26**	0.0	39.8	41.4
notebook	27	0	**19**	7	0.0	70.4	25.9
peewee	1601	0	0	**612**	0.0	0.0	38.2
seaborn	2571	91	933	**1422**	3.5	36.3	55.3
click	1067	0	**571**	540	0.0	53.5	50.6
flake8	408	83	147	**176**	20.3	36.0	43.1
Flask	713	31	185	**246**	4.3	25.9	34.5
ipython	3464	0	1643	**1906**	0.0	47.4	55.0
Jinja2	1607	277	644	**744**	17.2	40.1	46.3
pre_commit	725	0	**437**	367	0.0	60.2	50.6
pylint	3078	0	1201	**1773**	0.0	39.0	57.6
sphinx	7163	0	375	**3568**	0.0	5.2	49.8
urllib3	707	0	326	**376**	0.0	46.1	53.2
Werkzeug	2046	0	532	**1058**	0.0	26.0	51.7

Table 11. Comparison of QuAC and TypeT5 in matching existing non-trivial type annotations, under Exact Match and Match to Parametric.

	% Exact Match		% Match to Parametric	
Repository	Q	T	Q	T
click	34.7	**52.5**	39.4	**52.5**
flake8	21.1	**58.0**	32.4	**64.6**
Flask	24.3	**45.5**	27.4	**45.5**
ipython	42.0	**60.1**	48.5	**64.8**
Jinja2	31.1	**52.1**	35.9	**52.1**
pre_commit	26.9	**85.4**	36.6	**88.1**
pylint	25.2	**42.0**	29.9	**46.7**
sphinx	22.0	**58.1**	29.6	**63.2**
urllib3	28.5	**45.1**	30.0	**49.7**
Werkzeug	33.6	**48.5**	37.1	**49.2**

A natural question is thus how QuAC compares to TypeT5 on our benchmarks. We note that as our benchmarks are popular Python repositories, it is possible that CodeT5, the LLM underlying TypeT5, has trained on them.

We report the comparison in Table 9. First, let's look at Columns 1-2 reporting the total number of non-trivial type predictions and Columns 3-4 reporting the percent of those on which mypy raises no errors. Over both untyped and typed benchmarks, TypeT5 emits more non-trivial type predictions than QuAC—1.9× on average. TypeT5's predictions are also more correct than QuAC on all benchmarks except pytest-cov. Looking at matching existing non-trivial type annotations on typed benchmarks in Table 11, we also see that TypeT5 outperforms QuAC on all benchmarks. However, TypeT5's increases in coverage and accuracy come with a heavy run time cost (Columns 5-6, Table 9 and Figure 2). TypeT5 takes significantly longer than QuAC to predict types on all benchmarks, from 25× on `seaborn` to 254× on `peewee`. The geometric mean of TypeT5's run time over QuAC's is a staggering 92×, greatly exceeding TypeT5's coverage and accuracy increases.

A more interesting story emerges when looking at the number of parameterized containers in Columns 7-8, Table 9. TypeT5 builds on top of CodeT5, which is capable of predicting both parametric and user-defined types [Wei et al. 2023]. On all untyped benchmarks, QuAC has a higher number of parameterized containers (on average 6.1%, 2.9% of all typing slots in Table 1 for QuAC and TypeT5). Though TypeT5 has higher correct percentages over nearly all benchmarks (Columns 9-10, Table 9), this does not make up for the much smaller number of parameterized containers on most untyped benchmarks. However, on the typed benchmarks, TypeT5 has a higher number of parameterized containers (on average 5.9%, 17.5% of all typing slots for QuAC and TypeT5). The difference between the results on untyped and typed benchmarks suggests there is a shift in performance depending on whether type annotations are present in CodeT5's training data. To validate this, we compare whether the difference in mean percentage of parameterized containers differs over all untyped vs. all typed benchmarks. A two-tailed t-test yielded $p = 0.91 > 0.05$ on the difference between these means for QuAC, and $p = 1.85 \times 10^{-5} \lll 0.05$ for TypeT5.

We see a similar shift in the results over untyped and typed benchmarks when looking at the percentages of errorless non-trivial type predictions that are non-builtin (Columns 11-12, Table 9). TypeT5 achieves parity with QuAC on the typed benchmarks, but QuAC has higher percentages on most (10/11) untyped benchmarks. On average, the mean percentages of errorless non-trivial type predictions that are non-builtin are 47.6% and 25.6% for QuAC and TypeT5 on the untyped benchmarks and 43.2% and 38.9% for QuAC and TypeT5 on the typed benchmarks. A two-tailed t-test yielded $p = 0.49 > 0.05$ on the difference between these means for QuAC and $p = 0.03 < 0.05$ for TypeT5. Again, there is a clear, and statistically significant, difference in the distribution of TypeT5's predictions between the untyped and typed benchmarks.

Unlike the non-LLM baselines, where QuAC's errorless non-trivial type predictions were complementary to those of the other techniques, TypeT5 covers most of Stray's, HiTyper's, and QuAC's errorless non-trivial type predictions. Looking at Table 10, we see that Stray, HiTyper, and QuAC cover 4.3%, 39.3%, 47.8% of TypeT5's errorless non-trivial type predictions on average. Interestingly, QuAC not only captures the largest share of TypeT5's errorless non-trivial type predictions on average but also does so in a consistent manner (standard deviation 9%, versus 19% for HiTyper).

Overall, our evaluation is consistent with the large lines of TypeT5's: the LLM-based approach has higher coverage and high overall accuracy. However, at the project level, these accuracies are lower than on the BetterTypes4Py/InferTypes4Py datasets. We also observe some distribution shifts in performance on typed vs. untyped projects when we look at container type parameters and non-builtin types. Finally, the run time cost of TypeT5 is much higher than QuAC and even HiTyper. On average, QuAC covers 47.8% of the LLM-based method's errorless non-trivial type predictions, with a run time $1/92\times$ that of the LLM-based method, demonstrating QuAC's efficiency.

7 Discussion

7.1 Future Research Directions

Based on the failure modes discussed above, there are several directions for future work.

Type Checker Integration. An important future research direction would be to reimplement QuAC based on a Python type checker, such as mypy [mypy Developers 2024]. These type checkers support more complex and precise internal representations and static analysis procedures that provide better support for Python's semantics and would be beneficial in addressing QuAC's failure modes of *Union Types*, *Variable Changing Type*, *Instance Variables*. Additionally, this would allow us to check and filter class predictions made by QuAC's BM25 query algorithm to find a class prediction that type checks. Such a design would help reduce the occurrence of some of QuAC's other failure modes related to the imprecision of the Top-1 queried class, such as *Query Algorithm*, *Sparse, Generic Attributes*, *Complex Python Operational Semantics*, and *Attribute Types*.

Including QuAC Within a Hybrid or Ensemble Method. Another interesting future research direction would be to include QuAC as part of a hybrid LLM-Symbolic type inference method or in an ensemble complementing other type inference methods. This is feasible as QuAC is successful on typing slots expecting a non-builtin type, in contrast to the rare types issue faced by machine learning-based type inference methods (Section 6.4.4), and its correct typing slots complement non-LLM baselines (Section 6.4.6). Moreover, this enables using machine learning-based type inference methods, especially LLM-based ones, to leverage natural language cues and overcome QuAC's *Sparse, Generic Attributes* failure mode. On the other hand, including QuAC also has the potential to improve performance and mitigate the effects of potential training bias when running on a diverse set of benchmarks compared to a purely LLM-based method.

7.2 Threats to Validity

The threats to *internal validity* lie in our implementations of type inference techniques and experiment scripts. To mitigate these threats, we reuse the existing reproduction packages for Stray [Sun et al. 2022], HiTyper [Peng et al. 2022], and TypeT5 [Wei et al. 2023]. We adopt a modular, functional coding style when developing QuAC and unit-test QuAC's components. Moreover, the implementations of Stray and QuAC require all of a project's dependencies to be installed beforehand. Therefore, we manually curate the dependencies of each benchmark in Section 6.2 and install all dependencies before running each type inference technique on a benchmark.

The threats to *external validity* lie in the baselines and benchmarks used in the evaluation. To reduce the threat, in terms of the baselines, we have used the state-of-the-art non-LLM approaches Stray [Sun et al. 2022] and HiTyper [Peng et al. 2022], representative static and machine learning techniques found to outperform other approaches in their evaluations, as well as a recent LLM-based approach, TypeT5 [Wei et al. 2023]. We did not evaluate the recent machine learning technique DLInfer [Yan et al. 2023], as it can only predict types for function parameters but not return values, and does not generate results for arguments if developer-provided type annotations are absent [Guo et al. 2024]. Concerning the benchmarks, we compiled a benchmark set in Section 6.2 consisting of several popular real-world projects spanning different domains and having vastly different project sizes that reduces the threat of selection bias. Moreover, including untyped projects follows the approach of a recent evaluation of type inference methods for migrating JavaScript codebases to TypeScript [Yee and Guha 2023]. It is justified as our motivating problem is similar—migrating untyped Python programs to type-annotated Python.

The threats to *construct validity* may come from our Correctness Modulo Type Checker criteria used on untyped Python benchmarks. To mitigate this, we have adopted the well justified approaches proposed in previous work [Allamanis et al. 2020; Yee and Guha 2023]. To prevent the effect of trivial type annotations such as `Any` hiding type errors and allowing more code to type check, we only type check *non-trivial type annotations* made by the type inference methods. As an additional mitigation, we have included a comparison in matching existing non-trivial type annotations on typed benchmarks under Exact Match and Match to Parametric.

8 Related Work

8.1 Static Type Inference Methods for Dynamic Languages

There are various static type inference methods for dynamic languages, including theoretical models such as gradually-typed lambda calculus [Campora et al. 2017; Castagna et al. 2019; Garcia and Cimini 2015; Migeed and Palsberg 2019; Miyazaki et al. 2019; Phipps-Costin et al. 2021; Siek and Vachharajani 2008], and real-world languages such as Python [Cannon 2005; Google 2024; Hassan et al. 2018; Maia et al. 2012; Meta 2024; Microsoft 2024; Salib 2004; Sun et al. 2022; Vitousek et al. 2014; Wang 2022], JavaScript [Anderson et al. 2005; Chandra et al. 2016; Jensen et al. 2009; Rastogi et al. 2012], and Ruby [Furr et al. 2009; Kazerounian et al. 2020]. These methods usually employ rule-based methods, data-flow analysis, and hand-coded heuristics to generate a set of *typing constraints* and infer types by computing solutions to these typing constraints. Despite aiming to be "correct by design" and achieving relatively high accuracy with simple types under simple typing contexts, they may only support a subset of their target languages [Anderson et al. 2005; Chandra et al. 2016], and may struggle with the dynamic nature of those languages [Richards et al. 2010], thus negatively affecting their *coverage*. Furthermore, generating and solving constraints may be computationally expensive, limiting their applicability on large-scale codebases.

Compared with static type inference methods, QuAC employs fewer hard-coded rules and heuristics and is more data-driven. Although theoretically unsound, QuAC achieves much higher coverage and competitive accuracy in our experimental evaluation against Stray, the state-of-the-art static type inference method for Python. Furthermore, QuAC, by virtue of only employing a lightweight static analysis, is highly performant and scales well to large-scale codebases.

8.2 Machine Learning-based Type Inference Methods for Dynamic Languages

Recent type inference methods for dynamic programming languages tend to employ *machine learning* techniques to handle the complexities and nuances of dynamic languages and enhance type inference coverage and accuracy.

In the research domain of type inference for Python, Xu et al. [Xu et al. 2016] introduced *probabilistic type inference*, offering multiple candidate types for variables by leveraging natural language cues and context within the code. DeepTyper [Hellendoorn et al. 2018] regards types as word labels and uses an RNN-based sequence model to predict types from a pre-defined type vocabulary. Dash et al. [Dash et al. 2018] introduce "conceptual types" which refine a single type such as `str` into more semantically detailed types such as `url` and `phone`.

However, ML-based techniques face their own set of challenges. Notably, they struggle to balance correctness and coverage, often generating a set of potential types, of which only a fraction are accurate in a given context. Additionally, ML-based techniques face difficulties in accurately predicting non-builtin types with minimal occurrences in datasets, leading to a pronounced drop in accuracy for those outlier types [Mir et al. 2021].

Recent works on machine learning-based type inference for Python focus on mitigating these issues. TypeWriter [Pradel et al. 2020] uses four separate sequence models to recommend types in Python and includes a validation phase using type checkers to filter out most wrong predictions. Given a non-type checking prediction, it searches its solution space for an alternative. Typilus [Allamanis et al. 2020] uses a graph model to represent code and utilizes meta-learning to recommend types from an open vocabulary. However, the method still requires that components of the predicted types are present in the training set. HiTyper [Peng et al. 2022] records type dependencies among variables in *type dependency graphs* and leverages type inference rules to validate predictions made by neural networks. DLInfer [Yan et al. 2023] collects *slice statements* for variables and uses a sequence model to predict types. Finally, TypeT5 [Wei et al. 2023], a recent large language model (LLM)-based approach, fine-tunes CodeT5 [Wang et al. 2021], a pretrained LLM for code. Although these models have shown great advances [Le et al. 2020], challenges remain in ensuring type correctness and predicting rare types not represented in training sets. Validation can filter invalid types out but cannot correct them, leading to potential drops in coverage. Moreover, LLM-based techniques demand substantial computational and energy resources [Chien et al. 2023; Šakota et al. 2024; Samsi et al. 2023], and despite extensive pre-training datasets, they struggle with out-of-distribution generalization and unpredictable inference behaviors [Hajipour et al. 2024].

Besides Python, there is plenty of work on machine learning-based type inference for other dynamically typed programming languages, notably JavaScript and TypeScript. DeepTyper [Hellendoorn et al. 2018] is also adapted to work on JavaScript, while NL2Type [Malik et al. 2019] is another system leveraging natural language hints to predict JavaScript types that improves on DeepTyper. LambdaNet [Wei et al. 2020] is a graph neural network that performs probabilistic type inference for JavaScript programs, and TypeBert [Jesse et al. 2021] is a model based on the BERT [Devlin et al. 2018] architecture model that achieves better performance than more sophisticated models. Building on top of TypeBert, DiverseTyper [Jesse et al. 2022] explicitly focuses on predicting *user-defined types* for TypeScript by leveraging TypeBert as a pre-trained model and using deep similarity learning to align new type declarations to uses of those declarations.

Compared with machine learning-based type inference methods, QuAC does not require a training set or training stage and works directly on the data in the Python codebase it runs on.

When attributes are abundant, QuAC can make more accurate predictions than machine learning models. It can also attain a higher coverage than letting a machine learning model predict types with no guarantee of correctness and filtering out those deemed invalid. In addition, QuAC dynamically constructs a type query database for each project where each type is treated equally, thus not suffering from the rare types problem. Furthermore, as machine learning models tend to be large, QuAC's lightweight design is also more efficient when running on large codebases.

However, the ability of machine learning-based type inference methods to leverage natural language cues and recommend types would be beneficial in situations where attributes are scarce and QuAC does not make accurate type predictions, one of QuAC's main failure modes in Section 6.4.7. Furthermore, QuAC's correct type predictions complement those made by Stray and HiTyper in Section 6.4.6, and QuAC is much more efficient and significantly more consistent at predicting container type parameters and non-builtin types than TypeT5 in Section 6.4.9. Thus, including QuAC in an ensemble with machine learning methods or as part of a hybrid LLM-Symbolic method to leverage each other's advantages would be a feasible direction for future work.

9 Conclusion

We propose QuAC (Quick Attribute-Centric Type Inference), a novel type inference approach for Python inspired by Python's duck typing. By collecting attribute sets for Python expressions, employing information retrieval techniques, and modeling container type parameter semantics, QuAC strikes a balance between correctness and coverage and achieves exceptional run-time performance, as demonstrated by our experimental results on popular untyped Python projects. Moreover, QuAC also excels in predicting container type parameters and non-builtin types, demonstrating great potential in synergistically complementing existing type inference methods. Finally, on average, QuAC is 92× faster than an LLM-based baseline while covering 47.8% of its errorless non-trivial type predictions and being significantly more consistent in predicting container type parameters and non-builtin types. This suggests QuAC could be used to soften the cost and generalization challenges of LLM-based methods.

Data-Availability Statement

The code, benchmarks, and data replication scripts supporting Sections 5 and 6 are available on Zenodo [Wu and Lemieux 2024].

Acknowledgments

We thank the anonymous reviewers for their feedback which greatly helped improve the paper, and the anonymous artifact reviewers who helped improve the artifact. This research is supported by a Google Research Scholar Award in Software Engineering and Programming Languages (2023). This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. NN66001-22-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific.

References

Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3385997

Murugan Anandarajan, Chelsey Hill, Thomas Nolan, Murugan Anandarajan, Chelsey Hill, and Thomas Nolan. 2019. Term-document representation. *Practical Text Analytics: Maximizing the Value of Text Data* (2019). https://doi.org/10.1007/978-3-319-95663-3_5

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *ECOOP-Object-Oriented Programming: 19th European Conference*. https://doi.org/10.1007/11531142_19

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2017. Migrating gradual types. *Proceedings of the ACM on Programming Languages, Volume 2, Issue POPL* (2017). https://doi.org/10.1145/3158103

Brett Cannon. 2005. *Localized type inference of atomic types in Python.* California Polytechnic State University.

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G Siek. 2019. Gradual typing: a new perspective. *Proceedings of the ACM on Programming Languages, Volume 3, Issue POPL* (2019). https://doi.org/10.1145/3290329

Satish Chandra, Colin S Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type inference for static compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/2983990.2984017

Andrew A Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. 2023. Reducing the Carbon Impact of Generative AI Inference (today and in 2035). In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*. https://doi.org/10.1145/3604930.3605705

Santanu Kumar Dash, Miltiadis Allamanis, and Earl T Barr. 2018. Refinym: Using names to refine types. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3236024.3236042

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). https://doi.org/10.48550/arXiv.1810.04805

Gregory Ewing. 2009. PEP 380 – Syntax for Delegating to a Subgenerator. https://peps.python.org/pep-0380/. Access Date: March 20, 2024.

Gregory Ewing and Guido van Rossum. 2005. PEP 342 – Coroutines via Enhanced Generators. https://peps.python.org/pep-0342/. Access Date: March 20, 2024.

Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*. https://doi.org/10.1145/1529282.1529700

Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2017.75

Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2676726.2676992

GitHub. 2023. Octoverse 2023 – The state of open source. https://octoverse.github.com/. Access Date: March 20, 2024.

Google. 2024. Pytype. https://github.com/google/pytype. GitHub repository.

Yimeng Guo, Zhifei Chen, Lin Chen, Wenjie Xu, Yanhui Li, Yuming Zhou, and Baowen Xu. 2024. Generating Python type annotations from type inference: how far are we? *ACM Transactions on Software Engineering and Methodology* (2024). https://doi.org/10.1145/3652153

Hossein Hajipour, Ning Yu, Cristian-Alexandru Staicu, and Mario Fritz. 2024. SimSCOOD: Systematic Analysis of Out-of-Distribution Generalization in Fine-tuned Source Code Models. In *Findings of the Association for Computational Linguistics: NAACL*. https://doi.org/10.18653/v1/2024.findings-naacl.90

Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *Computer Aided Verification: 30th International Conference*. https://doi.org/10.1007/978-3-319-96142-2_2

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3236024.3236051

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

IEEE Spectrum. 2023. The Top Programming Languages 2023. https://spectrum.ieee.org/top-programming-languages. Access Date: March 20, 2024.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*. https://doi.org/10.1007/978-3-642-03237-0_17

Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. 2021. 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*. https://doi.org/10.1145/3468264.3473135

Kevin Jesse, Premkumar T Devanbu, and Anand Sawant. 2022. Learning to predict user-defined types. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2022.3178945

Milod Kazerounian, Brianna M Ren, and Jeffrey S Foster. 2020. Sound, heuristic type annotation inference for Ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. https://doi.org/10.1145/3426422.3426985

Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* (2020). https://doi.org/10.1145/3383458

Libraries.io. 2023. Search Results for PyPI Packages Ordered by Rank. https://libraries.io/search?order=desc&platforms=PyPI&sort=rank. Access Date: July 31, 2023.

Eva Maia, Nelma Moreira, and Rogério Reis. 2012. A static type inference for Python. *Proc. of DYLA* (2012).

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41th International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2019.00045

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press. https://doi.org/10.1017/CBO9780511809071

Meta. 2024. Pyre. https://github.com/facebook/pyre-check. GitHub repository.

Microsoft. 2024. Pyright. https://github.com/microsoft/pyright. GitHub repository.

Zeina Migeed and Jens Palsberg. 2019. What is decidable about gradual types? *Proceedings of the ACM on Programming Languages, Volume 4, Issue POPL* (2019). https://doi.org/10.1145/3371097

Nevena Milojkovic, Mohammad Ghafari, and Oscar Nierstrasz. 2017. It's duck (typing) season!. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2017.10

Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. ManyTypes4Py: A benchmark Python dataset for machine learning-based type inference. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR52588.2021.00079

Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical deep similarity learning-based type inference for Python. In *Proceedings of the 44th International Conference on Software Engineering*. https://doi.org/10.1145/3510003.3510124

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic type inference for gradual Hindley–Milner typing. *Proceedings of the ACM on Programming Languages, Volume 3, Issue POPL* (2019). https://doi.org/10.1145/3290331

mypy Developers. 2024. mypy - Optional Static Typing for Python. https://mypy-lang.org/. Access Date: March 20, 2024.

mypy Developers. 2024. Type Narrowing - MyPy Documentation. https://mypy.readthedocs.io/en/stable/type_narrowing.html. Access Date: March 20, 2024.

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for Python. In *Proceedings of the 44th International Conference on Software Engineering*. https://doi.org/10.1145/3510003.3510038

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based gradual type migration. *Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA* (2021). https://doi.org/10.1145/3485488

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3368089.3409715

Python Software Foundation. 2020. The Python Language Reference, Version 3.9. https://docs.python.org/3.9/reference/. Access Date: March 20, 2024.

Python Core Developers. 2024. pyperformance: Python Performance Benchmark Suite. https://github.com/python/pyperformance. Access Date: March 20, 2024.

Python Software Foundation. 2020. Abstract Syntax Trees - Python 3.9 Documentation. https://docs.python.org/3.9/library/ast.html. Access Date: March 20, 2024.

Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. https://doi.org/10.1145/3426422.3426981

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The ins and outs of gradual type inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2103656.2103714

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/1806596.1806598

Stephen Robertson. 2004. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of Documentation* (2004). https://doi.org/10.1108/00220410410560582

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* (2009). https://doi.org/10.1561/1500000019

Marija Šakota, Maxime Peyrard, and Robert West. 2024. Fly-swat or cannon? cost-effective language model choice via meta-modeling. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. https://doi.org/10.1145/3616855.3635825

Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From words to watts: Benchmarking the energy costs of large language model inference. In *IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC58863.2023.10363447

Yury Selivanov. 2015. PEP 492 – Coroutines with async and await syntax. https://peps.python.org/pep-0492. Access Date: March 20, 2024.

Jeremy G Siek and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In *Proceedings of the Symposium on Dynamic Languages*. https://doi.org/10.1145/1408681.1408688

Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. 2022. Static Type Recommendation for Python. In *Proceedings of the 37th International Conference on Automated Software Engineering*. https://doi.org/10.1145/3551349.3561150

The Computer Language Benchmarks Game Team. 2023. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. Access Date: March 20, 2024.

Typeshed Contributors. 2024. Typeshed: Stubs for Python standard library and third-party libraries. https://github.com/python/typeshed. Access Date: March 20, 2024.

Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. https://peps.python.org/pep-0484/. Access Date: March 20, 2024.

Guido van Rossum, Ivan Levkivskyi, Jukka Lehtosalo, Łukasz Langa, and Michael Lee. 2018. PEP 544 – Protocols: Structural subtyping (static duck typing). https://peps.python.org/pep-0544/. Access Date: March 26, 2024.

Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*. https://doi.org/10.1145/2661088.2661101

Yin Wang. 2022. PySonar2. https://github.com/yinwang0/pysonar2. GitHub repository.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021). https://doi.org/10.48550/arXiv.2109.00859

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq type inference using static analysis. *arXiv preprint arXiv:2303.09564* (2023). https://doi.org/10.48550/arXiv.2303.09564

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020). https://doi.org/10.48550/arXiv.2005.02161

Jifeng Wu and Caroline Lemieux. 2024. *QuAC: Quick Attribute-Centric Type Inference for Python*. https://doi.org/10.5281/zenodo.13367665

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/2950290.2950343

Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *Proceedings of the 45th International Conference on Software Engineering*. https://doi.org/10.1109/ICSE48619.2023.00170

Ming-Ho Yee and Arjun Guha. 2023. Do Machine Learning Models Produce TypeScript Types that Type Check? *arXiv preprint arXiv:2302.12163* (2023). https://doi.org/10.48550/arXiv.2302.12163

Jelle Zijlstra. 2024. typeshed_client: Retrieve information from typeshed and other typing stubs. https://github.com/JelleZijlstra/typeshed_client. Access Date: March 20, 2024.

Received 2024-04-05; accepted 2024-08-18