QuAC: Quick Attribute-Centric Type Inference for Python

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

Python's dynamically typed nature 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 hints can make these tools much more useful. *Type inference tools* aim at reducing developers' burden of adding these type hints. Existing type inference tools struggle over aspects including type correctness, dynamic features, rare non-builtin types, and computational effort. Inspired by Python's duck-typed nature, 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 untyped Python projects. Compared to our baselines, QuAC generates type annotations with high accuracy complementary to those made by the baselines while not sacrificing coverage. Further, it demonstrates clear advantages in predicting non-builtin types and container type parameters and reduces run times by an order of magnitude.

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. In . ACM, New York, NY, USA, 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Python's ascent to prominence is notable in the realm of programming languages. According to analyses from GitHub Octoverse [GitHub 2023] and IEEE Spectrum [IEEE Spectrum 2023], Python has emerged as 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 hints. These type hints 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 with type hints. A 2020 study of Python types in the wild [Rak-amnouykit et al. 2020] found that six years after introducing PEP 484, only 2,678 of 70,000 analyzed repositories had type hints. Further, on average, 1,144 repositories have less than 1 type hint per file. This conflict—the clear advantages of type hints 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

SPLASH '24, Oct 22–27, 2024, Sacramento, CA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

type hints 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 hints 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 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. Finding a balance between correctness and coverage remains challenging for type inference tools.

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

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
```

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 `tuple`, an `array.array`, or any other *sequence type*. Python's `typing` module defines an interface that covers all such types: `typing.Sequence`. Note that providing the *attribute set* {`__getitem__`, `__iter__`} is necessary for the 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 `typing.Any`. The academic static type inference tool Stray [Sun et al. 2022] also fails to infer 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`.

Observe, above, that the *attribute set* accessed on each 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 high correctness and high coverage. 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 its database of available classes for classes implementing the given attribute set. Considering that rare attributes are more suggestive of specific classes, QuAC utilizes BM25 queries, a standard information retrieval technique [Robertson et al. 2009]. Additionally, QuAC recursively applies its attribute collection and database querying technique to infer container type parameters. For example, QuAC can successfully predict the type hints for the parameter `points` and the return value of the global function `maximize` in Listing 1 to be `typing.Sequence[Point]` and `Point`, respectively.

To evaluate QuAC's performance, we compare QuAC to state-of-the-art 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 inferring both parameters and return values found to outperform other approaches in their evaluations. Considering that a major goal of Python type prediction 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 *untyped* Python projects. Inspired by a similar evaluation of TypeScript type prediction 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 `typing.Any` or `None`) predictions. Further, we compare QuAC's ability to infer container type parameters and non-builtin types against Stray and HiTyper, and compare their run times on benchmarks of different sizes.

In total over all benchmarks, QuAC achieves type prediction correctness higher than HiTyper and Stray while retaining a competitive type prediction coverage. Moreover, it can infer type parameters for containers with high correctness and coverage. Furthermore, by analyzing the typing slots on which the different techniques infer 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 prediction methods. Its typing slots with correct type predictions, in general, complement both baseline approaches, suggesting its potential to be used in an ensemble type prediction method. Finally, QuAC is orders of magnitude faster than both baseline approaches, with analysis time remaining under a minute even for projects with 100k lines of code.

Overall, this paper makes the following contributions:

- We introduce QuAC (ref. Section 4), a method for type inference 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 (anonymized for review): https://anonymous.4open.science/r/quac-0C64
- We evaluate QuAC and baseline techniques on a set of popular untyped Python projects (ref. Section 6), demonstrating QuAC's advantages in overall accuracy, non-builtin types, container type parameters, and run times, while not sacrificing coverage.

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 (ref. Section 4.2.3) and recursively infers type parameters for containers (ref. 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)]
19       # ...
```

Inferring 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 4.2.3) 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`.

Inferring 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 `typing.Iterable[T]`, where `T` is a *type parameter* representing the type of the items iterated over it. To infer `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`) element 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 `typing.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`.

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 `typing.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 4.2.1.

3 BACKGROUND

This section provides additional background on topics necessary to more 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, opening doors for enhanced code completion in IDEs, more effective static analysis, refined refactoring, and code generation utilizing type information.

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 `typing.Any` represents a dynamically-typed value of an

Listing 3. Examples of Python Type Annotations

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

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

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 with new features frequently added, 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. This research aims to infer the most 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 infer 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 is a file that outlines the public interface (classes, variables, and functions) of a Python module and contains type annotations. They generally adhere to Python syntax but replace 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

Given a set of AST expression nodes $E = \{e_1, \ldots, e_n\}$ representing the usages of a variable, QuAC runs Algorithm 1 to predict its type. For a function parameter, E is simply the set of expression

Algorithm 1: QuAC Type Prediction, initialized with the set of AST expression nodes representing the usages of a variable whose type is being predicted.

Data: A set of AST expression nodes $E = \{e_1, \ldots, e_n\}$
Result: The predicted type, T, of the AST expression nodes E

1 $A \leftarrow \bigcup_{e \in E}$ `GetAttributeSet`(e);
2 $C \leftarrow$ `ClassPrediction`(A);
3 $\mathbf{T} \leftarrow []$;
4 **for** $\mathcal{R} \in$ `GetRelationSetsOfTypeParameters`(C, E) **do**
5     $E' \leftarrow \bigcup_{e \in E, r \in \mathcal{R}}$ `GetAssociatedExpressions`(e, r);
6     $T' \leftarrow$ `TypePrediction`(E');
7   add T' to $\mathbf{T}$;
8 **if** $\mathbf{T} \neq []$ **then**
9     $T \leftarrow$ `Parameterize`$(C, \mathbf{T})$;
10 **else**
11     $T \leftarrow C$;
12 **return** T

nodes corresponding to usages of that parameter. For a function return value, E contains a *symbolic return value node* ρ representing the value returned from that function, as described in Section 4.2.1.

In Algorithm 1, we first *collect* and *merge* the attribute sets of each expression (Line 1). We collect attributes accessed on those expressions (described in Section 4.2.1), before merging them with a simple union, as illustrated on Line 1. Then, we predict a class C based on the merged attribute set (Line 2, described in Section 4.2.3). If C is a *generic container* (e.g., `dict`), we predict its *type parameters*; otherwise, we go straight to Line 11 and return C as our type prediction.

For generic containers, we collect the set of AST nodes E' capturing the usages of each type parameter (Lines 4,5, described in Section 4.3). We run Algorithm 1 recursively on E' to infer the parameter type T' (Line 6), before adding T' to the list of parameter types $\mathbf{T}$ (Line 7). Then, we *parameterize* the predicted generic container with $\mathbf{T}$ (Line 9) to derive the final type prediction result T. This recursive algorithm allows us to predict non-parametric types (e.g., `int`) and parametric types with arbitrary nested levels (e.g., `dict[str, list[list[int]]]`) in a unified manner.

4.2 Predicting Classes

The first step in our type inference procedure is predicting what *class* an expression is most likely to be. To accomplish this task, we assign each Python expression an *attribute set* representing the attributes in an unknown class. We populate these attribute sets by *collecting attributes based on syntactic constructs* and *constructing subset relations among the attribute sets of related expressions*. Then, we *query classes* based on these attribute sets.

4.2.1 Collecting Attributes. We perform attribute collection by walking the AST of the Python code and adding attributes to the attribute sets of Python expressions in a *syntax-directed* manner. This involves adding not only attributes directly accessed (e.g., `x.y` accesses the attribute `y` on variable `x`) but also *special methods* (ref. Section 3.2) accessed internally by the Python interpreter through syntactical constructs. For example, the indexing of an object (e.g.,`x[y]`) requires that the object supports indexing via the `__getitem__` method. A `with` statement requires that its `with_item` term (the `x` in `with x as y`) is a *context manager* providing the `__enter__` and `__exit__` methods.

A complete list of what special methods each Python expression and statement implies can be found in Python's Language Reference [Python Software Foundation 2020].

We may encounter expressions without attribute accesses in a function body. In these situations, it may still be possible to populate their attribute sets through other *related expressions*, whose attribute sets we assume to be *subsets* of these expressions'. Specifically, we consider the following cases, where $A(e)$ represents the attribute set of expression e:

- **Assignments** (including *parameter default values*). Given $x = y$, we consider $A(y) \subseteq A(x)$.
- **Function return values.** A Python function may have multiple return statements and may return a *generator-iterator* 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. For functions returning a generator-iterator or coroutine object o, we initialize ρ's attribute set with the attribute set of o, and add relations between ρ and other returned, yielded, or awaited expressions within the function body to predict the type parameters of the generator-iterator or coroutine (ref. Section 4.3.1).
- **Function calls.** Suppose we can precisely determine which function is being called at a call site (discussed in detail in Section 4.2.2). In that case, we consider the attribute sets of *function parameters* within the function definition to be *subsets* of the attribute sets of values passed to those parameters at the call site. For example, if a function `f` has the definition `def f`(x_1, x_2) and there exists a call site `f`(y_1, y_2), then $A(x_1) \subseteq A(y_1)$ and $A(x_2) \subseteq A(y_2)$. Similarly, we add the attribute sets of *function return values* to those of *function call results* at call sites. This is our approach toward a *modular, interprocedural* analysis of typing constraints.

We also consider situations where two attribute sets are *mutual subsets* (i.e., *equivalent*):

- The operands and results of *arithmetic and logical operations* (except `*` which allows multiplying sequences and integers and `%` which allows formatting strings).
- The left and right hand sides of *comparisons*.
- 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.
- Accessing a previously defined (with respect to Python's scoping rules) name later on in the code. This is our approach to *name resolution*.

4.2.2 Resolving Function Calls. As mentioned before, accurately resolving function calls is the basis of a modular, interprocedural analysis of typing constraints. To accomplish this goal, we associate a *runtime term set* with each Python expression. Based on these runtime term sets, we can then resolve most calls either to user-defined code or the Python standard library. Runtime terms include *modules*, *classes*, *global functions*, *unbound methods*, *instances*, and *instance methods*. To identify these terms, we first populate the runtime terms of names that *directly resolve to defined or imported modules, classes, global functions, and instances.* Then, we populate the runtime terms of *derivative expressions* resulting from the following rules:

R-1. Accessing modules, classes, functions, and instances on modules.
R-2. Accessing unbound methods on classes.
R-3. Accessing unbound and instance methods on instances.
R-4. Calling a class results in an instance of that class.
R-5. Calling a global function, unbound method, instance, or instance method in the Python standard library results in an instance determined via a Typeshed (ref. Section 3.3) lookup.

Example. Fig. 1 shows sample code (left), and the runtime term sets (right) QuAC populates for different expressions in the code. We walk through QuAC's runtime term collection procedure on

```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	Runtime 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 runtime terms associated with each Python expression. The table on the right gives the runtime term sets QuAC populates for Python expressions in the code listing.

this example. From `import re as r`, we add Python's `re` module as a runtime term for `r`. Then, we apply the above rules to populate the runtime terms of derivative expressions:

- (R-1) We add the global function `re.compile` as a runtime term for `r.compile`.
- (R-5) Typeshed says `re.compile`'s return value is an instance of `re.Pattern`: we add this as a runtime term for `r.compile(pattern)` (and the assignment target `regex`).
- (R-3) We add the instance method `re.Pattern.match` as a runtime term for `regex.match`.
- (R-5) Typeshed says `re.Pattern.match` returns None or an instance of `re.Match`: we add both as runtime terms for `regex.match(characters, pos)` (and the assignment target `match`).
- (R-3) We add the instance method `re.Match.group` as a runtime term for `match.group`.
- (R-5) Typeshed says `re.Match.group` returns an instance of `str` or `bytes`: we add both as runtime terms for `match.group(0)` (and the assignment target `text`).

Through this procedure, we determine what class, global function, unbound method, instance, or instance method is being called at a large number of call sites. If the *called function or method*[1] is *user-defined*, we add subset relations between the attribute sets of parameters/arguments and return values/call results (ref. Section 4.2.1). If the called function or method is from the Python standard library, QuAC adds an extra step. It initializes *dummy parameters and return values* for the callable and initializes their attribute sets by looking up Typeshed, before adding subset relations as for user-defined callables.

4.2.3 Querying Classes. The last step in class prediction is querying classes (Line 2 in Algorithm 1) for an attribute set. To accomplish this goal, we first construct a database 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 `typing.Iterable` representing any object supporting iteration, and `typing.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., any class supporting iteration) rather than concrete classes satisfying that interface requirement (`list`, `set`, etc.) This is a commonly-encountered situation given the duck-typed nature of Python.
- User-defined classes within the Python files being analyzed, and classes within imported external modules (both Python standard library and third-party).

[1]Calling a class boils down to calling its constructor while calling an instance boils down to calling its `__call__` method.

From this database, we query candidate classes using the Okapi BM25 ranking function [Robertson et al. 2009], an information retrieval heuristic. Given an attribute set $A = \{a_1, \ldots, a_n\}$, the BM25 score of a class C is:

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

where $f(a_i, C)$ is the number of times[2] a_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 2008], we use $k_1 = 1.50$ and $b = 0.75$ in this study. $\text{IDF}(a_i)$ is the IDF (inverse document frequency) weight of the attribute a_i. It captures the *rarity* of the attribute, or how much *information* the attribute provides [Robertson 2004]. Given the set of candidate classes $\mathbf{C} = \{C_1, C_2, \ldots C_N\}$, the IDF of attribute a_i is calculated as:

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

where $n(a_i) = |\{C \in \mathbf{C} : a_i \in C\}|$ is the number of classes containing a_i.

The rationale for using IDF weighting stems from not all attributes being equal in class inference, 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 strongly suggests that `str` is a likely class.

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 specific types (e.g., `K = str`, `V = int` for `dict[str, int]`, ref. 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 extremely 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 relatively 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 which extends `list[int]` but does not contain type parameters itself.

[2] As Python classes do not include duplicate attributes, this is either 1 if the attribute is present, or 0 if it is absent.

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 `typing.Iterable` and `typing.Callable`.[3] 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 infer container 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* $\mathcal{R}(K) = \{\texttt{KeyOf}, \texttt{IterTargetOf}\}$ and $\mathcal{R}(V) = \{\texttt{ValueOf}\}$, respectively. A complete description of all relations can be found in Section 4.3.2 below. For each standard library container,[4] we have specified its number of type parameters and *relation sets* for each type parameter. QuAC retrieves these in the call to `GetRelationSetsOfTypeParameters` on Line 4 of Algorithm 1.

When analyzing the code, we *associate* (potential) container expressions with *semantically related* expressions based on *syntax-directed*, *type-agnostic* association rules for each relation. These rules are detailed in Section 4.3.2. As one example, given the expressions e_1, e_2, and $e_3 = e_1[e_2]$ in source code, we record that e_2 `KeyOf` e_1 and e_3 `ValueOf` e_1, even if the types of e_1, e_2, and e_3 are unknown.

After analyzing the code, given a potential container expression e and a relation r, we can query *all expressions associated with e via r* through $\texttt{GetAssociatedExpressions}(e, r)$ (Line 5 of Algorithm 1). In the example above, we have $e_2 \in \texttt{GetAssociatedExpressions}(e_1, \texttt{KeyOf})$ and $e_3 \in \texttt{GetAssociatedExpressions}(e_1, \texttt{ValueOf})$.

4.3.2 Relations and Association Rules. For each relation below, we give a natural language description of its semantics, example containers whose type parameters have this relation, and *association rules* describing under what circumstances we associate a potential container expression e with another expression e' via a relation r.

- **`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, $\texttt{ValueOf} \in \mathcal{R}(\texttt{T})$ for `list[T]`, $\texttt{KeyOf} \in \mathcal{R}(\texttt{K})$, $\texttt{ValueOf} \in \mathcal{R}(\texttt{V})$ for `dict[K, V]`.

Association Rule. Given a non-slicing indexing operation $e_1[e_2]$, e_2 `KeyOf` e_1, $e_1[e_2]$ `ValueOf` e_1.

- **`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, $\texttt{IterTargetOf} \in \mathcal{R}(\texttt{T})$ for `set[T]` and `list[T]`, $\texttt{IterTargetOf} \in \mathcal{R}(\texttt{K})$ for `dict[K, V]`, and $\texttt{IterTargetOf} \in \mathcal{R}(\texttt{Y})$ for `typing.Generator[Y, S, R]`.

Association Rules. (1) Given `for` e_1 `in` e_2, e_1 `IterTargetOf` e_2. (2) Given `yield` e in a function that returns a generator-iterator g, e `IterTargetOf` g.

- **`Element i Of`.** In Python, tuples are immutable and usually usually contain *heterogeneous* elements. Reflecting this usage pattern, tuples are frequently constructed using the *literal notation*

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

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

of separating items with commas (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` $\in \mathcal{R}(T_i)$.
Association Rule. Given a tuple literal $(e_1, \ldots, e_n)$, e_i `Element i Of` $(e_1, \ldots, e_n)$.

- **`Parameter i Of`, `ReturnValueOf`.** Python allows annotating simple *callable objects* (no variadic arguments, keyword-only parameters) using `typing.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 `typing.Callable`$[[T_1, \ldots, T_n], T_r]$, where `Parameter i Of` $\in \mathcal{R}(T_i)$, `ReturnValueOf` $\in \mathcal{R}(T_r)$.

Association Rule. Given a call $e(e_1, \ldots, e_n)$, e_i `Parameter i Of` e, $e(e_1, \ldots, e_n)$ `ReturnValueOf` e.

- **`SendTargetOf`.** PEP 342 [Ewing and van Rossum 2005] allows values to be sent to generator-iterators, which then become the results of yield expressions within the generator-iterator. The type parameter `S` of `typing.Generator[Y, S, R]` captures the type of values sent to generator-iterators of this type, i.e., `SendTargetOf` $\in \mathcal{R}(S)$.

Association Rule. Given `yield` e in a function returning a generator-iterator g, we record (`yield` e) `SendTargetOf` g.

- **`YieldFromAwaitTargetOf`.** PEP 380 [Ewing 2009] allowed a generator-iterator to delegate part of its operations to *another* generator-iterator through the `yield from` expression. Later on, PEP 492 [Selivanov 2015] introduced *coroutines* to Python, allowing a coroutine to obtain the result of *another* coroutine through the `await` expression. In both cases, a value in one of the return statements of the *second* generator-iterator or coroutine is assigned to the `yield from` or `await` expression of the *first* generator-iterator or coroutine. The type parameter R in `typing.Generator[Y, S, R]` or `typing.Coroutine[Y, S, R]` represents the type of this value, i.e., `YieldFromAwaitTargetOf` $\in \mathcal{R}(R)$.

Association Rules. (1) Given `return` r in a function that returns a generator-iterator g or a coroutine c, we record r `YieldFromAwaitTargetOf` g or r `YieldFromAwaitTargetOf` c. (2) Given `yield from` e, we record (`yield from` e) `YieldFromAwaitTargetOf` e, and given `await` e, we record (`await` e) `YieldFromAwaitTargetOf` e.

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 4.2.2). 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 4.2.3), we also let the Python interpreter import the Python files being analyzed as *modules* and utilize 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 4.2.2). We used the provided reproduction packages to run the baseline methods Stray [Sun et al. 2022] and HiTyper [Peng et al. 2022]. 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.

The code, anonymized for review, is available on https://anonymous.4open.science/r/quac-0C64. After the review process, we will include the benchmarks and data replication scripts and submit the code for evaluation by the artifact submission deadline (July 5, 2024).

Table 1. Stats of benchmark programs used in our evaluation; we will abbreviate python-dateutil as dateutil.

Repository Name	Version	Lines of Code	Typing Slots	GitHub Stars	Dependent Packages
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

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 good type inference tools. RQ (3) evaluates whether QuAC can infer container type hints effectively. RQs (4), (5), and (7) evaluate the complementarity of QuAC and its baselines. RQ (6) evaluates the runtime performance of QuAC.

(1) Is QuAC able to predict more type annotations than the baselines?
(2) Are QuAC's predictions more correct than the baselines?
(3) Is QuAC effective at inferring container type parameters?
(4) Does QuAC predict non-builtin types more often than the baselines?
(5) Are QuAC's predictions complementary to those of our baselines?
(6) How does the run time of QuAC compare against the baselines?
(7) What are the main failure modes of QuAC and the baselines?

6.2 Baselines and Benchmarks

We evaluate QuAC against the state-of-the-art static type prediction method Stray [Sun et al. 2022] and the state-of-the-art machine learning-based technique HiTyper [Peng et al. 2022] using the most popular untyped pure-Python projects [Libraries.io 2023] with greatly varying project sizes as benchmarks. We believe these benchmarks are representative of real-world Python projects to which type annotations can be added. Table 1 describes key statistics of the benchmarks.

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]`, but could be any "dict-like" type providing an `items` method which returns a `typing.Iterable[tuple[str, bool]]`. Thus, `typing.Mapping[str, bool]`, which parameterizes the protocol `typing.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 `typing.Mapping[str, bool]` is a correct type prediction for `params` under this approximation, so would `typing.Mapping[object, object]` and `typing.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, `typing.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 respects Python's duck-typed nature, is based on how expressions are actually used within the project, and would work even if type annotations are unavailable.

To achieve this goal, we take inspiration from the *correctness modulo type checker* approach proposed in Typilus [Allamanis et al. 2020] and used in a recent evaluation of type prediction methods for TypeScript [Yee and Guha 2023]. 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 Is QuAC able to predict more type annotations than the baselines? To investigate QuAC's ability to predict type annotations compared to the baselines, we analyze the number of typing slots with non-trivial (not empty, `None`, or `typing.Any`) type predictions, as presented in the "Non-Trivial Type Preds." column in Table 2. We see that Stray lags behind both HiTyper and QuAC regarding the total number of 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`). 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 state-of-the-art machine learning model at achieving high coverage. In fact, in terms of total non-trivial type predictions across our benchmarks, QuAC exceeds HiTyper.

6.4.2 Are QuAC's predictions more correct than the baselines? We then investigate the correctness of these non-trivial type predictions by examining the percentages of them that are correct using

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

	# Type Preds.			% Correct		
Repository	S	HT	QuAC	S	HT	QuAC
requests	0	**334**	283	—	83.5	**84.8**
Pygments	31	1079	**1133**	87.1	71.4	**90.6**
boto3	249	**565**	396	90.0	72.2	**91.4**
gunicorn	104	**387**	350	**86.5**	76.2	83.4
dateutil	0	340	**397**	—	80.6	**82.4**
pytz	6	**119**	88	66.7	81.5	**83.0**
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**	—	—	**91.7**
seaborn	101	1199	**1738**	**94.1**	80.2	86.0

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

	# Param'd. Preds.			% Correct		
Repository	S	HT	QuAC	S	HT	QuAC
requests	0	22	**34**	—	59.1	**67.7**
Pygments	5	205	**395**	**100**	26.3	91.1
boto3	29	**51**	45	**86.2**	68.6	80.0
gunicorn	4	**37**	33	50.0	46.0	**78.8**
dateutil	0	32	**52**	—	53.1	**76.9**
pytz	0	4	**8**	—	0.00	**75.0**
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**	70.0	74.1

the *correctness modulo type checker approach*, 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 two benchmarks (gunicorn, seaborn) where Stray is higher. However, on these two benchmarks, QuAC achieves 3× and 15× more total errorless non-trivial type predictions than Stray. These results suggest that QuAC's design focuses on accuracy and, compared to the baselines, predicts type annotations with higher overall accuracy while not sacrificing the absolute number of predictions made.

6.4.3 Is QuAC effective at inferring container type parameters? Recall QuAC has special handling for containers — recursively inferring their type parameters (ref. Section 4.3). We evaluate QuAC's success on this front by recording the (1) total number of containers with non-trivial type parameters predicted by QuAC and the baselines, 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 greatly outperform Stray on all benchmarks. QuAC further outperforms HiTyper on all but three benchmarks. Out of these predictions, QuAC's are most likely to be errorless, exceeding HiTyper on all benchmarks. Stray achieves a slightly higher correctness on three benchmarks, but QuAC has much higher coverage on these. This data suggests that QuAC's approach to inferring container type parameters is more effective than the baselines.

6.4.4 Does QuAC predict non-builtin types more often than the baselines? 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 prediction methods may prioritize builtin types [Sun et al. 2022]. Further, non-builtin types is also one of the bottlenecks of machine learning-based type prediction 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,

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

	% Preds. that are non-builtin			# Non-builtin preds.		
Repository	S	HT	QuAC	S	HT	QuAC
requests	—	12.56	**45.0**	0	51	**108**
Pygments	6.06	8.62	**59.8**	2	83	**614**
boto3	6.9	10.1	**64.4**	22	62	**233**
gunicorn	7.4	8.6	**42.1**	10	42	**123**
dateutil	—	4.8	**42.5**	0	62	**139**
pytz	11.11	29.5	**48.0**	3	**54**	35
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	**377**
seaborn	8.9	19.0	**41.4**	10	273	**619**

Table 5. Run times of each technique in seconds; QuAC runs in under a minute on all benchmarks.

	Run Time (s)		
Repository	S	HT	QuAC
requests	82.04	50.86	**3.45**
Pygments	2,482.63	338.46	**39.79**
boto3	31,717.97	79.88	**4.89**
gunicorn	186.63	76.70	**6.90**
dateutil	275.76	150.88	**14.64**
pytz	38.85	32.32	**2.11**
six	6.45	2.00	**1.32**
pytest-cov	36.91	9.81	**1.45**
notebook	17.20	9.39	**1.57**
peewee	3.10	**2.09**	7.10
seaborn	5,233.91	225.33	**23.59**

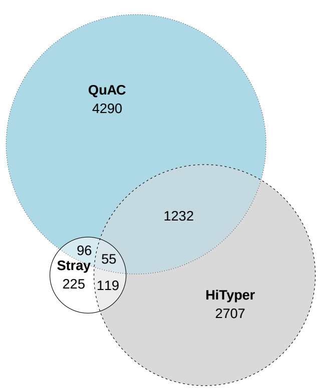

Fig. 2. Typing slots on which each technique makes errorless non-trivial type predictions over all benchmarks.

Repository	S, HT	S, QuAC	HT, QuAC
requests	0.00	0.00	0.29
Pygments	0.02	0.01	0.32
boto3	0.09	0.07	0.23
gunicorn	0.06	0.05	0.15
python-dateutil	0.00	0.00	0.00
pytz	0.02	0.00	0.32
six	0.00	0.00	0.00
pytest-cov	0.00	0.00	0.26
notebook	0.00	0.00	0.30
peewee	0.00	0.00	0.00
seaborn	0.05	0.03	0.22

Fig. 3. Intersection over union between typing slots with errorless non-trivial type predictions, per pair of techniques and benchmark.

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 stark contrast with Stray and HiTyper, which fail to generate *any* errorless 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 Are QuAC's predictions complementary to those of our baselines? Continuing on this note, we further investigate whether QuAC's type predictions *complement* those made by the baselines. In particular, we look at whether QuAC, Stray, and HiTyper make correct type predictions for the *same* or *different* typing slots. The results over all benchmarks are shown in the Euler diagram in Fig. 2. In Fig. 3, we break down the results per benchmark, showing the Jaccard Index (intersection over union) of the sets of errorless non-trivial typing slot prediction for each pair of techniques.

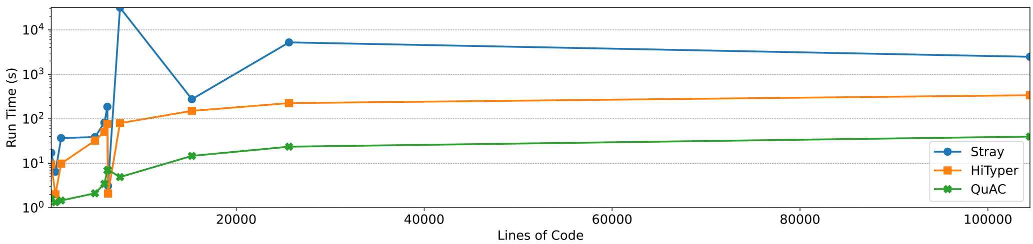

Fig. 4. **Log scale** run time of each technique (y-axis) plotted against lines of code in each benchmark (x-axis).

Both the Euler diagram showing all typing slots and the per-benchmark Jaccard Indices tell a story of there being little overlap between the typing slots where QuAC, Stray, and HiTyper make correct, non-trivial type predictions. Per-benchmark, the highest intersection over union is of 0.32, between HiTyper and QuAC. 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 (but not the baselines) 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 prediction methods.

6.4.6 How does the run time of QuAC compare against the baselines? We now investigate the performance of QuAC and the baselines. Table 5 presents the run times of each method on each benchmark, and Figure 4 plots the run time of each technique against the lines of code in each benchmark. We can see that QuAC outperforms Stray and HiTyper on all but one benchmark and improves 1-2 orders of magnitude on many benchmarks, and on all benchmarks exceeding 7,500 LoC. On the benchmark where QuAC is slower, peewee, QuAC takes 7 seconds to run and infers 726 non-trivial type hints; the baselines run in 2-3 seconds, but infer no non-trivial type hints. Overall, the trend in Figure 4 shows that QuAC's simple design makes it much more scalable in terms of project size compared with our baselines.

6.4.7 What are the main failure modes of QuAC and the baselines? Finally, we investigate the main failure modes of these type prediction 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 6.

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

```python
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 `str`s should be of type `str`. Second, Stray sometimes errs with container type parameter semantics. For example, given that Stray predicts the class of a parameter `d` as `dict` and infers the result of `d.get('path')` as

Table 6. Failure modes of each technique. For each method, we took the five most error-prone typing slots for each benchmark, and categorized the reasons for type prediction failure as below.

	Occurrences		
Failure Mode	S	HT	QuAC
Predictions Lacking Accessed Attributes	10	26	
Built-in and Standard Library Constraints	6		
Over-constrained Predictions		9	
Over-reliance on Parameter Default Values		6	
Union Types	1	1	13
Variable Changing Type			8
Class Query Database			7
Query Algorithm			6
Sparse, Generic Attributes			5
Complex Python Operational Semantics			4
Attribute Types	1		3
Instance Variables			3

`typing.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 as input.

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 on 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]
    # ...
```

```python
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).

```python
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 code, 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.

```python
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 runtime 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 inferring 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 infers 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)
9     # ...
```

7 DISCUSSION

7.1 Future Research Directions

Based on the failure modes discussed above, we believe 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 static analysis procedures that provide better support for the nooks and crannies of Python's semantics and would be beneficial at 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 an Ensemble Method. Another interesting future research direction would be to include QuAC in an ensemble complementing other type prediction 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 prediction methods (Section 6.4.4), and its correct typing slots complement baseline approaches (Section 6.4.5). Moreover, this enables utilizing machine learning-based type inference models to leverage natural language cues and overcome QuAC's *Sparse, Generic Attributes* failure mode.

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 our baseline methods, Stray [Sun et al. 2022] and HiTyper [Peng et al. 2022]. We adopt a modular, functional coding style when developing QuAC and unit-test QuAC's components. Moreover, the implementations of both 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 datasets used in the evaluation. To reduce the threat, in terms of the baselines, we have used the state-of-the-art 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. We did not evaluate the recent machine learning technique DLInfer [Yan et al. 2023], as it can only infer 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 dataset, we compiled a dataset in Section 6.2 consisting of several popular real-world untyped projects spanning different domains and having vastly different project sizes that reduce the threat of selection bias. Moreover, selecting untyped instead of typed projects follows the approach of a recent evaluation of TypeScript type prediction methods [Yee and Guha 2023]. It is justified as our motivating problem is not to recover type annotations for Python programs that already type check, but to migrate untyped Python programs to type-annotated Python, similar to the problem targeted by [Yee and Guha 2023].

The threats to *construct validity* may come from our "correctness modulo type checker" criteria used on untyped Python benchmarks. To mitigate, we have adopted the well-justified approaches proposed in [Allamanis et al. 2020] and [Yee and Guha 2023]. To prevent the effect of trivial type annotations such as `typing.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.

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 very 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 infer 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 cannot guarantee type correctness (failing our first criterion), 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 (rare), leading to a pronounced drop in accuracy for those outlier types (affecting the first criterion) [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. Finally, DLInfer [Yan et al. 2023] collects *slice statements* for variables and uses a sequence model to predict types. 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. Moreover, validation can filter invalid types out but cannot correct them, leading to potential drops in coverage.

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 to perform 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, and thus does not suffer 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 models 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, given that QuAC's correct type predictions complement those made by Stray and HiTyper in Section 6.4.5, including QuAC in an ensemble with machine learning methods 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-typed nature. 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 runtime performance, as demonstrated by our experimental results on popular untyped Python projects. Moreover, QuAC also excels in predicting container type parameters and rare, non-builtin types, demonstrating great potential in synergistically complementing existing type inference methods.

DATA-AVAILABILITY STATEMENT

The code supporting Sections 5 and 6, anonymized for review, is available on https://anonymous.4open.science/r/quac-0C64. After the review process, we will include the benchmarks and data replication scripts and submit the code for evaluation by the artifact submission deadline.

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

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*. 91–105.

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), 61–73.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *ECOOP 2005-Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings 19*. Springer, 428–452.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2017. Migrating gradual types. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

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* 3, POPL (2019), 1–32.

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. *ACM SIGPLAN Notices* 51, 10 (2016), 410–429.

Santanu Kumar Dash, Miltiadis Allamanis, and Earl T Barr. 2018. Refinym: Using names to refine types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–117.

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).

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*. 1859–1866.

Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769.

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*. 303–315.

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 Trans. Softw. Eng. Methodol.* (mar 2024). https://doi.org/10.1145/3652153 Just Accepted.

Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*. Springer, 12–19.

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 152–162.

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 *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*. Springer, 238–255.

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*. 1483–1486.

Kevin Jesse, Premkumar T Devanbu, and Anand Sawant. 2022. Learning to predict user-defined types. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1508–1522.

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*. 112–125.

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)* 53, 3 (2020), 1–38.

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* 5, 1 (2012), 1.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

Christopher D Manning. 2008. *Introduction to information retrieval*. Syngress Publishing,.

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* 4, POPL (2019), 1–29.

Nevena Milojkovic, Mohammad Ghafari, and Oscar Nierstrasz. 2017. It's duck (typing) season!. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 312–315.

Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 585–589.

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*. 2241–2252.

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic type inference for gradual Hindley–Milner typing. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

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*. 2019–2030.

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based gradual type migration. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

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*. 209–220.

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* (Virtual, USA) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3426422.3426981

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The ins and outs of gradual type inference. *ACM SIGPLAN Notices* 47, 1 (2012), 481–494.

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*. 1–12.

Stephen Robertson. 2004. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of documentation* 60, 5 (2004), 503–520.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

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

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 2008 symposium on Dynamic languages*. 1–12.

Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. 2022. Static Type Recommendation for Python. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

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*. 45–56.

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

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).

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 foundations of software engineering*. 607–618.

Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2009–2021.

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

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 5 April 2024