Type-check Python Programs with a Union Type System

Senxi Li Tetsuro Yamazaki Shigeru Chiba

We propose that a simple type system and type inference can type most of the Python programs with an empirical study on real-world code. Static typing has been paid great attention in the Python language with its growing popularity. At the same time, what type system should be adopted by a static checker is quite a challenging task because of Python's dynamic nature. To exmine whether a simple type system and type inference can type most of the Python programs, we collected 806 Python repositories on GitHub and present a preliminary evaluation over the results. Our results reveal that most of the real-world Python programs can be typed by an existing, gradual union type system. We discovered that 82.4% of the expressions can be well typed under certain assumptions and conditions. Besides, expressions of a union type are around 3%, which indicates that most of Python programs use simple, literal types. Such preliminary discovery progressively reveals that our proposal is fairly feasible.

1 Introduction

Static typing has been paid great attention in the Python language with its growing popularity. At the same time, what type system should be adopted by a static checker is quite a challenging task because of Python's dynamic nature. There have been sufficient existing works on static typing for Python. Mypy [5], one of the most famed static type checkers in Python, use type annotations introduced from Python 3.0 and statically typecheck programs by giving semantics to the annotations. Mypy has a powerful type system with features such as bidirectional type inference and generics, while the effectiveness of the type system has not been empirically studied. Other works such as Pytype implement a strong type inferencer instead of enforcing type annotations.

As a dynamically typed language, Python excels at rapid prototyping and its avail of metaprogramming and reflection. Pythonists can also customize their own metaclasses and create classes at runtime. Beside, programmers can introspect or further create, modify Python objects dynamically in any sense. However, these powerful features would be nightmare for static checkers. Modifying a class attribute with `eval('Rect.setColor("red")')` at runtime will not provide any type information to a static checker. Therefore, the checker will complain and raise an attribute error if the program inquires this attibute by accessing `Rect.color`, or will just throw an *unknown* type if the type system is gradual. The operations can hardly provide any type information before runtime and thus a checker will fail to typecheck that piece of code. Consequently,

*Type-check Python Programs with a Union Type System

Senxi Li, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, University of Tokyo.

it is impossible for any exising type system to typecheck all Python programs at a theoretical level.

While such discussion can easily drive us to an end that no existing type system can fully typecheck all Python programs, the following assumptions play a substantial role in our motivation:

- Python programs seems easy. Being a popular programming language used by people from web developers to data scientists, most of the Python programs shall be easy to read and share. As another aspect, machine learning tasks take a significant role in Python applications in recent years. In spite of their indeed complicated algorithms, it can be smoothly imagined that simple data types can handle their code and many of the expressions are just literals like numbers and lists. Initializing a tensor with a value of `List[float]`, function of mathemetical multiplication over matrix and vector being typed as `List[List[float]]` → `List[float]` → `List[float]` and so on may serve as evidence for our assumption.
- Programmers are competent and well-educated. This hypothesis is heavily rooted in the *competent programmer hypothesis* [1], which states that most software faults introduced by experienced programmers are due to small syntactic errors. Furthermore, the majority of Python programmers are believed to have a background of some statically typed languages. In other words, well-educated programmers write programs that may obey a type system they are familiar with implicitly, even Python syntatically never impedes them writing "untypable" code. Accordingly, a simple type system for objects can type most of such Python programs.

These concerns and assumptions lead us to our research question: Is there an exising type system able to type most of real-world Python programs? To answer the question, we collected Python repositories from GitHub and analyzed their source code with a static checker driven by a designed, simple gradual union type system. We believe that union type can handle abstraction and polymorphism within bounds, which can meet the damand of real-world Python code. In addition, we apply a type inference approach that infers function and method types by type information from runtime. While runtime can only tell properties of objects under limited conditions, we claim that variables, functions and objects can be correctly inferred and typed if runtime can consistently provide their types.

Our preliminary result shows that our designed, simple type system and type inference algorithms can typecheck most of the Python programs. The investigation indicates that 82.4% of the expressions in the source code are well typed under certain conditions. Besides, only 3.1% of the expressions are of union type. This paper also involves further evaluation to better analyze if such a type system can satisfy Python's need.

This paper proceeds in the following manner. Section 2 discuesses our motivation and briefly reviews benefits of static typing over dynamic typing. Section 3 proposes a pseudo language and by describing its core calculus we further discuss how our system performs type checking. Section 4 shows our empirical study of real-world Python programs and its results will help answer the research questions that we address. Section 5 mentions related works and a conclusion with directions for future work ends this paper.

2 Motivation

There have been many complex type systems and type inference algorithms proposed for Python in order to statically typecheck programs. However, they are not successful because of the dynamic na-

ture of Python. Many of the existing works apply strong type systems such as callable type to type functions/methods and generics to abstract objects more precisely. Type inference algorithms infer types with hints avaiable in the source code. In the following of this section, we will illustrate how some of existing type systems can and cannot type Python programs with examples.

One of the primary benefits of static typing over dynamic typing is that it helps localize errors. For example, suppose a programmer misunderstands the use of a library function, such as a `getSum` function in a module `arith` which expects one argument of a list of integers but instead the programmer passes in a list of strings.

```
1 from arith import getSum
2
3 lst = ["sum", "the", "list"]
4 getSum(lst)
```

In a purely dynamically typed language, calling the function in such a way will cause a runtime error. Perhaps such runtime error will occur deep inside the body of the function in `arith`, which contains operations that can be performed on numbers but not strings. On the other hand, if the programmer make use of a static type checker, the error can be detected and caught before the call to `getSum`. The following shows that the parameter and return of function `getSum` is annotated using type annotations.

```
1 from typing import List
2
3 def getSum(argList: List[int]) -> int:
4   ...
```

The parameter of the function is well annotated with a generic list type so that a static checker can understand what type of elements a list should hold. By documenting and enforcing type constrains on the definition of `getSum`, a static checker can prevent the usage of the function if the given argument conflicts as declared before indeed running the user code.

At this moment it seems that we can type Python programs if a programmer carefully inject type annotations with the usage of a well implemented type checker. However, the situation will not let us stay optimistic as long as we are engaging with Python. As a powerful dynamic language, Python has a strong functionality on introspection and reflection, enabling programmers to introspect or further modify objects at runtime. Consider the following piece of code:

```
1 class Rect:
2   pass
3
4 def touchColor(arg: Rect) -> str:
5   exec("arg.color = 'red'")
6   return arg.color
```

Though this program can be seriously complained about the bad attribute usage, the given code is absolutely valid in Python. In the body of function `touchColor`, the behavior of the argument `arg` is modified by the builtin function `exec` such that a string is assigned to an attribute `color` of `arg`. Calling this function by passing in an instance of class `Rect` at runtime will return `"red"` just as designed. Nevertheless, this piece of code cannot be well typed with a normal type system, or any existing ones even its argument and return are finely annotated. A static checker, such as Mypy [5], will complain that *"Rect" has no attribute "color"*. The confliction stems from the fact that the reflection at line 5 does not render any type information to the checker statically.

3 Core Calculus

Observations and presumptions bring us to the following idea: a simple but well designed type system can type *most of* the Python programs. We choose a simple, gradual union type system which seems powerful enough for typing Python code and by applying the type system, we shall carry out an

empirical study to investigate to what extent real-world Python programs are typeable. To show the core calculus of our designed type system, we define a simple pseudo language `MiniPy` and give its formal calculus. The static semantics of `MiniPy` is almost a subset of the Gradual Typed Object Calculus defined in [8]. For simplicity, `MiniPy` does not have statements and only contains part of the expressions in Python. Meanwhile, the typing rules and type checking of `MiniPy` are the core part of that of our developed type system.

The type system of `MiniPy` is a gradual, union type system. The purpose of choosing the gradual typing, which allows the mixture of static and dynamic typing, is to distinguish the typeable and untypeable with the power of *the unknown type*. We use the unknown type to mark the parts of the program that cannot be typed as dynamically typed, and use the other types in the type system to type those typeable parts.

In the following subsections we first give the syntax and typing rules of `MiniPy` and then show how the type system typechecks programs.

3.1 Syntax

The syntax of `MiniPy` is given as follows:

```
CL ::= class C(C): {M} [class]

M ::= def m(p):  return e; [method]

e ::= x | C() | e.f | e(e) [expressions]
```

The metavariables are listed as follows: `C`, `D` range over class names; `M` ranges over method declarations; `f` ranges over fields; `m` ranges over methods; `x` ranges over variables; `e` ranges over expressions;

In `MiniPy`, we can define a class with the class definition `CL`. A class definition `class C(D): {M}` introduces a class named `C` inheriting a superclass `D`. The body of the class definition is a single method declaration `M`. The body of the method is a single `return` statement. Expressions `e` includes some of the primary expressions in Python. `C()` refers to instance initialization, which will create an instance of a class `C`. `e.f` refers to field access. `e(e)` suggests an invocation. Since there is no function definition in `MiniPy`, `e(e)` can only be method invocation.

3.2 Subtyping

Now let us define the types, denoted as follows:

$$\texttt{T ::= ? } \mid \texttt{ C} \mid \texttt{ T} \rightarrow \texttt{ T} \mid \texttt{T} \vee \texttt{ T}$$

`T`, `S` range over types; `?` refers to the unknown type in a gradual type system.

As the types suggest, our type system is gradual, which mixs static and dynamic type checking by introducing the unknown type `?`. Gradual typing integrates *the unknown type* `?` and *type consistency* into an existing static type system in order to support fully dynamic type checking, static type checking, and any point on the continuum. [3]

Besides basic types and function type, the type system allows union type. A union type, more precisely, an untagged union, noted $\texttt{T}_1 \vee \texttt{T}_2$, denotes the value of such a type can be either $\texttt{T}_1$ or $\texttt{T}_2$ at runtime. In the set-theoretic interpretation for union type by Frisch [2], for example, `Int` $\vee$ `Int` is the same type as `Int`; a value of `Int` can be a value of `Int` $\vee$ `String`. Union type can be used to type methods whose argument shares some same properties. For instance, a method declaration `def m(x): return x.f;` can be well typed as type $(\texttt{T}_1 \vee \texttt{T}_2) \rightarrow (\texttt{S}_1 \vee \texttt{S}_2)$ if both $\texttt{T}_1$ and $\texttt{T}_2$ have the field `f` and their resulting types are $\texttt{S}_1$ and $\texttt{S}_2$, respectively.

Before describing the subtyping rules, let us briefly discuss the unknown type. `?` is completely unknown: we do not know what type it represents; in other words, `?` represents *any* type. For instance, a type `Int` $\rightarrow$ `?` represents a function from `Int` to some unknown type. In other words, this type can be *any* type of function as long as it maps from `Int` to some other type.

While the type consistency relation will be defined in the next subsection, here we present the subtyping rules as the following:

$$\frac{}{\texttt{?} \ \texttt{<:} \ \texttt{?}} \ (\texttt{<:-?})$$

$$\frac{}{\texttt{C} \ \texttt{<:} \ \texttt{C}} \ (\texttt{<:-ID})$$

$$\frac{\texttt{C}_1 \ \texttt{<:} \ \texttt{C}_2 \qquad \texttt{C}_2 \ \texttt{<:} \ \texttt{C}_3}{\texttt{C}_1 \ \texttt{<:} \ \texttt{C}_3} \ (\texttt{<:-TRANS})$$

$$\frac{\texttt{class C(D): \{...\}}}{\texttt{C} \ \texttt{<:} \ \texttt{D}} \ (\texttt{<:-INHER})$$

$$\frac{\texttt{S}_2 \ \texttt{<:} \ \texttt{S}_1 \qquad \texttt{T}_1 \ \texttt{<:} \ \texttt{T}_2}{\texttt{S}_1 \to \texttt{T}_1 \ \texttt{<:} \ \texttt{S}_2 \to \texttt{T}_2} \ (\texttt{<:-}\to)$$

$$\frac{\texttt{S}_1 \ \texttt{<:} \ \texttt{T} \qquad \texttt{S}_2 \ \texttt{<:} \ \texttt{T}}{\texttt{S}_1 \vee \texttt{S}_2 \ \texttt{<:} \ \texttt{T}} \ (\texttt{<:-}\vee\texttt{LEFT})$$

$$\frac{\texttt{S} \ \texttt{<:} \ \texttt{T}_1}{\texttt{S} \ \texttt{<:} \ \texttt{T}_1 \vee \texttt{T}_2} \ (\texttt{<:-}\vee\texttt{RIGHT1})$$

$$\frac{\texttt{S} \ \texttt{<:} \ \texttt{T}_2}{\texttt{S} \ \texttt{<:} \ \texttt{T}_1 \vee \texttt{T}_2} \ (\texttt{<:-}\vee\texttt{RIGHT2})$$

The subtype relation is straightforward to define. Most of the rules just follow a traditional subtyping for objects. A special construct is about the unknown type, which is given by `? <: ?`. A class inheritance simply gives a subtype relation from the subclass to its superclass. `<:-`$\vee$`LEFT` describes that a union type is the subtype of another type if both of its components are the subtype of that type. `<:-`$\vee$`RIGHT1` describes that one type is the subtype of a union type if that type is the subtype of one of the components of the union type. `<:-`$\vee$`RIGHT2` just explains the other case.

3.3 Type Consistency

The intuition behind type consistency is to check whether the two types are equal in the parts where both types are known. A gradual type system uses type consistency, which replaces type equality that a simple type system typically uses. A gradual type check allows an implict conversion between two types if they are *consistent* with each other. We present the consistency relation $\sim$ on types with the following definition.

$$\frac{}{\texttt{C} \sim \texttt{C}} \ (\sim\texttt{-ID})$$

$$\frac{}{\texttt{?} \sim \texttt{T}} \ (\sim\texttt{-?1})$$

$$\frac{}{\texttt{T} \sim \texttt{?}} \ (\sim\texttt{-?2})$$

$$\frac{\texttt{S}_1 \sim \texttt{S}_2 \qquad \texttt{T}_1 \sim \texttt{T}_2}{\texttt{S}_1 \to \texttt{T}_1 \sim \texttt{S}_2 \to \texttt{T}_2} \ (\sim\texttt{-}\to)$$

$$\frac{\texttt{S}_1 \sim \texttt{T}_1 \qquad \texttt{S}_2 \sim \texttt{T}_2}{\texttt{S}_1 \vee \texttt{S}_2 \sim \texttt{T}_1 \vee \texttt{T}_2} \ (\sim\texttt{-}\vee\texttt{1})$$

$$\frac{\texttt{S}_1 \sim \texttt{T}_2 \qquad \texttt{S}_2 \sim \texttt{T}_1}{\texttt{S}_1 \vee \texttt{S}_2 \sim \texttt{T}_1 \vee \texttt{T}_2} \ (\sim\texttt{-}\vee\texttt{2})$$

The definition of type consistency is almost the same as the one described in [8], while our type system involves the union type. `(`$\sim$`-`$\vee$`1)` describes that two union types are consistent if their components are consistent with each other. `(`$\sim$`-`$\vee$`2)` tells the same property with a reverse consistent relation of their components since two union types are identical with the same components even their order are different.

3.4 Consistent-Subtyping

Consistent-Subtyping is defined that takes both type consistency and subtyping into account.

$$\frac{\texttt{S} \ \texttt{<:} \ \texttt{T'} \qquad \texttt{T'} \sim \texttt{T}}{\texttt{S} \sim\texttt{<:} \ \texttt{T}} \ (\sim\texttt{<:})$$

Although this rule still remains some non-determinacy because of the type `T'`. Previous researches have conceived the consistent-subtyping relation in terms of a restriction operator $\texttt{T}_1|_{\texttt{T}_2}$ which masks off the parts of a type $\texttt{T}_1$ that are unknown in a type $\texttt{T}_2$. For space reason we will not explain its formular or a formal definition of the consistent-subtyping relation: all details are given by [8].

3.5 Typing Rules

The typing rules are outlined in this subsection. An environment Γ is a finite mapping from variables to types, written `x: T`.

$$\Gamma \ ::= \varnothing \mid \Gamma\texttt{, x: T}$$

Before illustrating the typing rules, let us define

some operators on types and a lookup function that are necessary to define and reason the typing rules and further the type inference. The two operators on types, especially on function types, are: *(i)* the `dom(.)` operator that given a funciton type `T` returns the domain of that type and *(ii)* the `cod(.)` operator that returns its codomain. While the formal definition of these two operators are omitted for space reason, since the unknown type `?` in our type system can be *any* type (not only basic types), the patterns for `?` proceed as `dom(?) = ?` and `cod(?) = ?`.

The intuition of the lookup function is that we need to be able to identify the type of a field or method upon a given type. Function `field(T, f)` returns the field type of `T` if type `T` holds such a field, and it is defined by:

$$\texttt{field(C, f)} = \texttt{T}$$

$$\texttt{field(?, f)} = \texttt{?}$$

$$\texttt{field}(\texttt{T}_1 \vee \texttt{T}_2\texttt{, f)} = \texttt{field}(\texttt{T}_1\texttt{, f)} \vee \texttt{field}(\texttt{T}_2\texttt{, f)}$$

$$\texttt{field(\_, \_)} = \texttt{undefined}$$

With these operators and the lookup function, the typing rules are given as:

$$\frac{\texttt{x: T} \in \Gamma}{\Gamma \vdash \texttt{x: T}}\ \texttt{(T-VAR)}$$

$$\frac{}{\Gamma \vdash \texttt{C(): C}}\ \texttt{(T-INST)}$$

$$\frac{\Gamma \vdash \texttt{e: } \texttt{T}_0 \qquad \texttt{field}(\texttt{T}_0\texttt{, f) = T}_1}{\Gamma \vdash \texttt{e.f: } \texttt{T}_1}\ \texttt{(T-FIELD)}$$

$$\frac{\Gamma \vdash \texttt{m: T} \qquad \Gamma \vdash \texttt{e: S} \qquad \texttt{S} \sim\texttt{<: dom(T)}}{\Gamma \vdash \texttt{m(e): cod(T)}}\ \texttt{(T-INVK)}$$

The type judegement for expressions is almost a subset of the one provided by Siek and Taha [Gradual Typing for Objects]. Since our type system is gradual, `T-INVK` will judge not the subtype relation `<:` between the argument and the parameter, but their consistent-subtyping relation $\sim$`<:`.

4 Can the Union Type System Type Real-World Python Programs?

In this section we will describe the empirical study that examines whether our simple type system can type most expressions in real-world Python programs. Detailed construction of the experiment will be illustrated and a preliminary result, though still under rough analysis currently, will be given to progressively show to what extent expressions in real-world Python programs are typeable under our designed type system.

To build our dataset, we collected 806 Python repositories from Github. Our dataset contains 806 repositories, 101,442 Python files and 11,367,317 lines of code in total. Since we didn't apply any bias while searching repositories, which can indicate that the applications of the collected repositories are of various fields. Algorithm 1 shows pseudocode for the overview of our experiment setup and how we typecheck programs and detect static type errors from a given set of repositories.

Algorithm 1: overview

```
Data: Repositories ← set of Python
      repositories
Result: report of type errors
for repo ∈ Repositories do
    run unit tests in repo;
    collect runtime type information;
    database ← collected runtime type
     information;
    for .py file ∈ repo do
        code ← content of .py file;
        typechecker(code, database);
    end
end
```

In a word, for a given set of repositories we first

run their test code if there is. During the execution, runtime type information is collected and recorded into database. After that, we extract all valid Python files inside each repositories and then use a type checker driven by the type system we presented in Section 3 to typecheck those Python files. Finally, a report of static type errors detected by the checker is carried out.

In the following subsections, we explain how we conduct each step of the experiment in detail, analyze the results and then answer the research question we addressed.

4.1 Collecting Runtime Type Information

As the first step of the whole experiment, for each repository we try to retrive runtime type information to help type inference in the type checking step. Our system infers function and method types with the type information obtained at runtime. While runtime can only tell properties of objects under limited conditions, if collected runtime types during the execution can give consistent types to all variables, expressions and functions in our type system, then we claim that these variables, expressions and functions can be typed.

Yet running the application is a far more complicated task, fortunately unit test releases us so that we can easily run the application. Unit test is a software testing to validate that each unit of the software performs as designed, quite lightweighted but able to provide rich runtime information. Regarded as an alternative of static typing, unit test is widely accepted and used by Python programmers. While not all of the repositories in our dataset are equipped with well-written unit tests, for those repositories which have unit tests, we invoke their tests by `pytest` [6], which is one of the most used Python testing tool to build tests.

During the execution of unit tests, we inject scripts which use the `settrace` function from the builtin `sys` module to interpose on function calls and method invocations. Furthermore, the arguments and returns of the traced callables are profiled so that the data type of the objects are obtained and they are treated as the runtime type information of the corresponding functions/methods. Additionally, builtin container types such as *list* and *tuple* are recorded in generic types, i.e., the data type of `[1, 2, 3]` will be profiled as `List[Int]`, while that of `[42, 3.14, 'tokyo']` will be profiled as `List[?]`.

For example, consider the following test code:

```
1 from PhoneMod import Phone
2
3 def test_phonecall():
4   phone = Phone()
5   phone.call(123456)
6   phone.call("police")
```

Class `Phone`, from a module `PhoneMod`, has a method `call` and its instance invokes this method with an integer and a string, respectively, and both will return `None`. When this piece of test code is executed, method invocations are traced and their arguments and returns are profiled. In other words, the runtime type information of these two method invocation will be observed as `Phone` $\rightarrow$ `Int` $\rightarrow$ `None` and `Phone` $\rightarrow$ `Str` $\rightarrow$ `None`. Since the method is invoked by the arguments with two different types, these types will be merged into a union type and therefore the type of method `call` will be inferred as `Phone` $\rightarrow$ `Int` $\vee$ `Str` $\rightarrow$ `None`.

As we mentioned before, unit test can only test functions/methods under limited conditions, thus our type inference from runtime may fail to infer function/method types if not enough runtime information. However, if the given type information in the unit test can consistently type the tested functions, such type inference can be practically feasible. Hence, in the following type checking and further analysis, we keep an optimistic attitude that

the collected runtime type information can give consistent types to functions/methods and assume that the inferred function/method types are typed correctly.

4.2 Type Checker

After collecting the runtime type information, a static type checker is invoked to typecheck Python programs under each repository. The target programs are all valid `.py` files (an unvalid example could be `setup.py` file which is a special named script file for Python package management). We implement the static type checker in Python, parsing a `.py` file and applying the typing rules described in Section 3.5 for typechecking. For example, following `T-INVK` defined in Section 3.5, function calls and method invocations are checked by consistent-subtyping relation between the parameters and the given arguments. For example, `List[?]` and `List[Car]` are of consistent-subtyping relation, while `List[Person]` and `List[Car]` are not.

```
1  from typing import Any, List
2  from mod import Car, Person
3
4  def runCars(arg: List[Car]) -> None:
5    pass
6
7  lst1: List[Any]
8  lst2: List[Person]
9  runCars(lst1) # OK
10 runCars(lst2) # Error!
```

In the above example, the function call at line 9 will be totally fine while the call at line 10 will cause a type error raised by the checker.

Since the calculus described in Section 3 shows only the core part of the Python language, one of the uncovered typing rules is the flow-sensitive typing in the control flow. An example can be:

```
1  x = "string" if condition else 42
```

In the above code, variable `x` can be a string or integer at runtime according to variable `condition` which has the type `Bool`. Flow-sensitive typing is applied (also being applied to `If` statement and other branch statements in Python) following the semantics of the `If` expression. Thanks to our union type system, variable `x` can be well typed as `Str` $\vee$ `Int`.

4.3 Expressions

In the following subsection, we will investigate how expressions in the source code are typed in order to better understand how our designed type system can typecheck programs. Here an expression is defined as *typed* if the checker evaluates or infers it into a type that is neither type error nor the unknown type `?`. For instance, an expression of type `Int` is judged as *typed*; type `(?` $\rightarrow$ `Float)` $\vee$ `?` is *typed*; `?` is not *typed*.

The expressions we will evaluate are all Python expressions recognized by a parser. As a tiny example to show how count the expressions, consider the following piece of code

```
1  def fib(n):
2    if n == 0:
3      return 0
4    elif n == 1:
5      return 1
6    else:
7      return fib(n-1) + fib(n-2)
```

which shows the definition of a function computing the Fibonacci number. The number of expressions inside the body of `fib` is counted as 16. Part of the expressions can be `n`, `1`, `n-1`, `fib(n-1)` and so on at line 5.

The results show that there are totally 14,966,689 expressions in all repositories. 12,333,335 (82.4%) of the expressions are *typed*. The remaining expressions are 409,524 detected errors (1.4%) and 2,223,830 the unknown type `?` (14.9%). The results reveal that most of the expressions in the real-world Python programs can be well typed with static types under a gradual type system. Since our system is still under development, the detected static errors are believed to be *false positive* ones,

which are errors convicted by the checker but being innocent at runtime.

4.4 Categorizing Unions

Here we want to focus on how many expressions are evaluated as union type according to our type system. As shown in Fig. 1, the majority, totally 452,840, of the union types has the size of 2, which is 95.7% over all the union types. Additionally, only 0.19% (896 over 473,312) of the union types have the size larger than 10 and the largest union size reaches 49. These results suggest that most of the objects can be handled with simple data types such as union type with commputable sizes and the programs can be reasonably typed by a union type system.

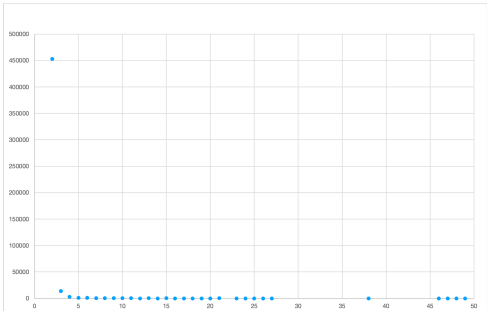

Fig.1 Distribution graph of the union types over size. The horizontal axis shows the size of the union type; The vertical axis shows the number

4.4.1 Optional Type

According to our data, 5.06% (22,896 over 452,840) of the union types of size 2 are optional types. Optional type is a quite common polymorphic type that can represents encapsulation of an optional value avaiable in many programming languages. In Python, a value of type `Option[T]` refers to a value which may result in of type `T` or `None`. As comparison, the number of union type of size 3, which is the second most union type of all sizes, is 14,005, which indicates that optional type is more than other union types larger than 2. The discovery reveals that optional type is rather practical in real-world Python programs.

4.4.2 Large Unions

We randomly sampled the larger unions to figure out what kinds of code snippets result in such large unions. Surprisingly, all such unions point to the same object `unittest.TestCase.assertEqual`, even those we sampled are from different repositories. `assertEqual` is a method in the class `TestCase` defined in the builtin library `unittest`, which is unit test framework for constructing and running tests. The method is called to check for an expected result and its "ground truth" signature can be roughly written as $\texttt{T} \to \texttt{T} \to \texttt{Bool}$, where `T` is type variable defined by `T = TypeVar('T')`. The scenario is that users heavily apply this function to test equality for diverse objects of many types $\texttt{T}_1, \ldots, \texttt{T}_n$. Since our type inferencer blindly merges all types into one union for tested functions, multiple invocations of the same function at the test phase will be traced and therefore the type of `unittest.TestCase.assertEqual` will be inferred into type $(\texttt{T}_1 \vee \ldots \vee \texttt{T}_n) \to (\texttt{T}_1 \vee \ldots \vee \texttt{T}_n) \to \texttt{Bool}$, whose argument types are unions with many components.

Such observation can lead us to two meaningful discussions. First, union type can handle most of Python objects. Larger unions only corresponds to a smalle part of the whole and the vast majority of them are the same object from a builtin library. Namely, user defined objects tend to be of literal types or smaller unions like optional type, which indicates that such a union type system can type-check most of the Python programs within bounds. Second, a slightly enhanced type inference algorithm can fluently bring higher accuracy to type checking. Since the large unions all point to the

same object, or large unions are caused by a particular scenario, we can apply additional ad-hoc typing rules upon our type system. A special typing rule can be, for example, unions of size larger than, i.e. 10, are replaced with type parameters so that we can well type highly generic functions.

5 Related Works

There has been enough attempts of static typing for dynamic languages. In this section we will introduce some of them that are targeted at Python.

Mypy [5] is probably the best known static type checker for Python based on gradual typing. It can lift Python programs to a statical level and type check your code by adding type annotations. By distributing type annotations with the programmer's will, Mypy can smoothly mix dynamic and static typing in the program. Our approach is based on type inference so it does not force type annotation.

There exists other works at type checking Python programs with pure inference instead of requiring type annotations. One of the celebrated is Pytype [7], which uses type inference instead of gradual typing. Pytype generates 'false-positive' errors such as a late attribute initialization in a class definition, which from the perspective of the type checker are, but from a programmer's view are not. Alternatively, Pytype allows users to inject specific comments to slicent such warning. Our design employs gradual typing and emphasizes that a light-weight type inferencer can handle most of the Python programs.

6 Conclusion

In this paper we argued that a simple, well designed type system that can mostly type Python programs. An empirical study on the static type checking in Python was given to show a preliminary evident to support our statement. Our analysis quantitatively revealed the potency of a simple, union type system typing real-world Python programs.

Though our research have made progressive accomplishment, more investigation should be carried out to strengthen our statement. Designed as a test bed, it is imperative to investigate how other type systems are qualified for Python programs. Besides, the amount of the dataset is not large enough so that our current assessmemt is not highly convincible. It is also beneficial to explore what specific pieces of code the type sytem is not able to type. All of those studies are our central future work.

参 考 文 献

[1] DeMillo, R., Lipton, R. J., and Sayward, F.: Hints on Test Data Selection: Help for the Practicing Programmer, *Computer*, Vol. 11(1978), pp. 34–41.

[2] Frisch, A., Castagna, G., and Benzaken, V.: Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types, *J. ACM*, Vol. 55, No. 4(2008).

[3] Garcia, R., Clark, A. M., and Tanter, E.: Abstracting Gradual Typing, *SIGPLAN Not.*, Vol. 51, No. 1(2016), pp. 429–442.

[4] MonkeyType: https://github.com/Instagram/MonkeyType.

[5] mypy: http://mypy-lang.org.

[6] pytest: https://docs.pytest.org/en/stable/.

[7] pytype: https://github.com/google/pytype.

[8] Siek, J. and Taha, W.: Gradual Typing for Objects, 08 2007, pp. 2–27.

[9] Siek, J. G.: Gradual Typing for Functional Languages, 2006.